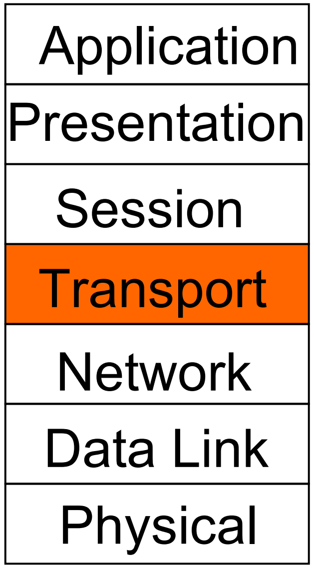
# CSE 461 – Module 11

# Connections

# This Time

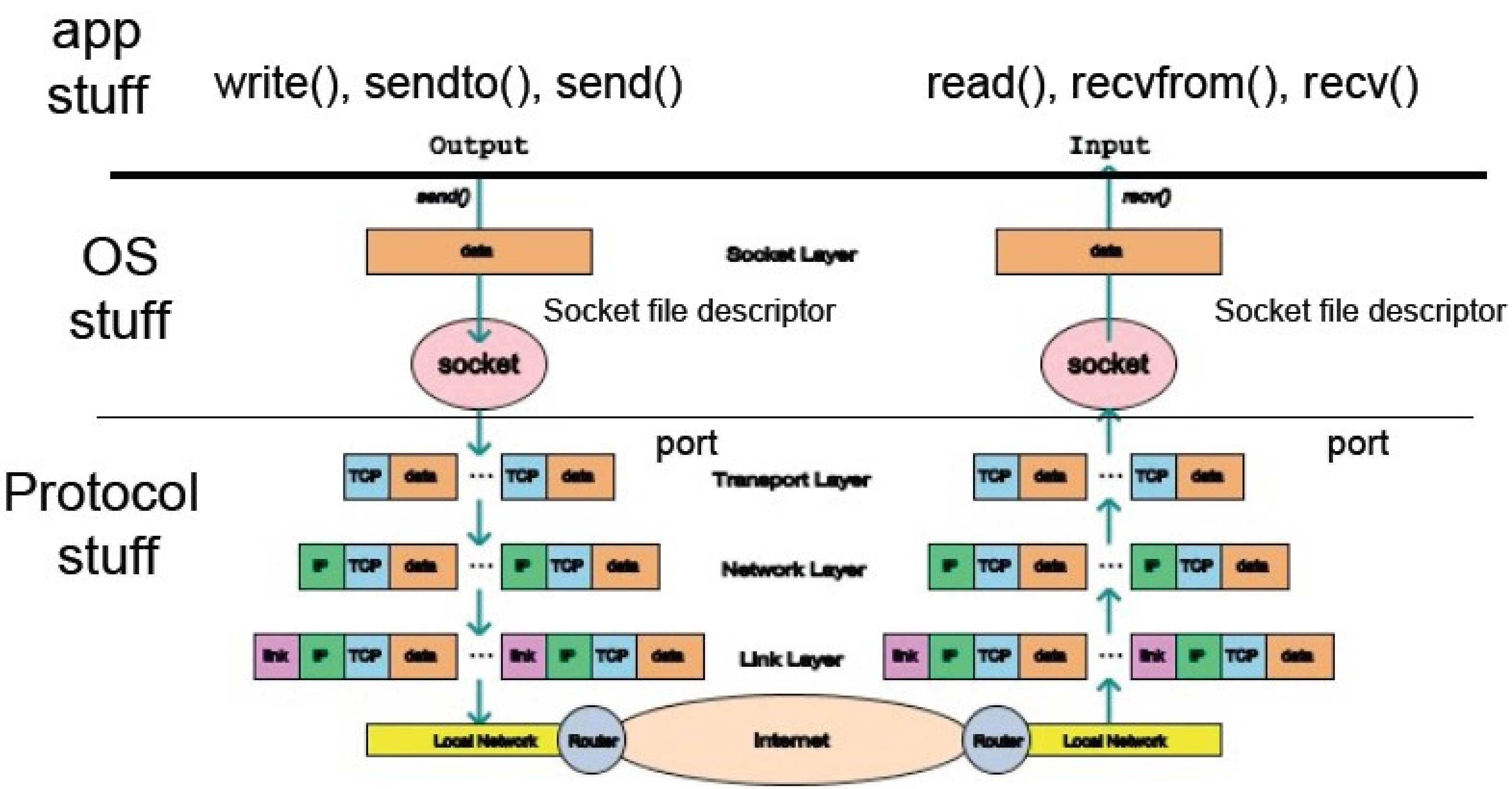
- More on the Transport Layer
- Focus
  - How do we connect processes?
- Topics
  - Naming processes
  - Connection setup / teardown
  - Flow control



# Naming Processes/Services

- Process here is an abstract term for your Web browser (HTTP), Email servers (SMTP), hostname translation (DNS), mp3 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”
    - typically from OS
  - Identify endpoint uniquely as (IP address, protocol, port)
    - OS converts into process-specific channel, like “socket”

# Processes as Endpoints

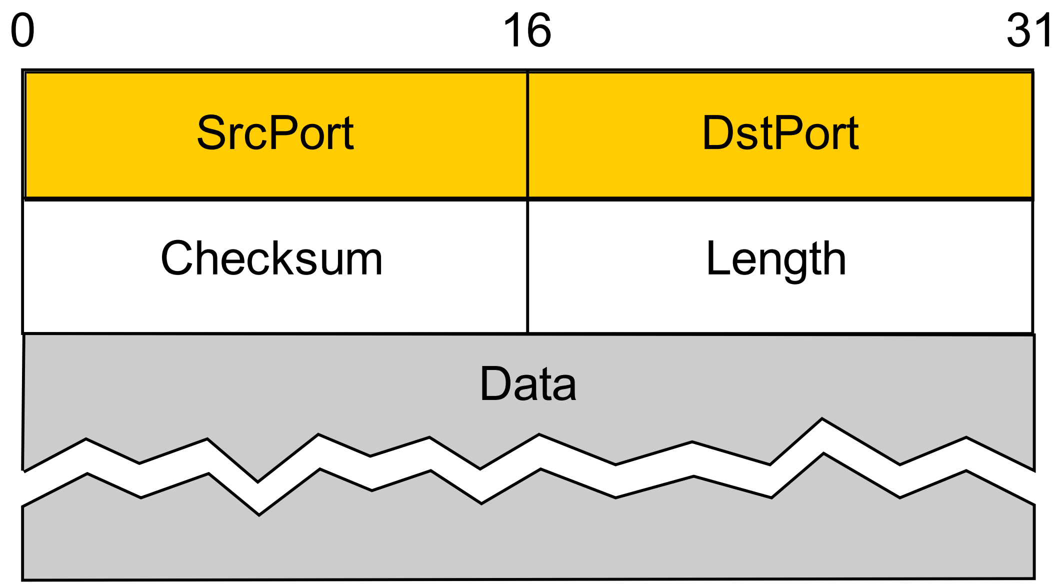


# Picking Port Numbers

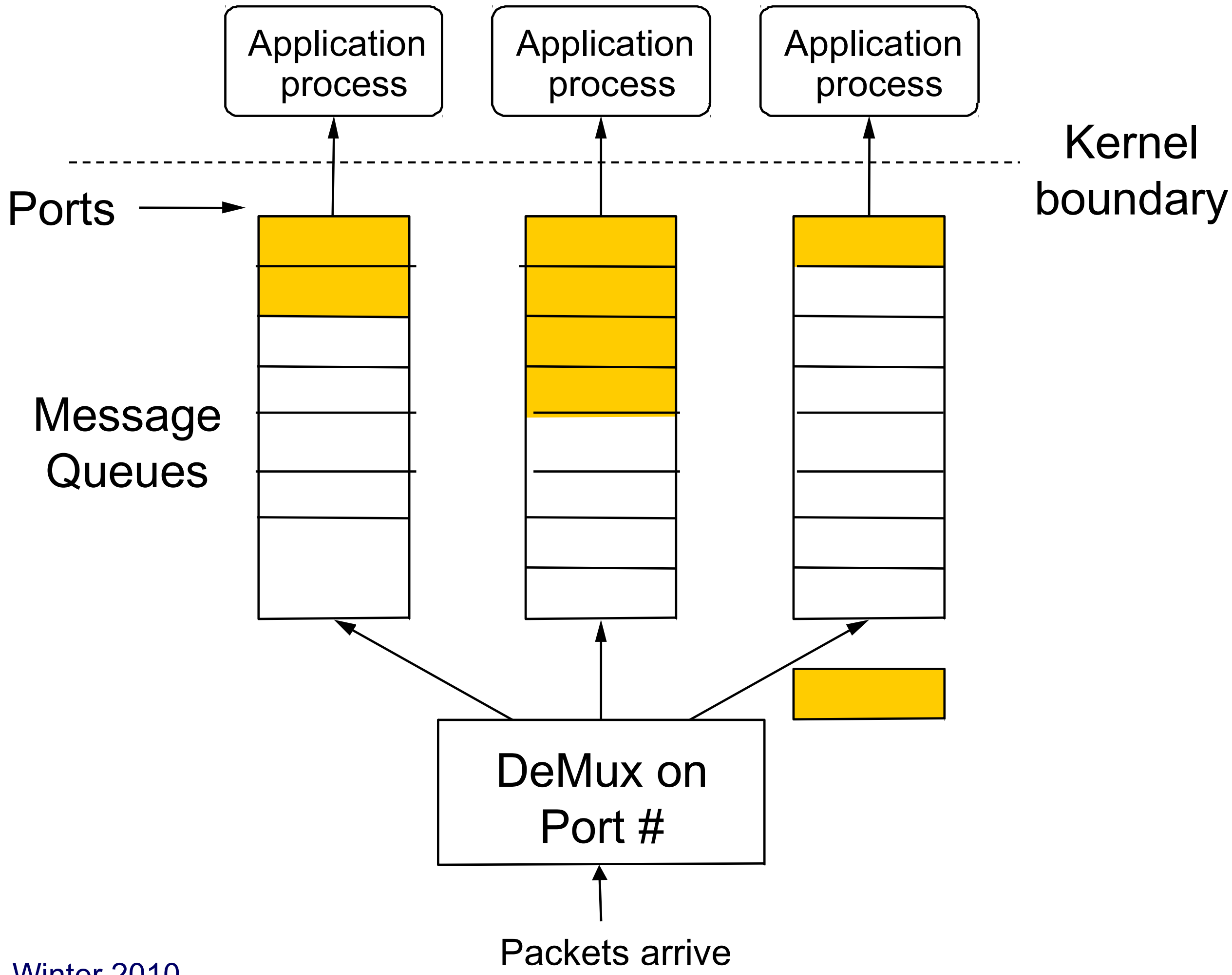
- We still have the problem of allocating port numbers
  - What port should a Web server use on host X?
  - To what port should you send to contact that Web server?
- Servers typically bind to “well-known” port numbers
  - e.g., HTTP 80, SMTP 25, DNS 53, … look in /etc/services
  - Ports below 1024 reserved for “well-known” services
- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished

# 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

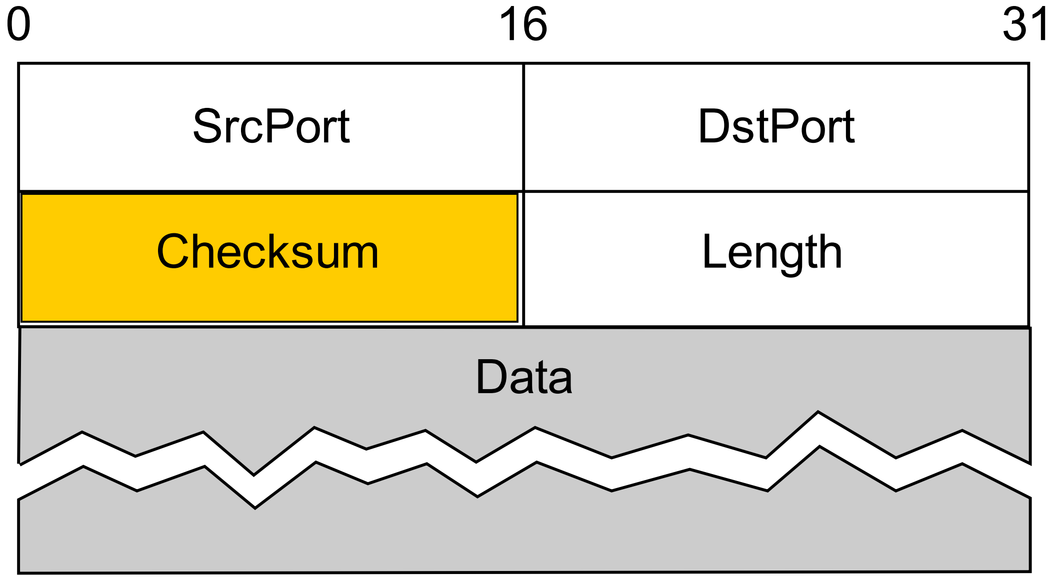


# UDP Delivery



# UDP Checksum

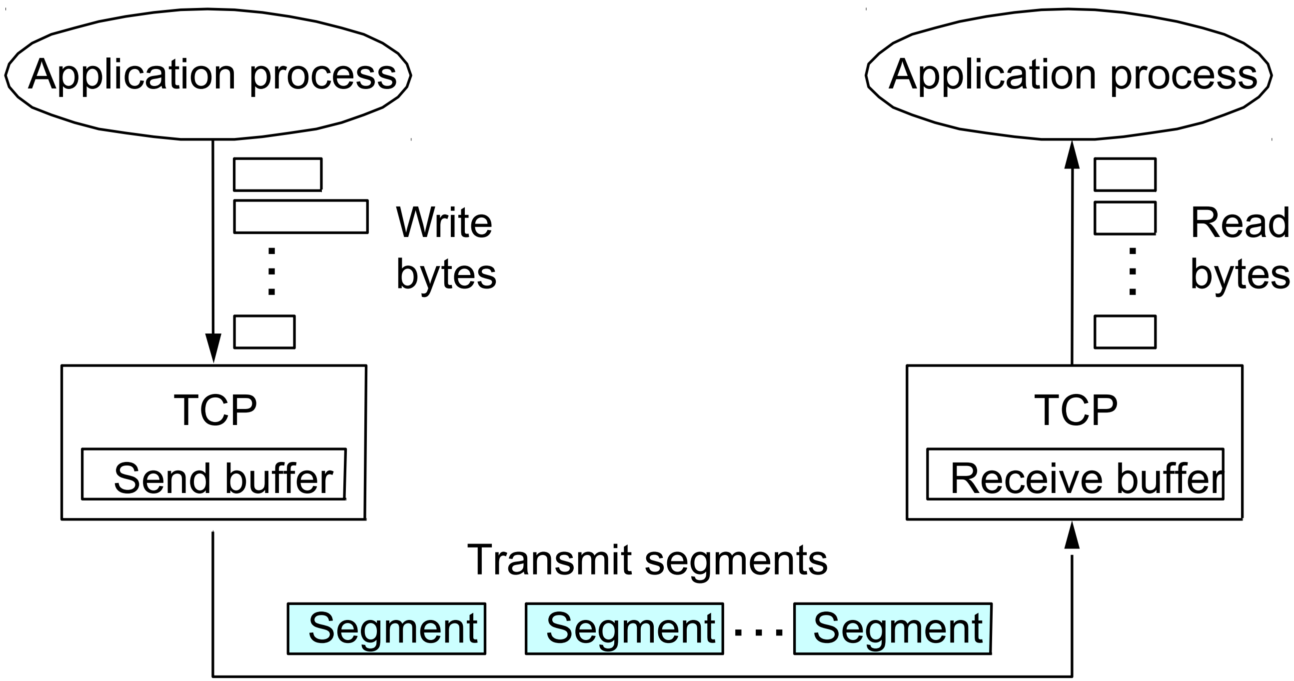
- UDP includes optional protection against errors
  - Checksum intended as an end-to-end check on delivery
  - So it covers data, UDP header, and IP pseudoheader



# Transmission Control Protocol (TCP)

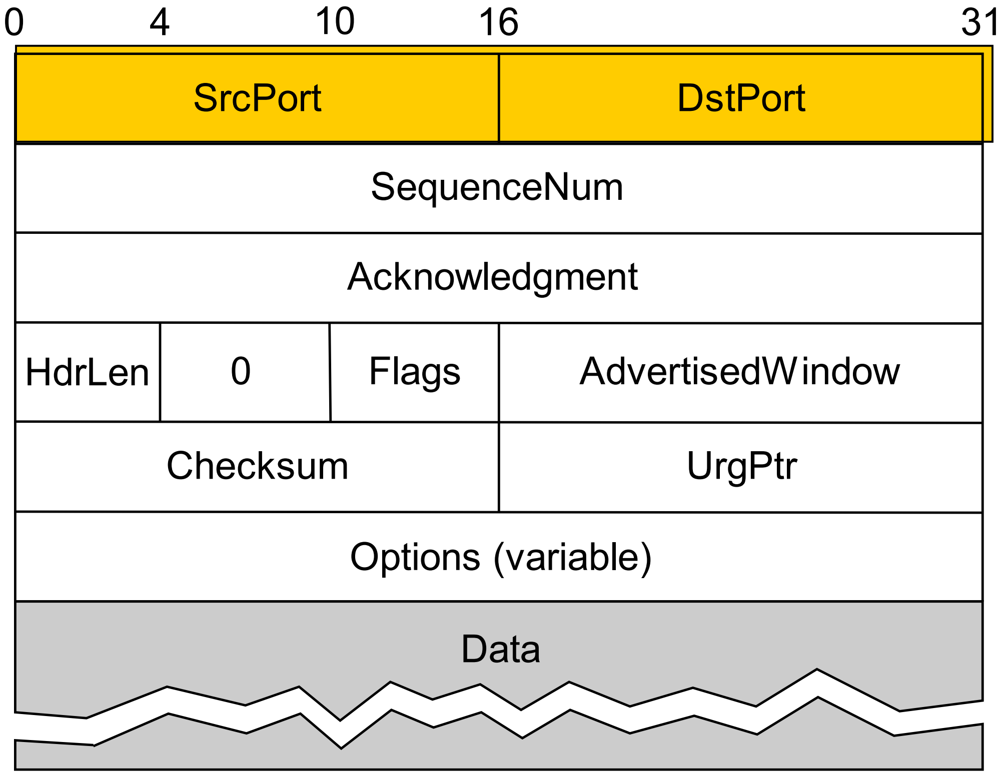
- Reliable bi-directional bytestream between processes
  - Message boundaries are not preserved
- Connections
  - Conversation between endpoints with beginning and end
- Flow control
  - Prevents sender from over-running receiver buffers
- Congestion control
  - Prevents sender from over-running network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

# TCP Delivery



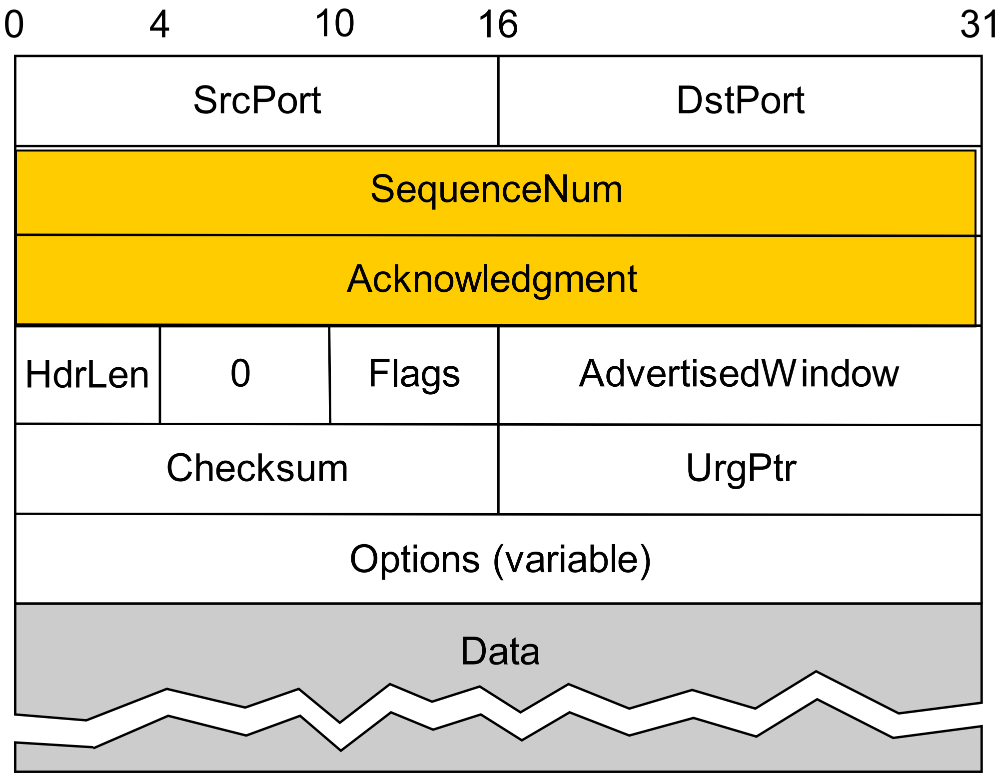
# TCP Header Format

- Ports plus IP addresses identify a connection/flow



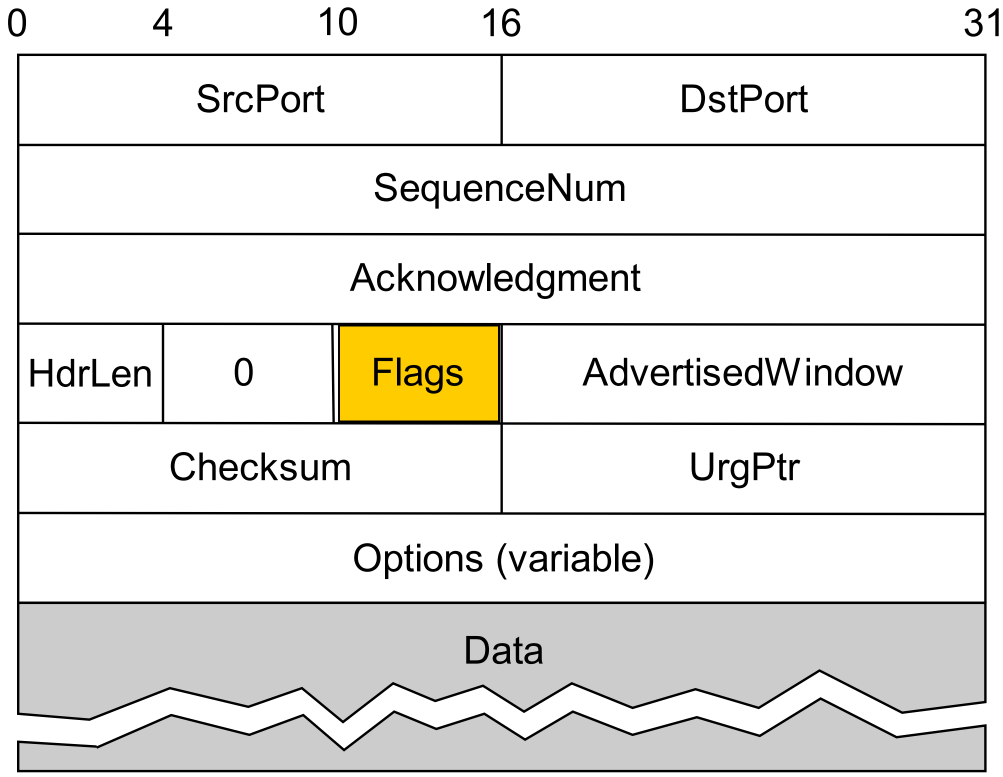
# TCP Header Format

- Sequence, Ack numbers used for the sliding window



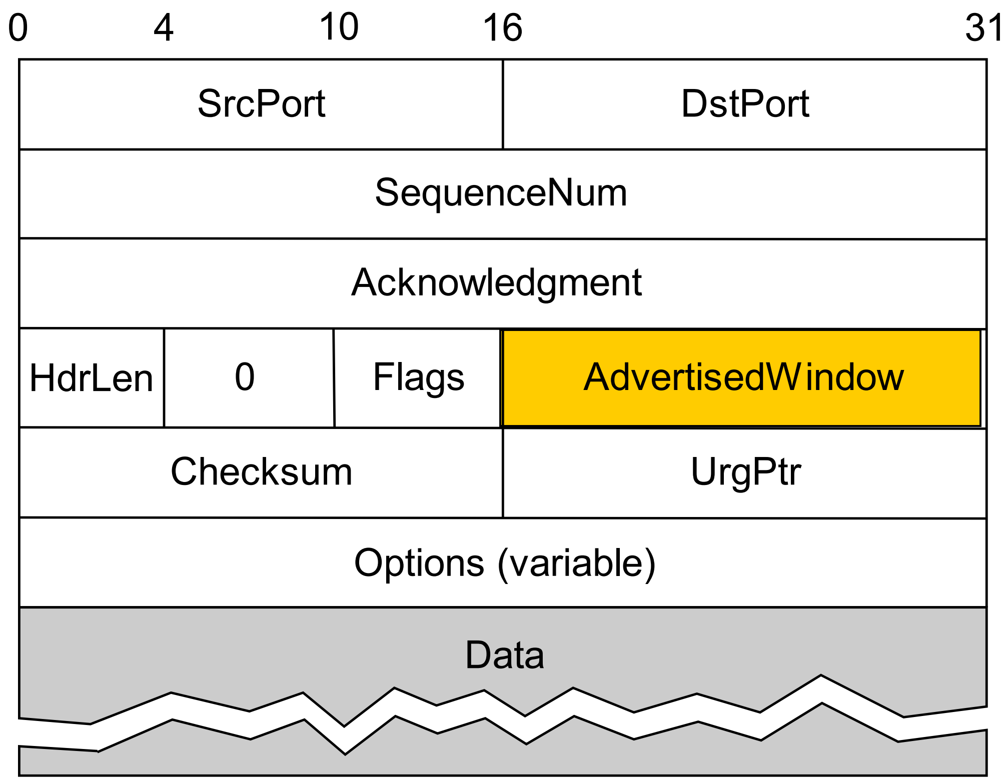
# TCP Header Format

- Flags may be URG, ACK, PUSH, RST, SYN, FIN



# TCP Header Format

- Advertised window is used for flow control



# TCP Connection Establishment

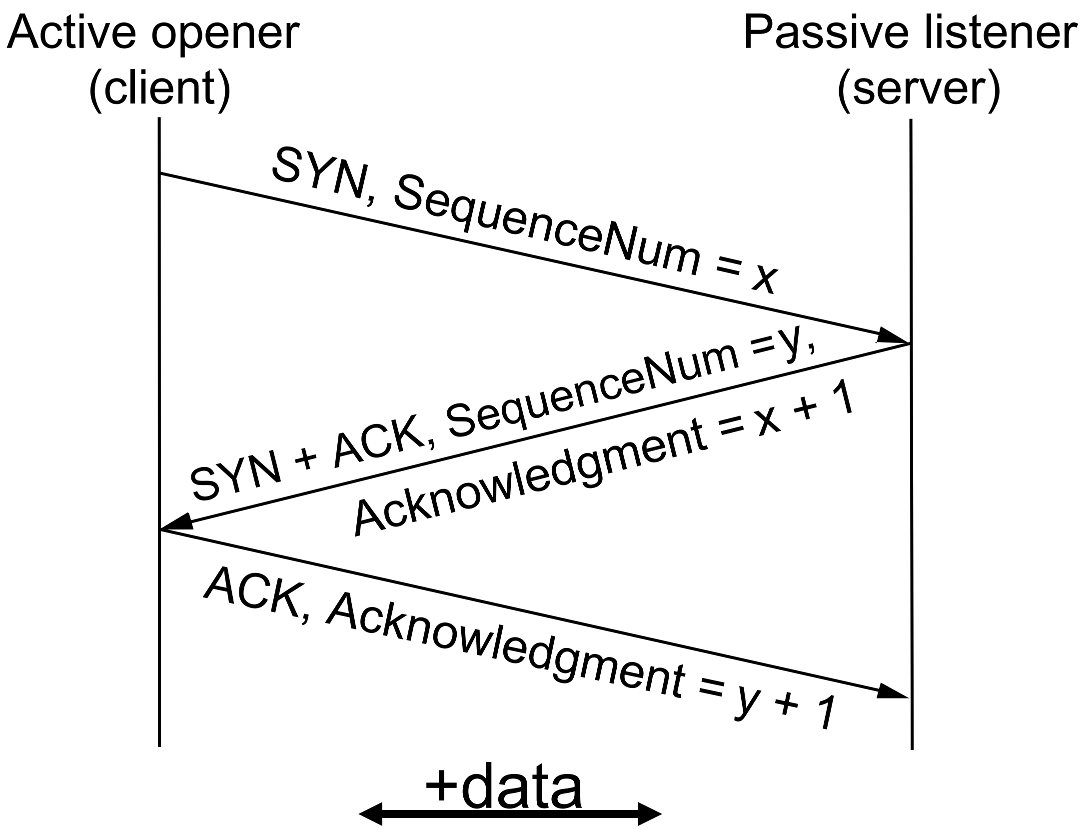
- Both connecting and closing are (slightly) more complicated than you might expect
- That they *can* work is reasonably straightforward
- Harder is what to do when things go wrong
  - TCP SYN+ACK attack
- Close looks a bit complicated because both sides have to close to be done
  - Conceptually, there are two one-way connections
  - Don't want  to hang around forever if other end crashes

# TCP 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

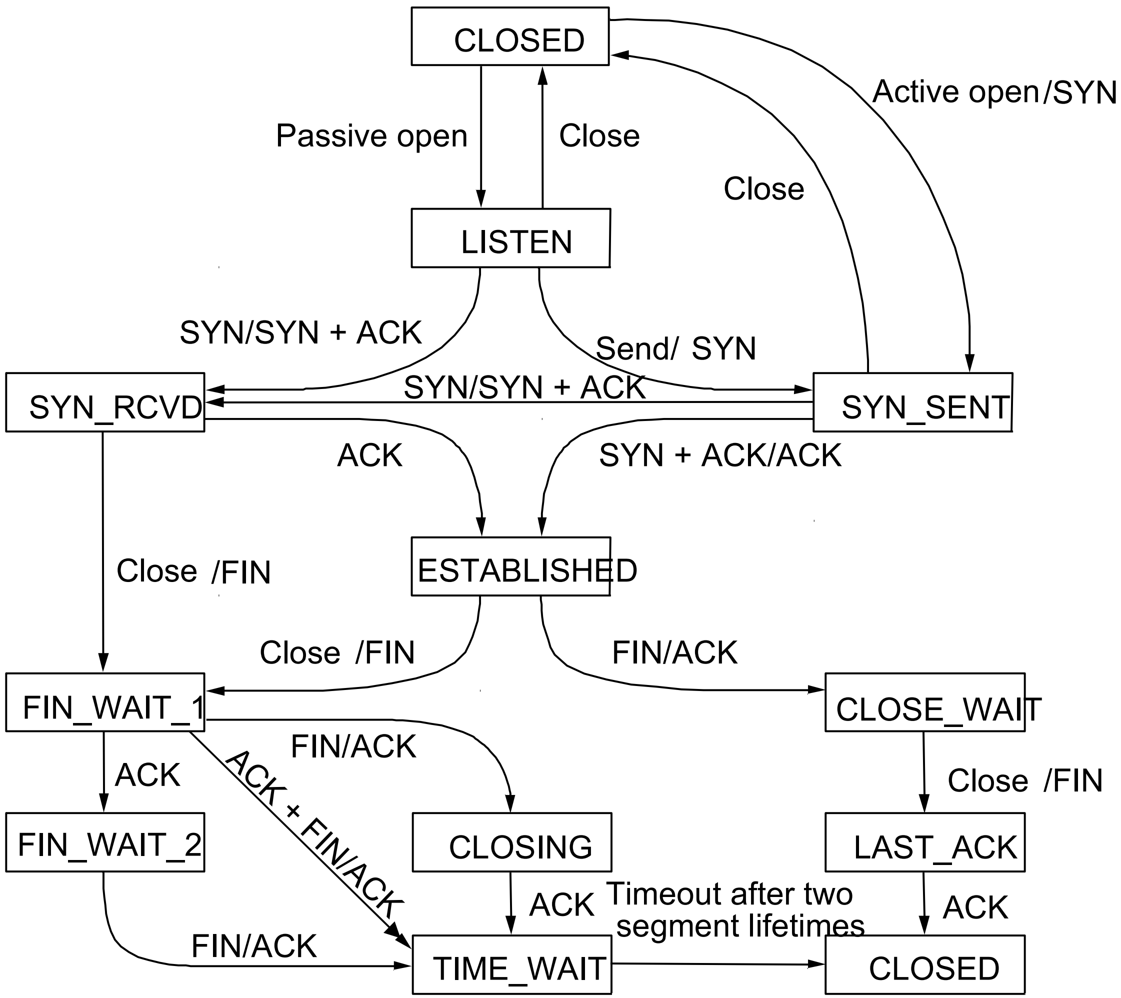
- Opens both directions for transfer



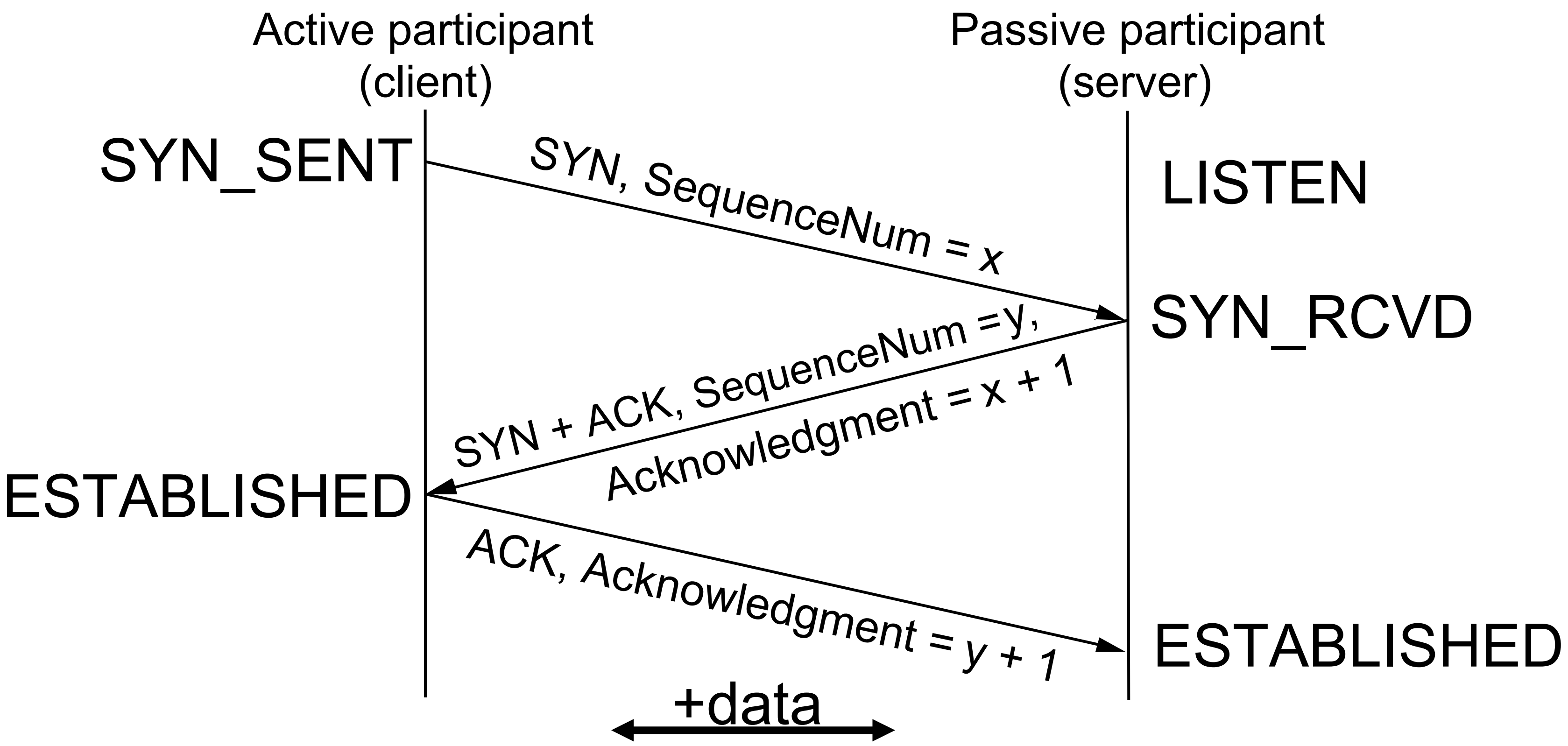
# Some Comments

- 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
- But with random ISN it actually “proves” that two hosts can communicate
  - Weak form of authentication

# TCP State Transitions



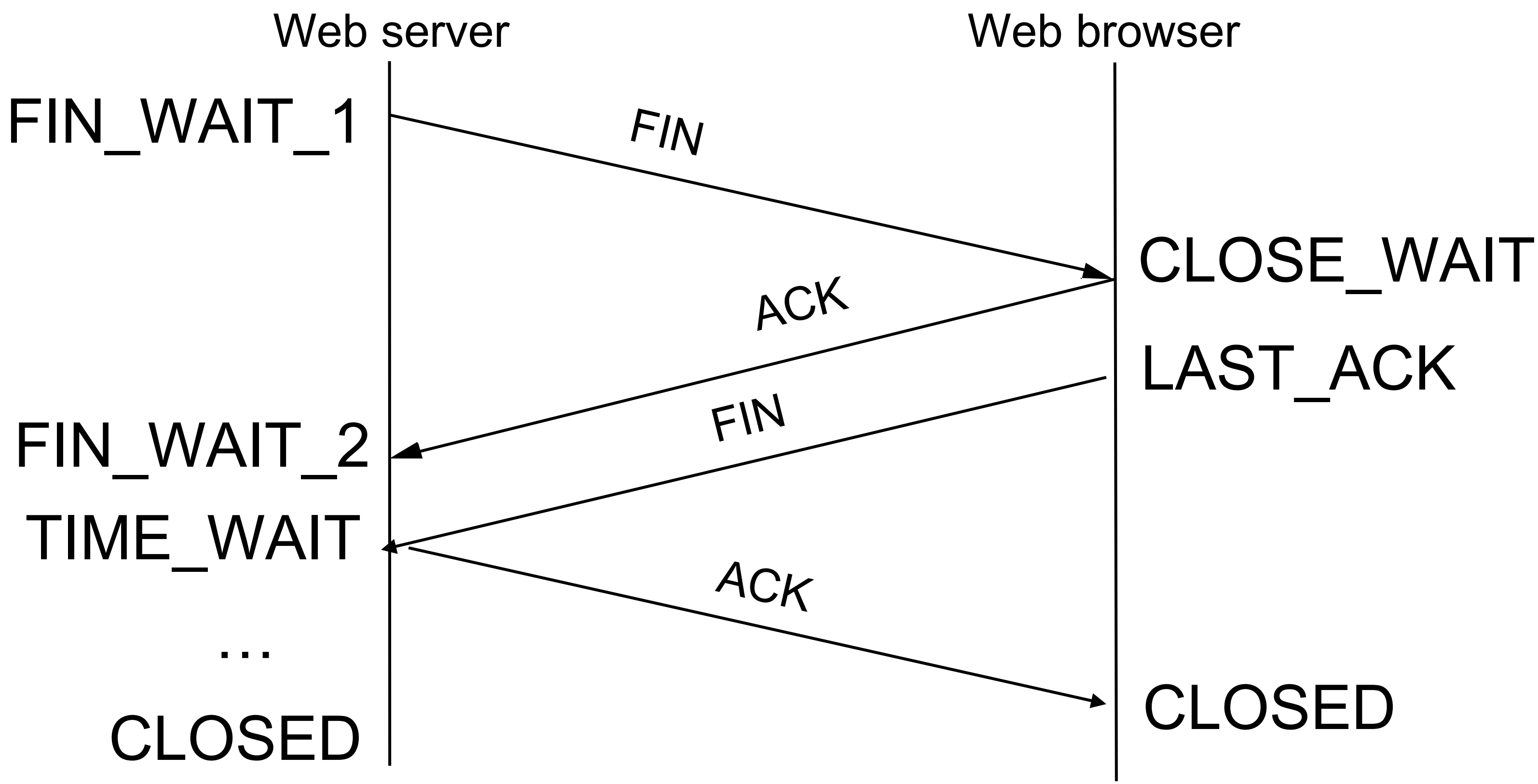
# Again, with States



# 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

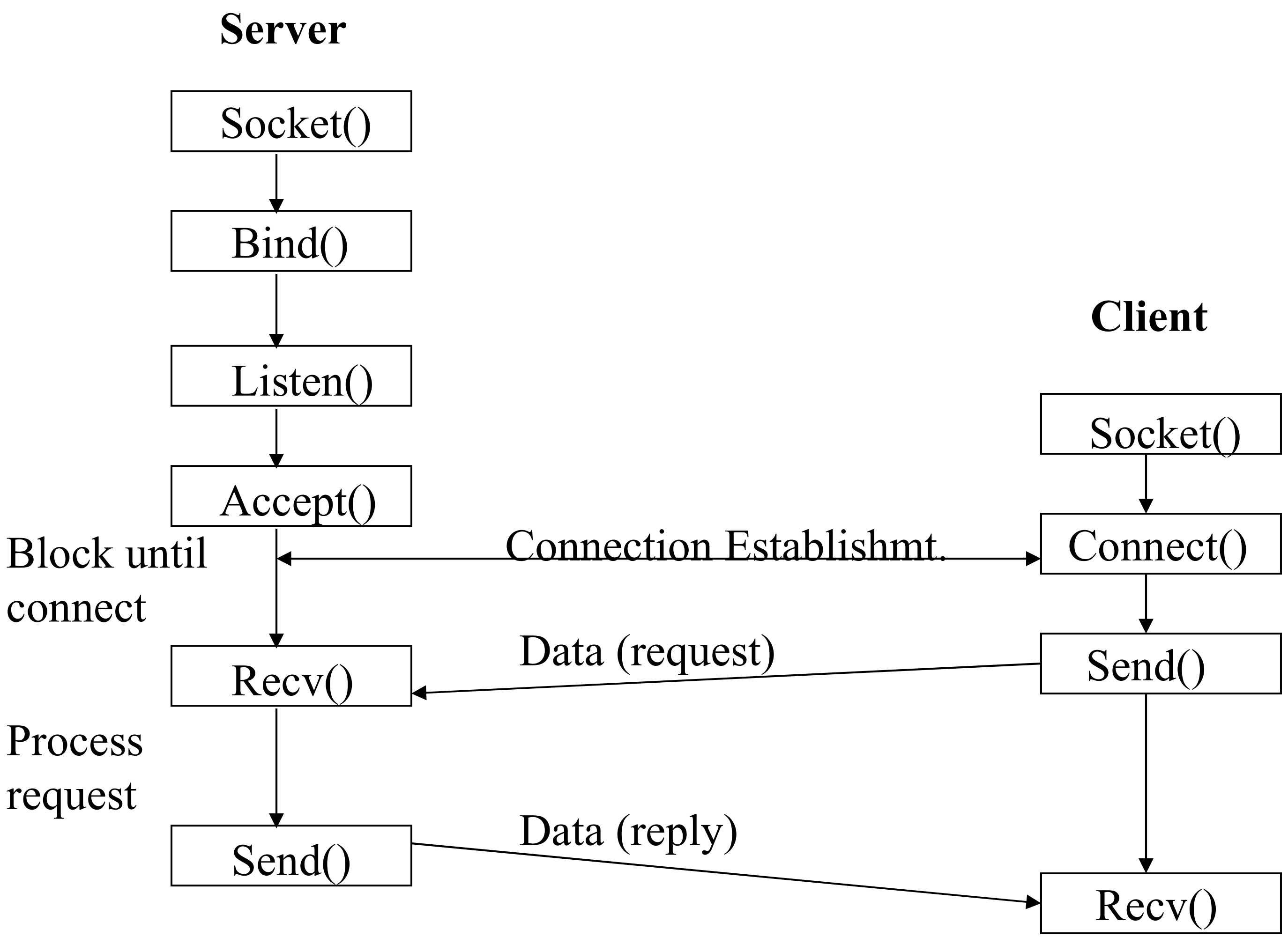
- We wait 2MSL (two times the maximum segment lifetime of 60 seconds) before completing the close
- Why?
- ACK might have been lost and so FIN will be resent
- Could interfere with a subsequent connection

# Berkeley Sockets interface

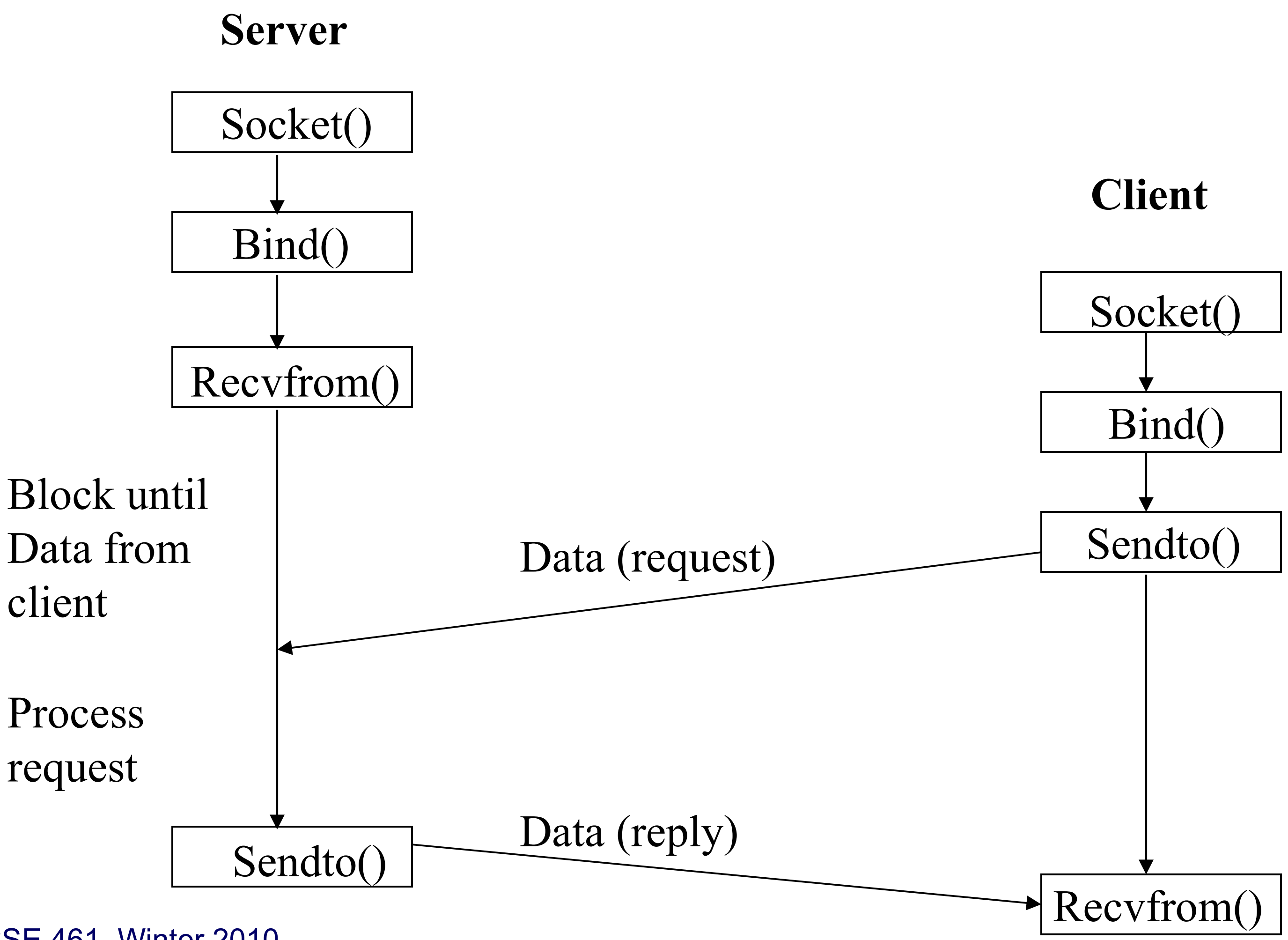
- Networking protocols implemented in OS
  - OS must expose a programming API to applications
  - most OSs use the "socket" interface
  - originally provided by BSD 4.1c in ~1982.

- 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

# TCP (connection-oriented)



# UDP (connectionless)



# Using Sockets: UDP

- import java.net.*;
- UDP sockets:
  - new DatagramSocket(); // binds to ephemeral port number
  - new DatagramSocket(port); // tries to bind to 'port'
- DatagramPacket
  - Unit of transfer between application and networking software
  - new DatagramPacket( byte[] buf, int len);
  - new DatagramPacket( byte[] buf, int len, InetAddress addr, int port);

- Sending data:
  - Construct a DatagramPacket
  - Set its data field, and its address components
  - myDatagramSocket.*send*( myDatagramPacket)

# Java / UDP

- Java also has an interface supporting `connect(SocketAddr addr)`, but it's a layer above UDP
  - Filters incoming packets not from *addr*
  - Filters outgoing packets not to *addr*

- Performance / correctness issue:
  - Is a copy of the data portion of a `DatagramPacket` made when `send()` is invoked, or is a reference to the `byte[] buf` kept?

- Blocking vs. non-blocking IO
  - Non-blocking options
    1. import java.net.*;
       - DatagramSocket.setSOTimeout(int timeout);
    2. import java.nio.*;
       - More general (complicated) support

# Using Sockets: TCP

- The TCP distinction between passive and active open is embedded in the (typical) socket interfaces
  - There are two kinds of sockets:
    - Socket
    - ServerSocket
- Server starts, creates a ServerSocket, binds it to a local port, and listens for a client to connect
- Client starts, creates a Socket on an ephemeral port, and connects to the server socket
- As a result of the connection, the server socket creates a *new* Socket to return to the application
  - Provides a handy way to identify/name a single flow in the application code

# TCP Server-side: Java

- Create:
  - `ServerSocket ss = new ServerSocket();`
  - `ServerSocket ss = new ServerSocket(port);`
- Listen:
  - `Socket s = ss.accept();`

# TCP Client side: Java

- Create:
  - `Socket s = new Socket();`
- Connect:
  - `s.connect(serverAddress);`
  - `S.connect(serverAddress, timeout);`
- Use:
  - It's Java, the sockets support streams, the mind boggles
  - `BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));`
    - `in.readLine();`
  - `PrintWriter out = new PrintWriter(s.getOutputStream(), true);`
    - `Out.print(data);`
  - `OutStream outStream = s.getOutputStream();`
    - `outStream.write( buf, 0, n);  // byte[] buf for n bytes starting at offset 0`

# Blocking Operations

- read() is a *blocking* operation
- What if other side crashes?
  - No data sent
  - No FIN sent
- Solutions on these slides are for Java, but general ideas apply universally

# Method 1: Timeouts

- Most language/socket interfaces will provide a way to way to say:

```
read N bytes, but wait no longer than T milliseconds
```

- On return, you either have up to N bytes or some indication that you timed out
  - Note: read(N) can often mean “read up to N” not “wait for N”
  - Note: readline(), if it's available, means “wait until you can read a \n”

- In Java, this is done by setting a *socket option*
  - `socket.setSOTimeout( 5 );    // set 5 msec. timeout`
  - `reader.read(buf, off, len); // wait up to 5 msec.`

# Method 2: Non-blocking IO

- Every language / OS will provide some way to do non-blocking IO
  - `read()` can be made to always return immediately, sometimes with an indication that it read nothing
  - A `willBlock()` method is probably available
  - A `waitFor( datasource[] )` method will be available
    - Means "block until at least one of the data sources has data available"
- In Java, these are provided by `java.nio` and related packages

# Not A Method: Multi-threading

- Multi-threading isn't really a solution when you need non-blocking semantics
- It is a fine solution when you're willing to block, but your goal is:
  - To overlap some processing with blocking/waiting
  - To read from more than one source
- Basic problem:
  - The application (probably) can't terminate cleanly until all threads have terminated
  - The only thread that can terminate a thread is itself
  - There's no general way to wake up a thread blocked on IO
  - 

# Key Concepts

- We use ports to name processes in TCP/UDP
  - "Well-known" ports are used for popular services
- Connection setup and teardown complicated by the effects of the network on messages
  - TCP uses a three-way handshake to set up a connection
  - TCP uses a symmetric disconnect