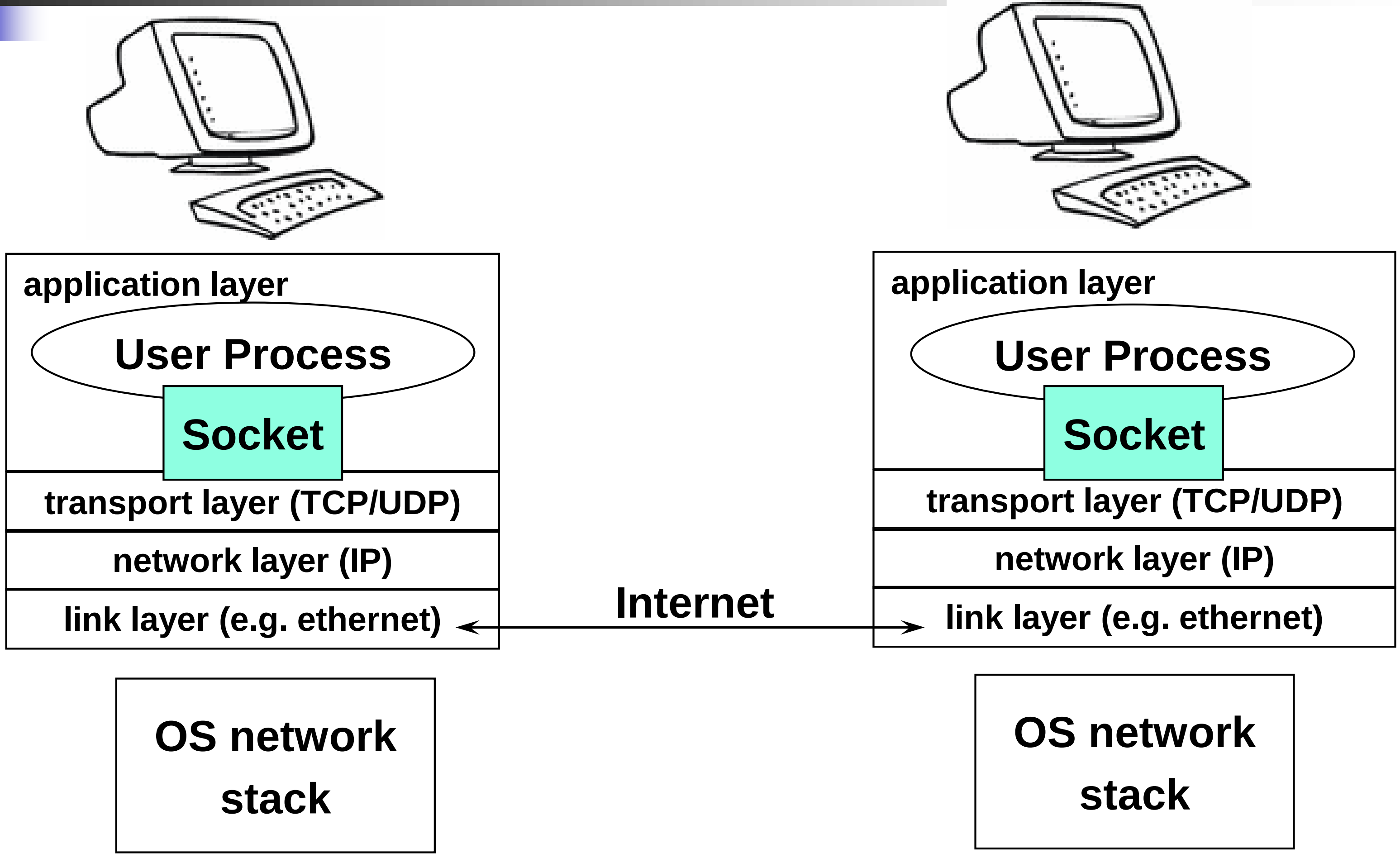
UNIX Sockets

Developed for the Azera Group

By: Joseph D. Fournier B.Sc.E.E., M.Sc.E.E.

Socket and Process Communication



The interface that the OS provides to its networking subsystem

WinSock

- Derived from Berkeley Sockets (Unix)
  - includes many enhancements for programming in the windows environment
- Open interface for network programming under Microsoft Windows
  - API freely available
  - Multiple vendors supply winsock
  - Source and binary compatibility
- Collection of function calls that provide network services

Data Delivery Architecture

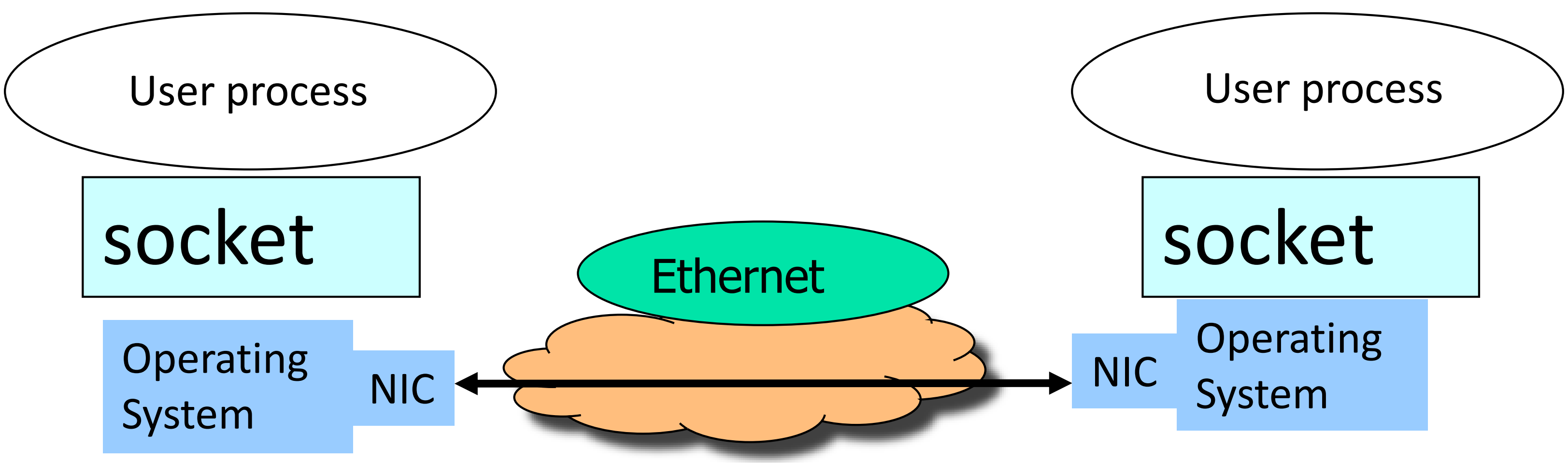
- Network
  - Deliver data packet to the destination host
  - Based on the destination IP address
- Operating system
  - Deliver data to the destination socket
  - Based on the destination port number (e.g., 80)
- Application
  - Read data from and write data to the socket
  - Interpret the data (e.g., render a Web page)

Socket: End Point of Communication

- Sending message from one process to another
  - Message must traverse the underlying network
- Process sends and receives through a “socket”
  - In essence, the doorway leading in/out of the house
- The “socket” is a small piece of software that lives between the application layer and the transport layer

Socket: End Point of Communication

- Socket as an Application Programming Interface
  - Supports the creation of network applications



Application Processes Protocol

- Datagram Socket (UDP)
  - Collection of messages
  - Best effort
  - Connectionless
- Stream Socket (TCP)
  - Stream of bytes
  - Reliable
  - Connection-oriented

User Datagram Protocol (UDP):

UDP

- Single socket to receive messages
- No guarantee of delivery
- Not necessarily in-order delivery
- Datagram – independent packets
- Must address each packet

Example UDP applications
Multimedia, voice over IP (Skype)

(TCP): Stream Socket

TCP

- Reliable – guarantee delivery
- Byte stream – in-order delivery
- Connection-oriented – single socket per connection
- Setup connection followed by data transfer

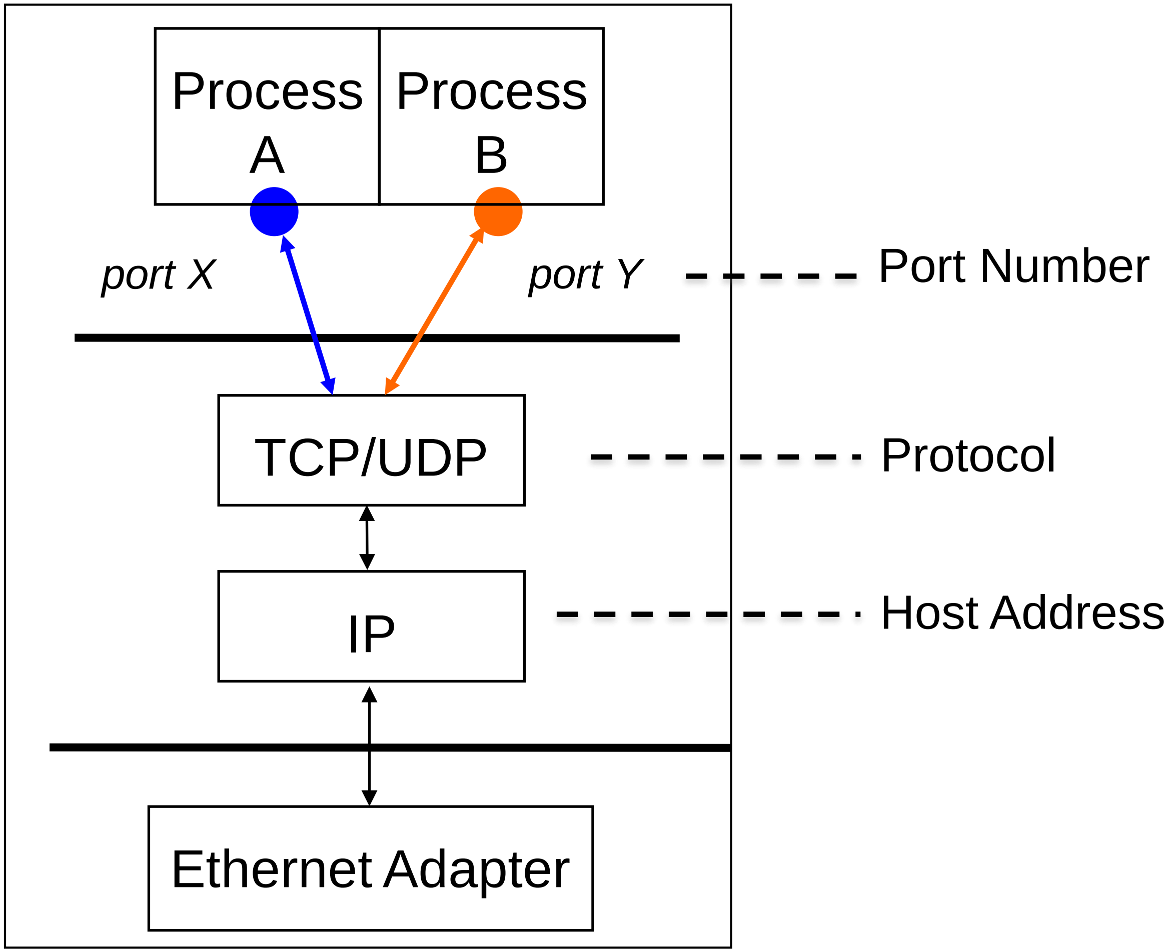
Example TCP applications

Web, Email, Telnet

Socket Identification: Part One

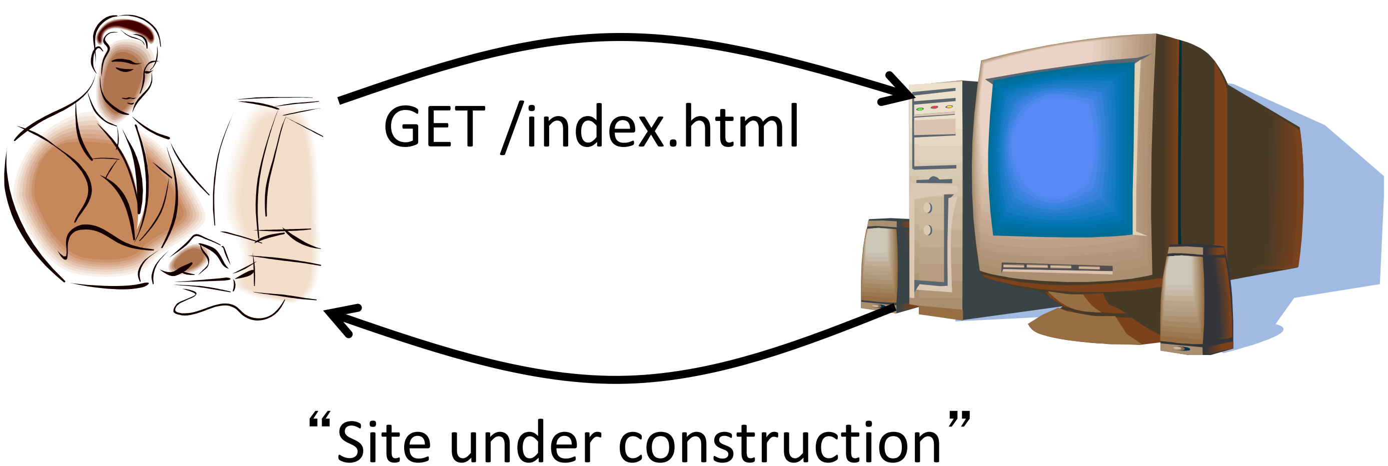
- Communication Protocol
  - TCP (Stream Socket): streaming, reliable
  - UDP (Datagram Socket): packets, best effort
- Receiving host
  - Destination **address** that uniquely identifies the host
  - An **IP address** is a 32-bit quantity
- Receiving socket
  - Host may be running many different processes
  - Destination **port** that uniquely identifies the socket
  - A **port number** is a 16-bit quantity

Socket Identification (Cont.)



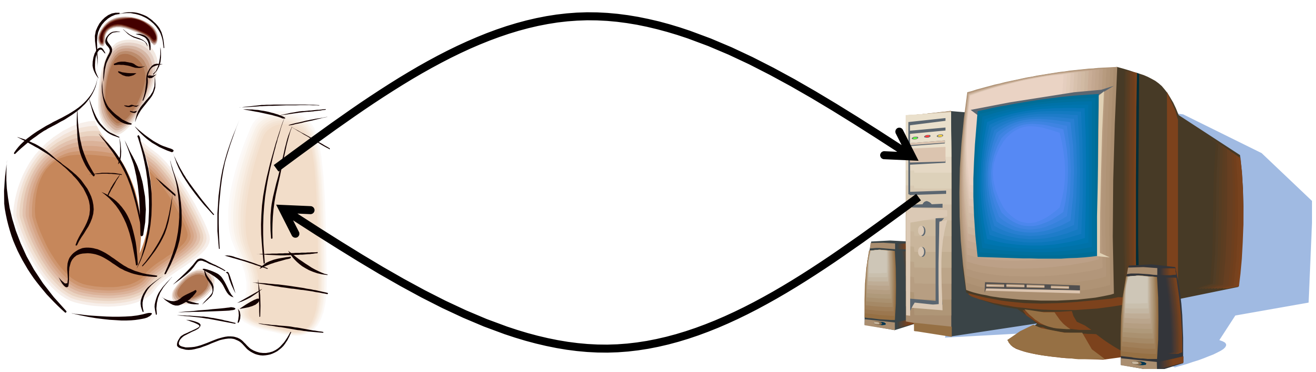
Clients and Servers

- Client program
  - Running on end host
  - Requests service
  - E.g., Web browser
- Server program
  - Running on end host
  - Provides service
  - E.g., Web server



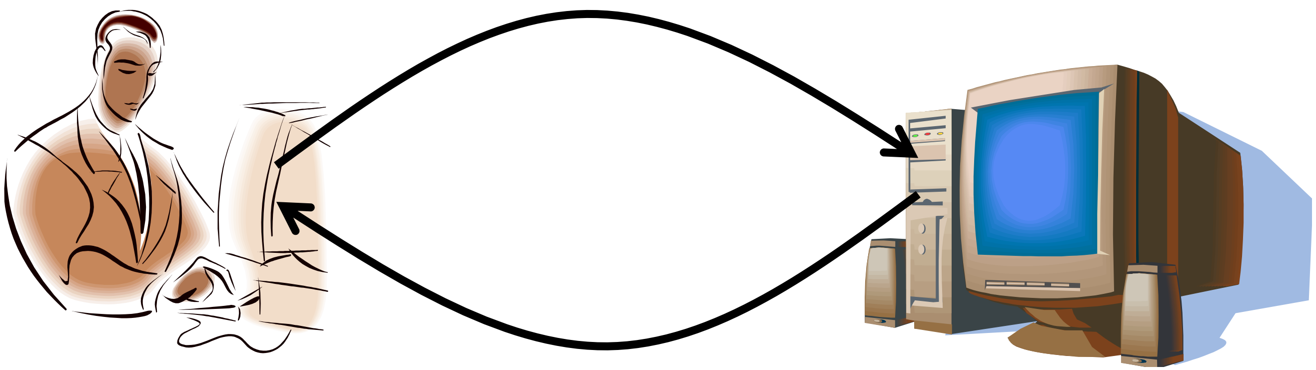
Client-Server Communication

- Client "sometimes on"
  - Initiates a request to the server
  - E.g., Web browser on your laptop
  - Doesn't communicate directly with other clients



Client-Server Communication

- Server is "always on"
  - Handles services requests from many client hosts
  - E.g., Web server for the www.cnn.com Web site
  - Doesn't initiate contact with the clients
  - Needs fixed, known address



Client and Server Processes

- Client process
  - process that initiates communication
- Server Process
  - process that waits to be contacted

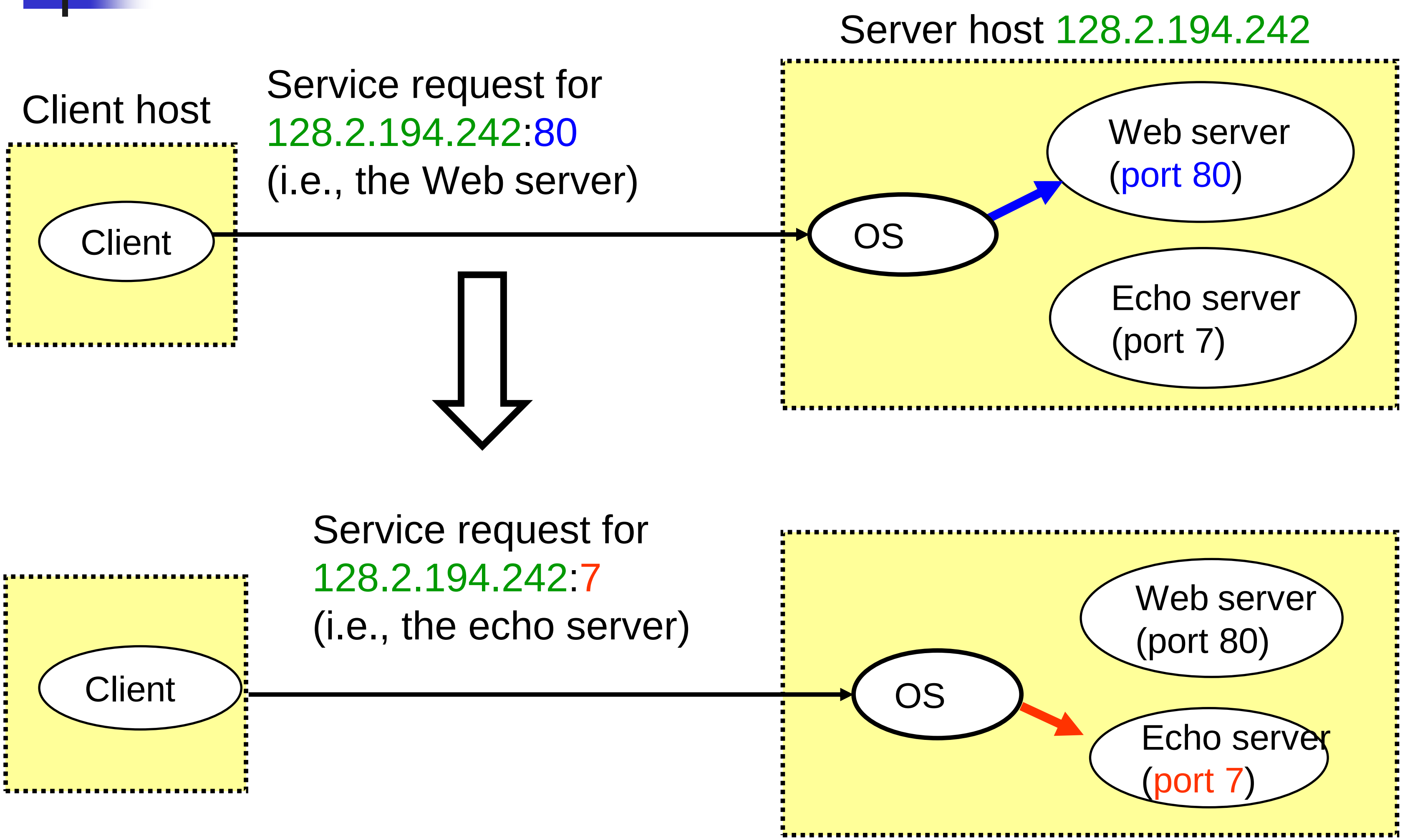
Knowing What Port Number To Use

- Popular applications have well-known ports
  - E.g., port 80 for Web and port 25 for e-mail
  - nsiiops 261/tcp IIOP Name Service over TLS/SSL
  - https 443/tcp http protocol over TLS/SS
  - ftps-data 989/tcp ftp protocol, data, over TLS/SSL
  - ftps 990/tcp ftp, control, over TLS/SSL
  - telnets 992/tcp telnet protocol over TLS/SSL
  - imaps 993/tcp imap4 protocol over TLS/SSL
  - ircs 994/tcp irc protocol over TLS/SSL
  - pop3s 995/tcp pop3 protocol over TLS/SSL

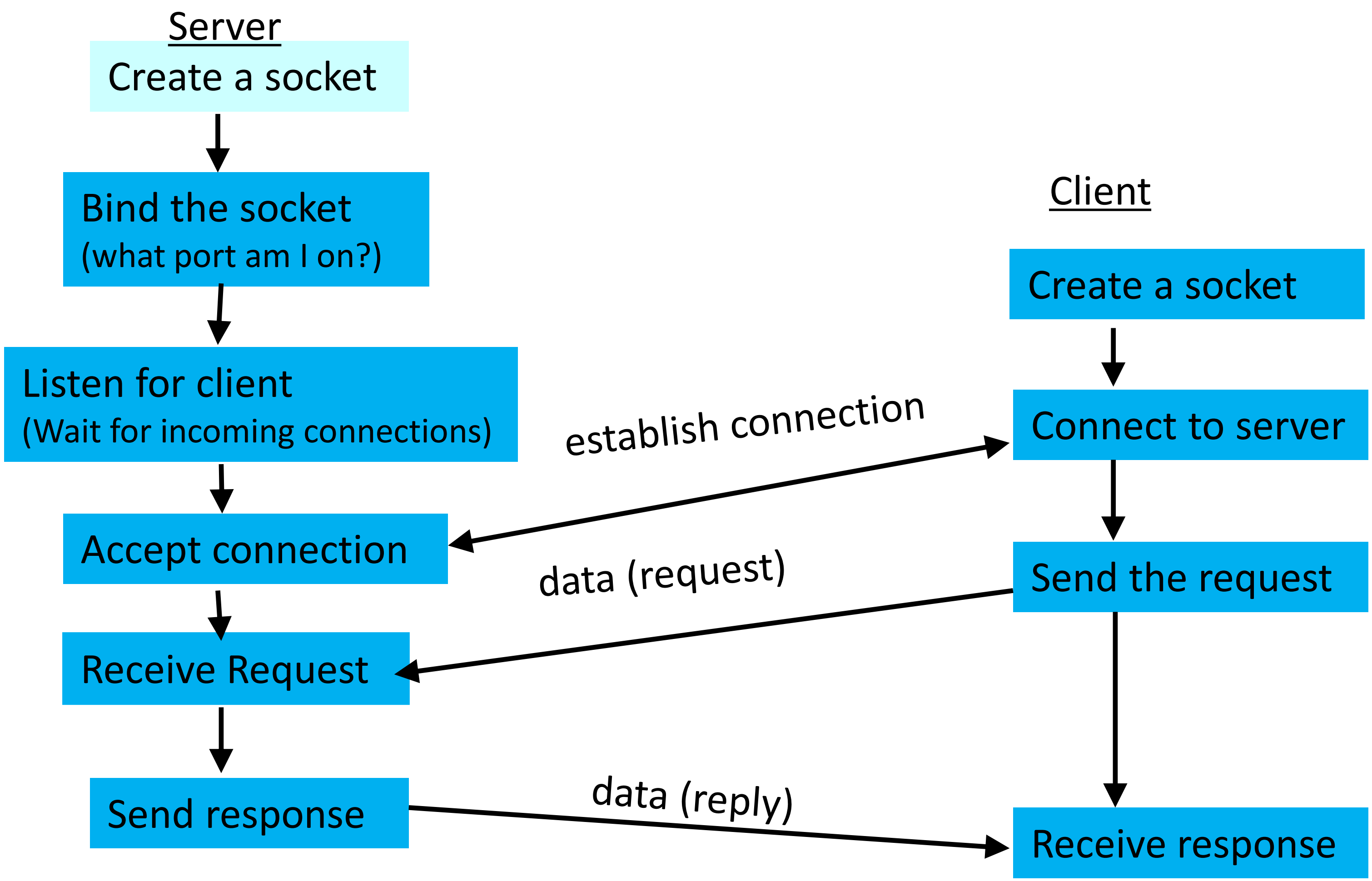
Knowing What Port Number To Use

- Well-known vs. ephemeral ports
  - Server has a well-known port (e.g., port 80)
    - Between 0 and 1023 (requires root to use)
  - Client picks an unused ephemeral (i.e., temporary) port
    - Between 1024 and 65535
- Uniquely identifying traffic between the hosts
  - Two IP addresses and two port numbers
  - Underlying transport protocol (e.g., TCP or UDP)

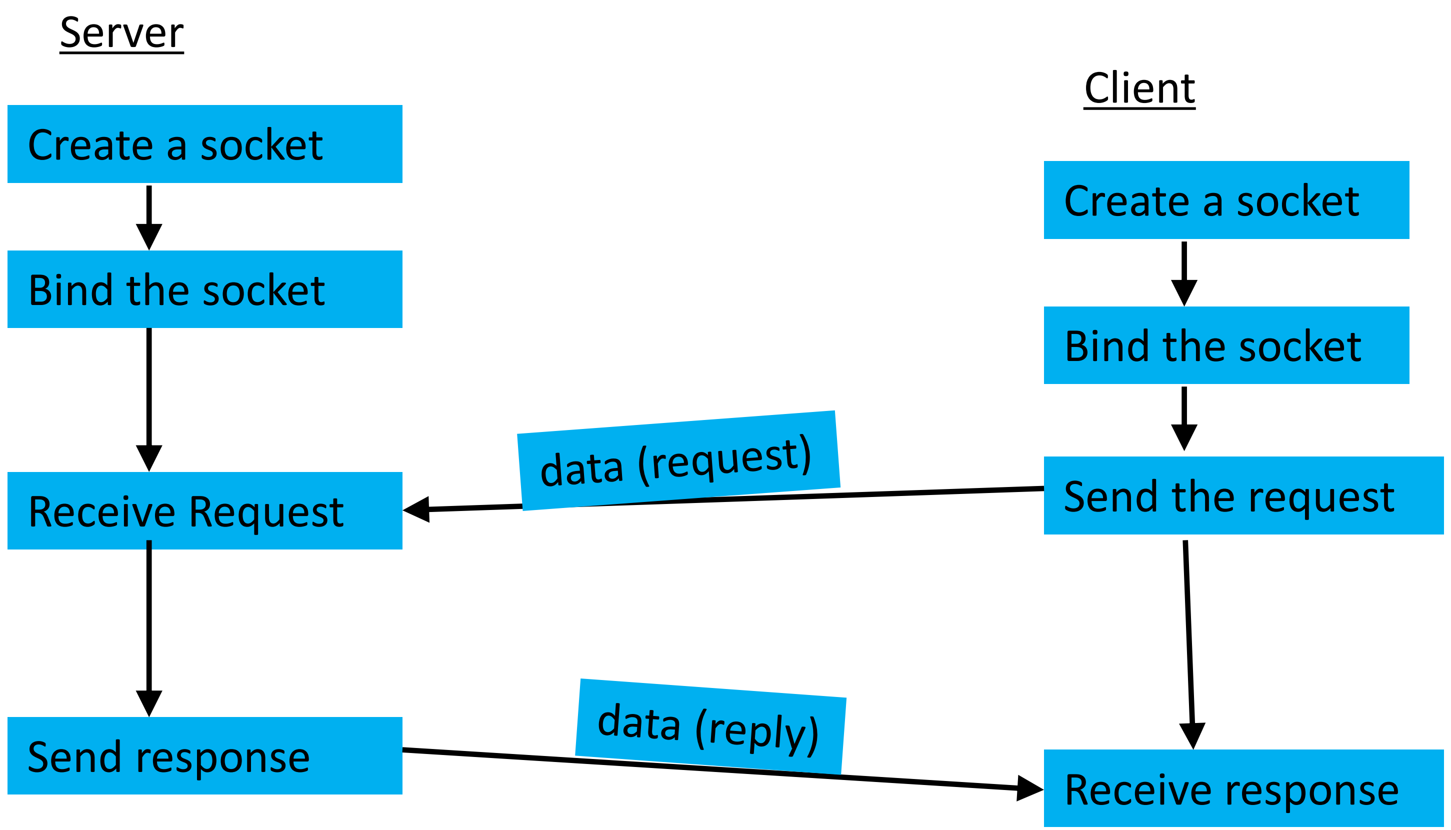
Using Ports to Identify Services



Client-Server Communication



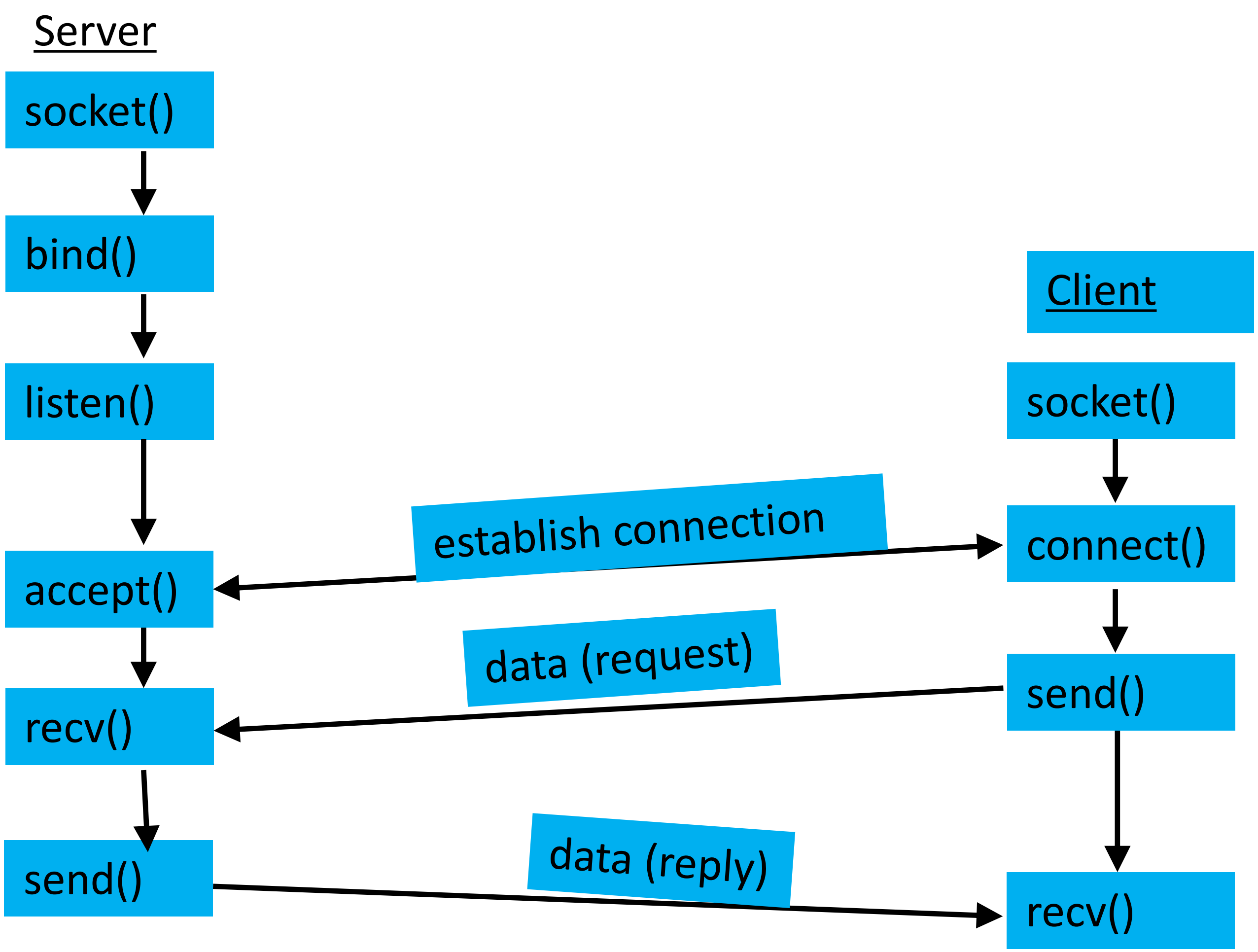
Datagram Sockets (UDP): Connectionless



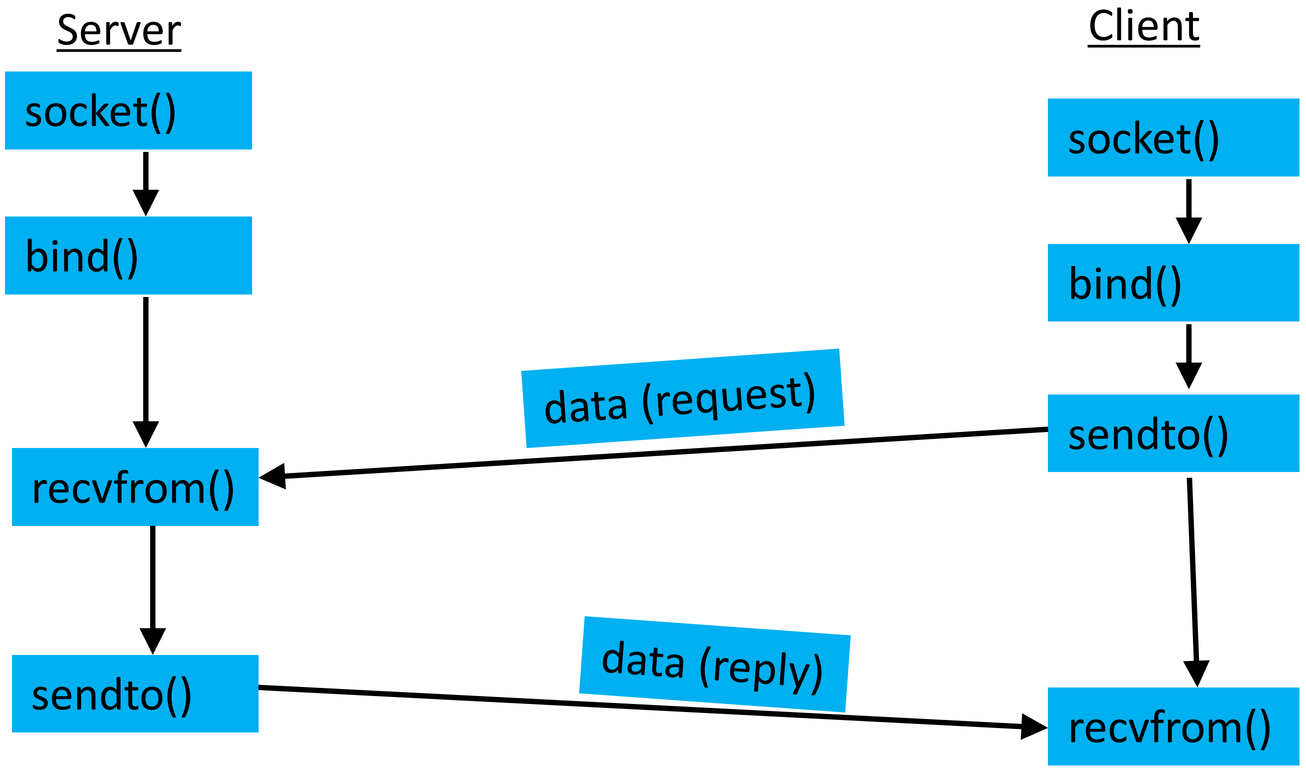
UNIX Socket API

- Socket interface
  - Originally provided in Berkeley UNIX
  - Later adopted by all popular operating systems
  - Simplifies porting applications to different OSes
- In UNIX, everything is like a file
  - All input is like reading a file
  - All output is like writing a file
  - File is represented by an integer file descriptor
- API implemented as system calls
  - E.g., connect, send, recv, close, …

Connection-oriented Example-TCP



Connectionless Example- UDP



Server Address/Port

- Server typically known by name and service
- Need to translate into IP address and port #
  - E.g., “64.236.16.20” and “80”
- Get address info with given host name and service
  - ```
    int getaddrinfo( char *node,
                     char *service
                     struct addrinfo *hints,
                     struct addrinfo **result)
    ```
  - *node: host name (e.g., “www.cnn.com”) or IP address
  - *service: port number or service listed in */etc/services* (e.g. ftp)
  - hints: points to a *struct addrinfo* with known information
    ```

Server Address/Port (cont.)

- Data structure to host address information

```
struct addrinfo {
        int                         ai_flags;

        int                         ai_family;
   //e.g. AF_INET for IPv4
        int                         ai_socketype;
   //e.g. SOCK_STREAM for TCP
        int                         ai_protocol;
   //e.g. IPPROTO_TCP         size_t ai_addrlen;
        char                        *ai_canonname;
        struct sockaddr      *ai_addr;
   // point to sockaddr struct
        struct addrinfo      *ai_next;
}
```

Client: Creating a Socket

- Creating a socket
  - `int socket(int domain, int type, int protocol)`
  - Returns a file descriptor (or handle) for the socket
- Domain: protocol family
  - PF_INET for IPv4
  - PF_INET6 for IPv6
- Type: semantics of the communication
  - SOCK_STREAM: reliable byte stream (TCP)
  - SOCK_DGRAM: message-oriented service (UDP)
- Protocol: specific protocol
  - UNSPEC: unspecified
  - (PF_INET and SOCK_STREAM already implies TCP)

Client: Connecting Socket to Server

- Client contacts the server to establish connection
  - Associate the socket with the server address/port
  - Acquire a local port number (assigned by the OS)
  - Request connection to server, who hopefully accepts
  - connect is **blocking**
- Establishing the connection
  - ```
    int connect(int sockfd,
     struct sockaddr *server_address,
              socketlen_t addrlen )
    ```
  - Args: socket descriptor, server address, and address size
  - Returns 0 on success, and -1 if an error occurs
    ```

Client: Sending Data

- Sending data
  - ```
    int send(int sockfd, void *msg,
                      size_t len, int flags)
    ```
  - Arguments: socket descriptor, pointer to buffer of data to send, and length of the buffer
  - Returns the number of bytes written, and -1 on error
  - send is **blocking**: return only after data is sent
  - Write short messages into a buffer and send once
    ```

Client: Receiving Data

- Receiving data
  - ```
    int recv(int sockfd, void *buf,
                 size_t len, int flags)
    ```
  - Arguments: socket descriptor, pointer to buffer to place the data, size of the buffer
  - Returns the number of characters read (where 0 implies "end of file"), and -1 on error
  - Why do you need len? What happens if buf's size < len?
  - recv is **blocking**: return only after data is received
    ```

Server: Server Preparing its Socket

- Server creates a socket and binds address/port
  - Server creates a socket, just like the client does
  - Server associates the socket with the port number

- Create a socket
  - ```
    int socket(int domain,
               int type, int protocol   )
    ```
- Bind socket to the local address and port number
  - ```
    int bind(int sockfd,
     struct sockaddr *my_addr,
                      socklen_t addrlen   )
    ```

Server: Allowing Clients to Wait

- Many client requests may arrive
  - Server cannot handle them all at the same time
  - Server could reject the requests, or let them wait
- Define how many connections can be pending
  - `int listen(int sockfd, int backlog)`
  - Arguments: socket descriptor and acceptable backlog
  - Returns a 0 on success, and -1 on error
  - Listen is **non-blocking**: returns immediately
- What if too many clients arrive?
  - Some requests don't get through
  - The Internet makes no promises…
  - And the client can always try again

Server: Accepting Client Connection

- Now all the server can do is wait…
  - Waits for connection request to arrive
  - **Blocking** until the request arrives
  - And then accepting the new request
- Accept a new connection from a client
  - ```
    int accept(int sockfd,
               struct sockaddr *addr,
                socketlen_t *addrlen)
    ```
  - Arguments: sockfd, structure that will provide client address and port, and length of the structure
  - Returns descriptor of socket for this new connection
    ```

Client and Server: Cleaning House

- Once the connection is open
  - Both sides and read and write
  - Two unidirectional streams of data
  - In practice, client writes first, and server reads
  - … then server writes, and client reads, and so on
- Closing down the connection
  - Either side can close the connection
  - … using the `int close(int sockfd)`
- What about the data still "in flight"
  - Data in flight still reaches the other end

Server: One Request at a Time?

- Serializing requests is inefficient
  - Server can process just one request at a time
  - All other clients must wait until previous one is done
  - What makes this inefficient?
- May need to time share the server machine
  - Alternate between servicing different requests
    - Do a little work on one request, then switch when you are waiting for some other resource (e.g., reading file from disk)
    - “Nonblocking I/O”
  - Or, use a different process/thread for each request
    - Allow OS to share the CPU(s) across processes
  - Or, some hybrid of these two approaches

Handle Multiple Clients using fork()

- Steps to handle multiple clients
  - Go to a loop and accept connections using accept()
  - After a connection is established, call fork() to create a new child process to handle it
  - Go back to listen for another socket in the parent process
  - close() when you are done.
- Want to know more?
  - Checkout out *Beej's guide to network programming*

Example Clients and Servers

- Apache Web server
  - Open source server first released in 1995
  - Name derives from “a patchy server” ;-)
- Mozilla Web browser
- Sendmail
- BIND Domain Name System
  - Client resolver and DNS server