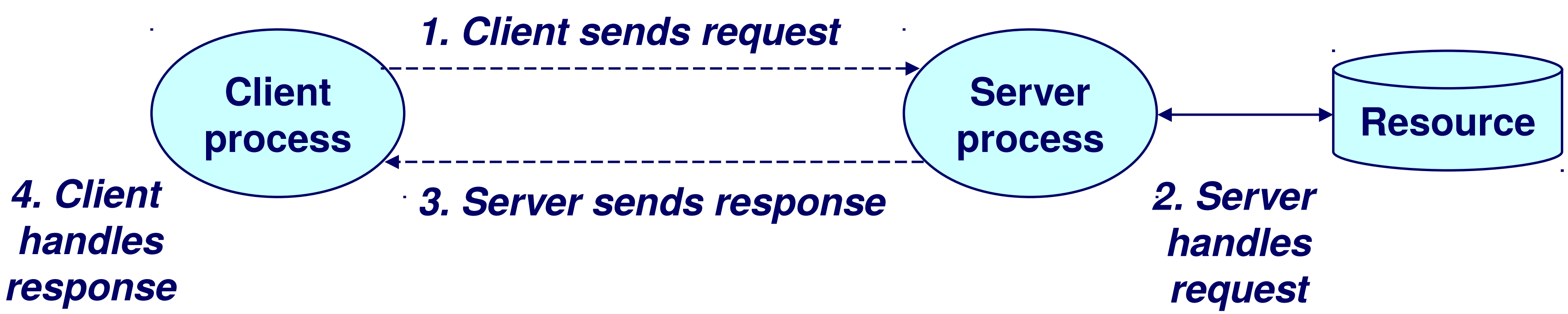
# Socket programming

# A Client-Server Exchange

- A *server* process and one or more *client* processes
- Server manages some *resource*.
- Server provides *service* by manipulating resource for clients.




***Note: clients and servers are processes running on hosts (can be the same or different hosts).***

# Basic networking theory

- The TCP/IP communication stack has four layers
  - Application layer (DHCP, SSH, LDAP, FTP, HTTP)
  - Transport layer (TCP, UDP)
  - Internet layer (IPv4, IPv6, ICMP)
  - Link layer (MAC, ask EE people)

# Link layer connections: LAN

- IEEE 802.3, the ethernet, is the most popular LAN setup
- Each machine on a LAN has a network interface (e.g. ethernet card) connected to a common broadcast medium
- Each interface card is associated with a Medium Access Control (MAC) address

# MAC addresses

- 48 bit unique identifier assigned to a network interface controller
- Human readable format: six octets represented in hexadecimal number pairs
  - 01-23-45-67-89-AB
  - 32:12:34:6F:45:A3
- Of six octets
  - First three are organization identifiers
  - Last three are device identifiers assigned by manufacturer

# Ethernet frame format

- Data frames on the ethernet are packets of a specific size and format
  - Preamble
  - Start delimiter
  - MAC dest
  - MAC source
  - Payload
  - Gap

# Ethernet protocol

- The sender broadcasts
- Each host connected to the LAN checks destination address in dataframe with own MAC address
- If destination matches, receive the packet

- Is this a good protocol?

# Enter the (inter)net

- Multiple LANs want to communicate with each other
- Most popular protocol is the internet protocol (IP)
- Each device connected to the internet has an IP address (IPv4 or IPv6)
- IPv4 uses 32 bit addressing
- IPv6 uses 128 bit addressing

# IPv4 problems

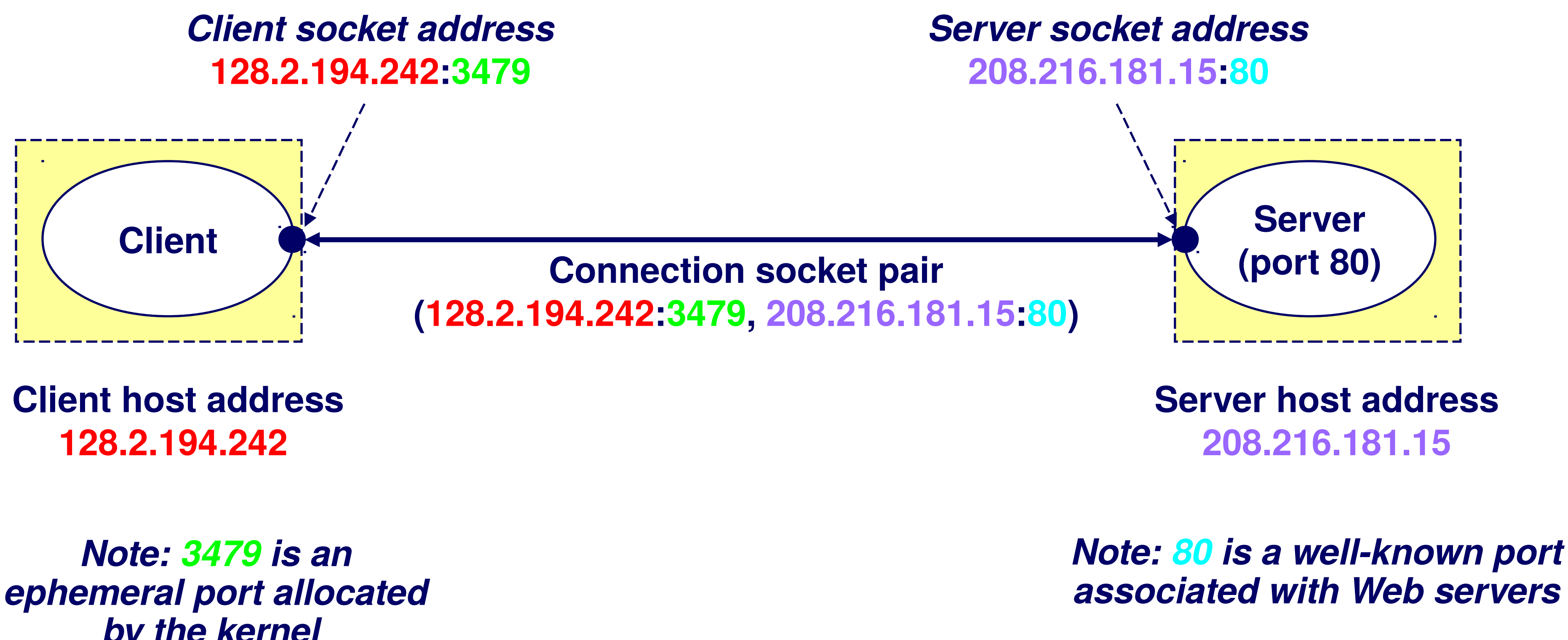
- Too few unique addresses available
- Forced lots of hacky solutions
- Example: NAT
  - Try to find your IP using ifconfig
  - Try to find your IP using a third party lookup service
  - Why is there a difference?
- Eventually will be replaced by IPv6
  - For the time being, have to be aware of hacks

# Internet Connections (TCP/IP)

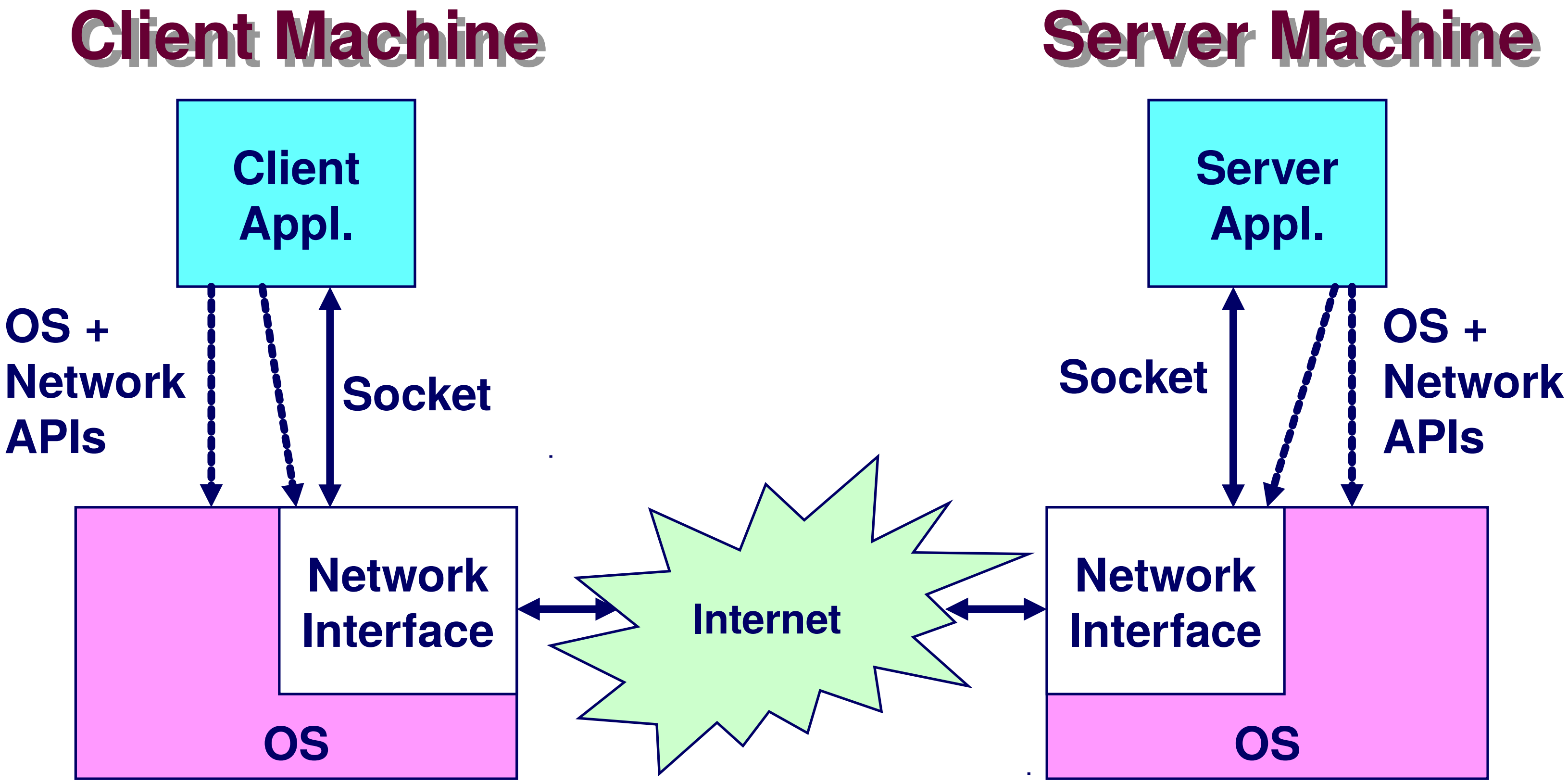
Two common paradigms for clients and servers communication

- Datagrams (UDP protocol SOCK_DGRAM)
- Connections (TCP protocol, SOCK_STREAM)

Connections are point-to-point, full-duplex (2-way communication), and reliable.



# Network Applications



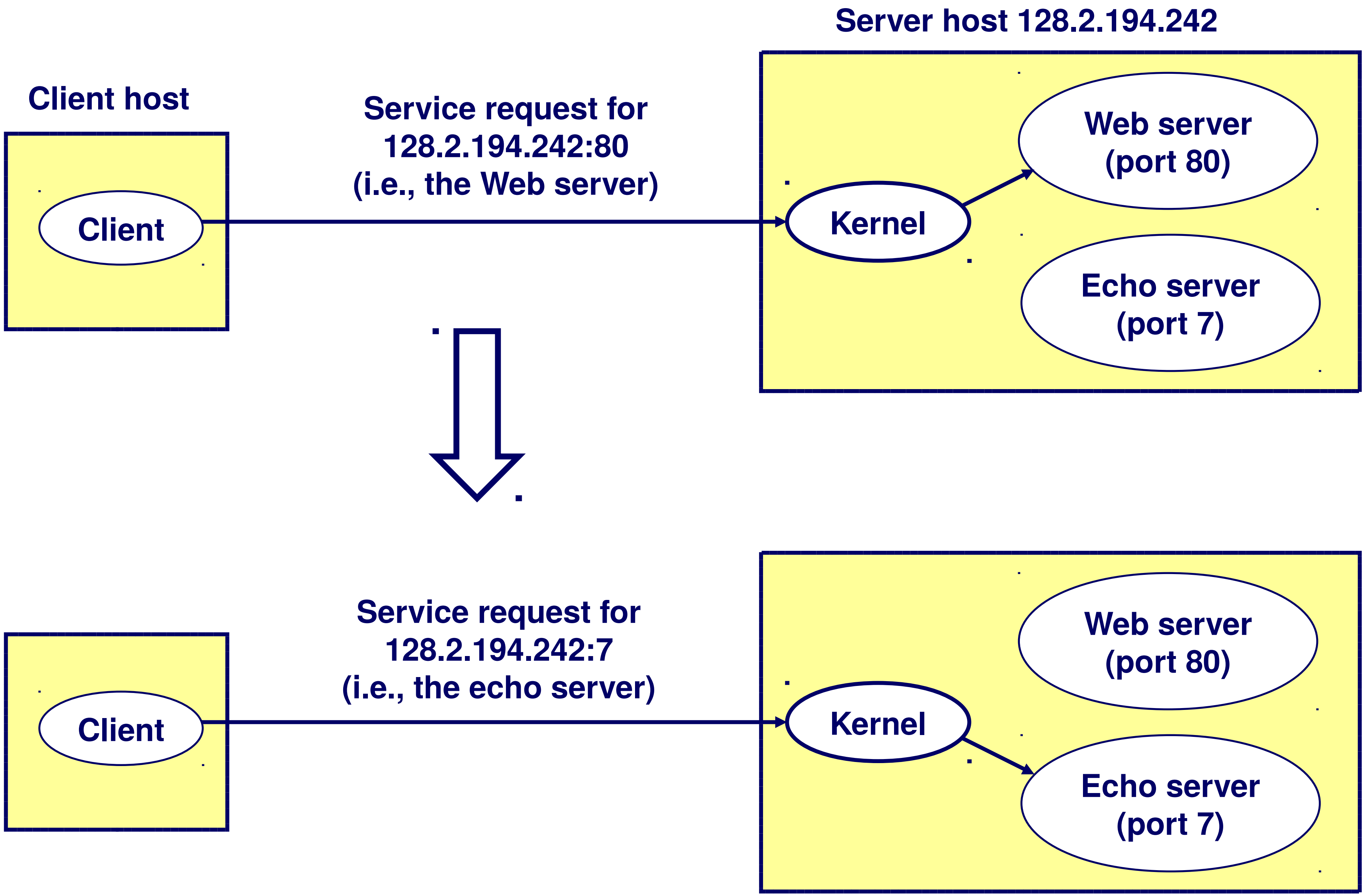

Access to Network via Program Interface

- Sockets make network I/O look like files
- Call system functions to control and communicate
- Network code handles issues of routing, segmentation.

# Clients

- Examples of client programs
  - Web browsers, `ftp`, `telnet`, `ssh`
- How does a client find the server?
  - The IP address in the server socket address identifies the host *(more precisely, an adaptor on the host)*
  - The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
  - Examples of well known ports
    - Port 7: Echo server
    - Port 23: Telnet server
    - Port 25: Mail server
    - Port 80: Web server

# Using Ports to Identify Services



# Servers

Servers are long-running processes (daemons).

- Created at boot-time (typically) by the init process (process 1)
- Run continuously until the machine is turned off.

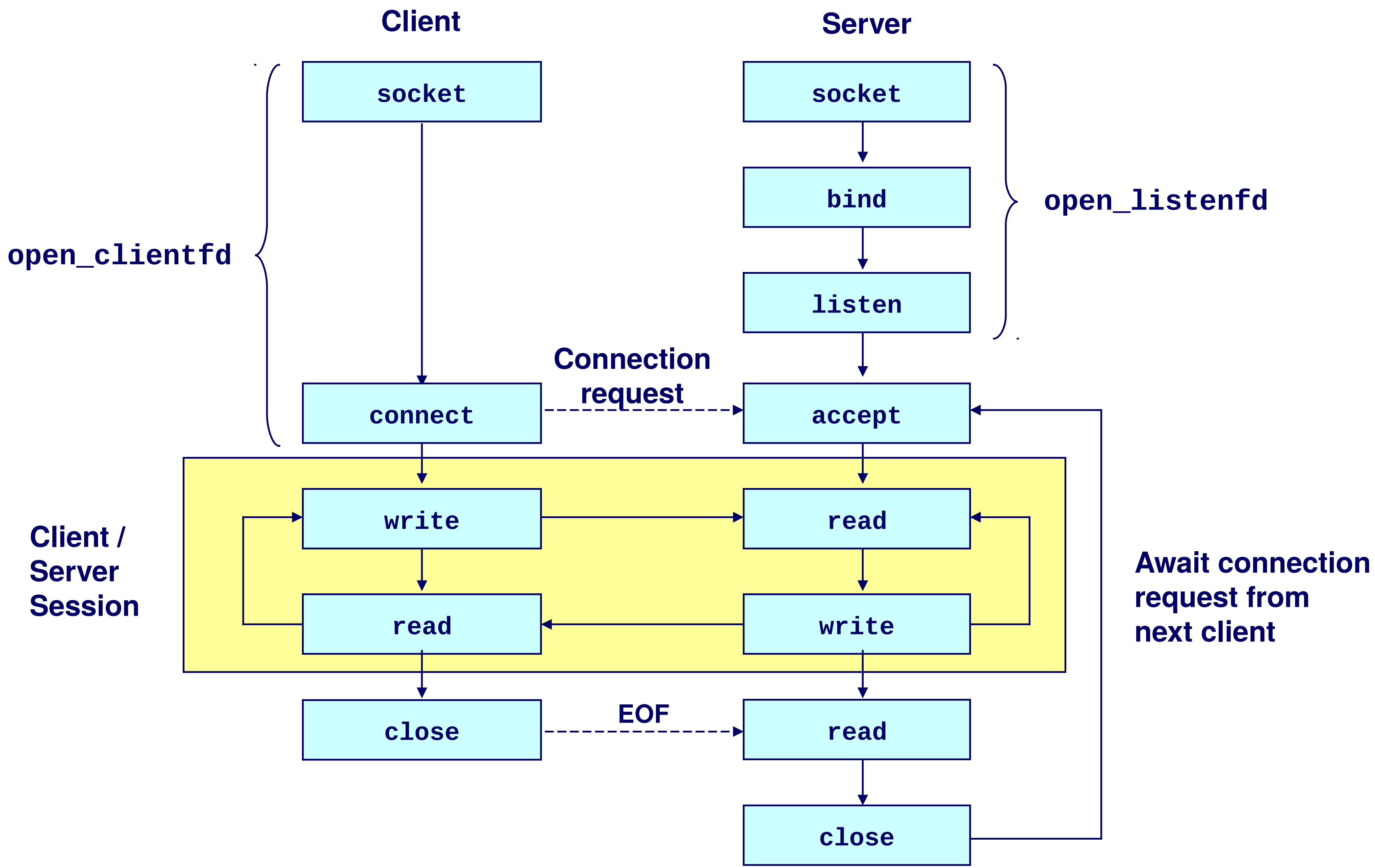
Each server waits for requests to arrive on a well-known port associated with

> **See `/etc/services` for a comprehensive list of the services available on a Linux machine.**

- Port 7: echo server
- Port 23: telnet server
- Port 25: mail server
- Port 80: HTTP server

A machine that runs a server process is also often referred to as a “server.”

# Overview of the Sockets Interface



# Sockets

What is a socket?

- To the kernel, a socket is an endpoint of communication.
- To an application, a socket is a file descriptor that lets the application read/write from/to the network.
  - Remember: All Unix I/O devices, including networks, are modeled as files.

Clients and servers communicate with each by reading from and writing to socket descriptors.

The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors.

# Socket Programming Cliches

Network Byte Ordering

- Network is big-endian, host may be big- or little-endian
- Functions work on 16-bit (short) and 32-bit (long) values
- htons() / htonl() : convert host byte order to network byte order
- ntohs() / ntohl(): convert network byte order to host byte order
- Use these to convert network addresses, ports, …

Structure Casts

- You will see a lot of ‘structure casts’

```
struct sockaddr_in serveraddr;
/* fill in serveraddr with an address */
…
/* Connect takes (struct sockaddr *) as its second argument */
connect(clientfd, (struct sockaddr *) &serveraddr,
     sizeof(serveraddr));
…
```

# Socket Programming Help

- man is your friend (aka RTFM)
  - man accept
  - man select
  - Etc.
- The manual page will tell you:
  - What #include<> directives you need at the top of your source code
  - The type of each argument
  - The possible return values
  - The possible errors (in errno)

# The Socket Interface


- The basic ideas:
  - a ***socket*** is like a file:
    - you can read/write to/from the network just like you would a file
  - For connection-oriented communication (e.g. TCP)
    - servers (passive open) do **listen** and **accept** operations
    - clients (active open) do **connect** operations
    - both sides can then do **read** and/or **write** (or **send** and **recv)**
    - then each side must **close**
    - There are more details, but those are the most important ideas
  - Connectionless (e.g. UDP): uses **sendto** and **recvfrom**

# Sockets And Socket Libraries

- **In Unix, socket procedures** (e.g. listen, connect, etc.) **are *system calls***
  - part of the operating system
  - implemented in the “top half” of the kernel
  - when you call the function, control moves to the operating system, and you are using “system” CPU time

# Socket Address Structures

## Generic socket address:

- For address arguments to `connect`, `bind`, and `accept`.

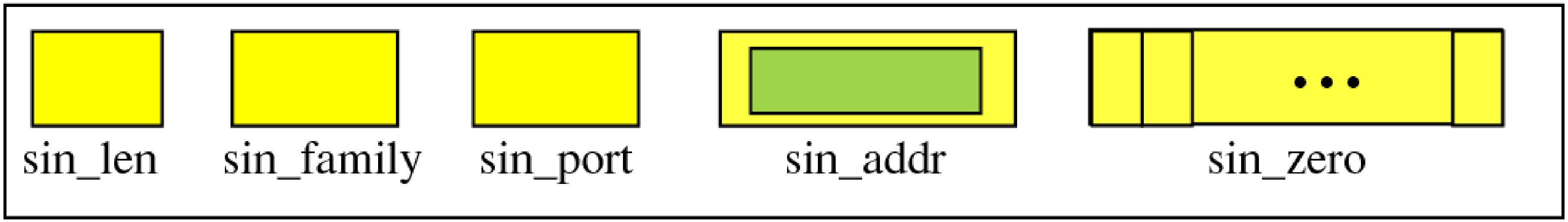
```
struct sockaddr {
  unsigned short  sa_family;    /* protocol family */
  char            sa_data[14];  /* address data.  */
};
```

## Internet-specific socket address:

- Must cast (`sockaddr_in *`) to (`sockaddr *`) for `connect`, `bind`, and `accept`.

```
struct sockaddr_in  {
  unsigned short  sin_family;  /* address family (always AF_INET) */
  unsigned short  sin_port;    /* port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```

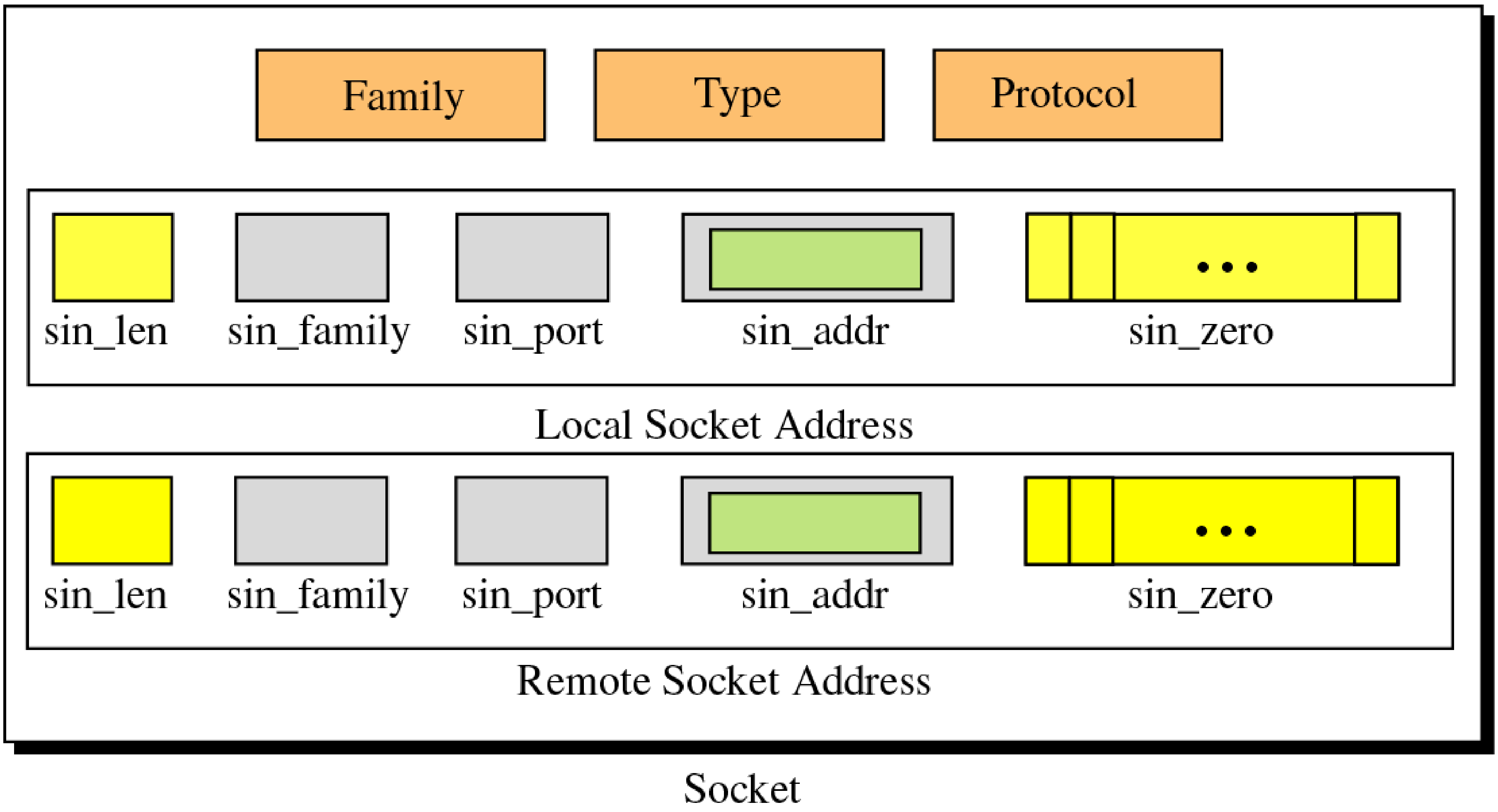
# Socket address structure



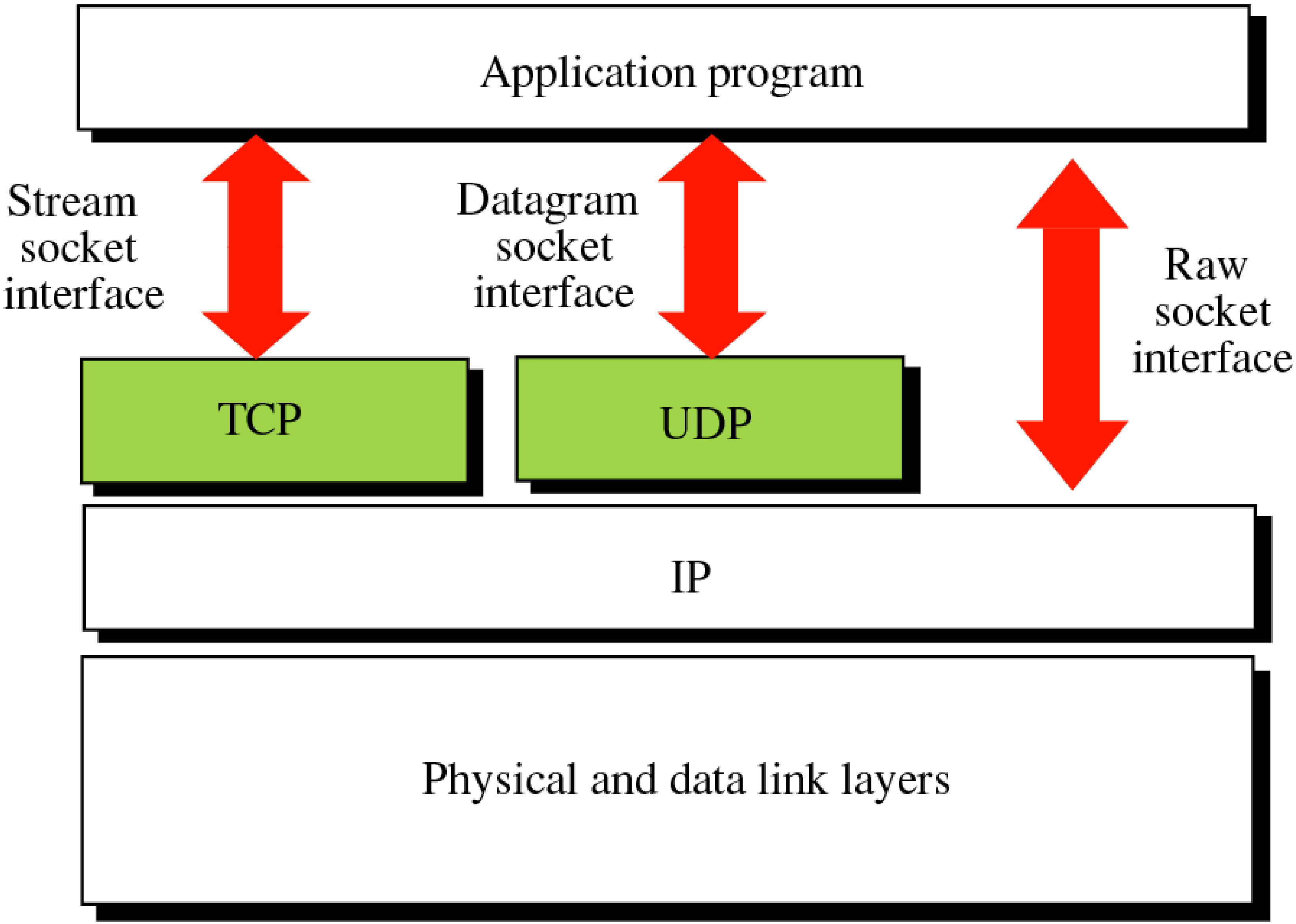

**sockaddr_in**

```
struct   sockaddr_in
{
        u_char                  sin_len ;
        u_short                 sin_family ;
        u_short                 sin_port ;
        struct  in_addr         sin_addr ;
        char                    sin_zero [8] ;
} ;
```

# Socket Structure

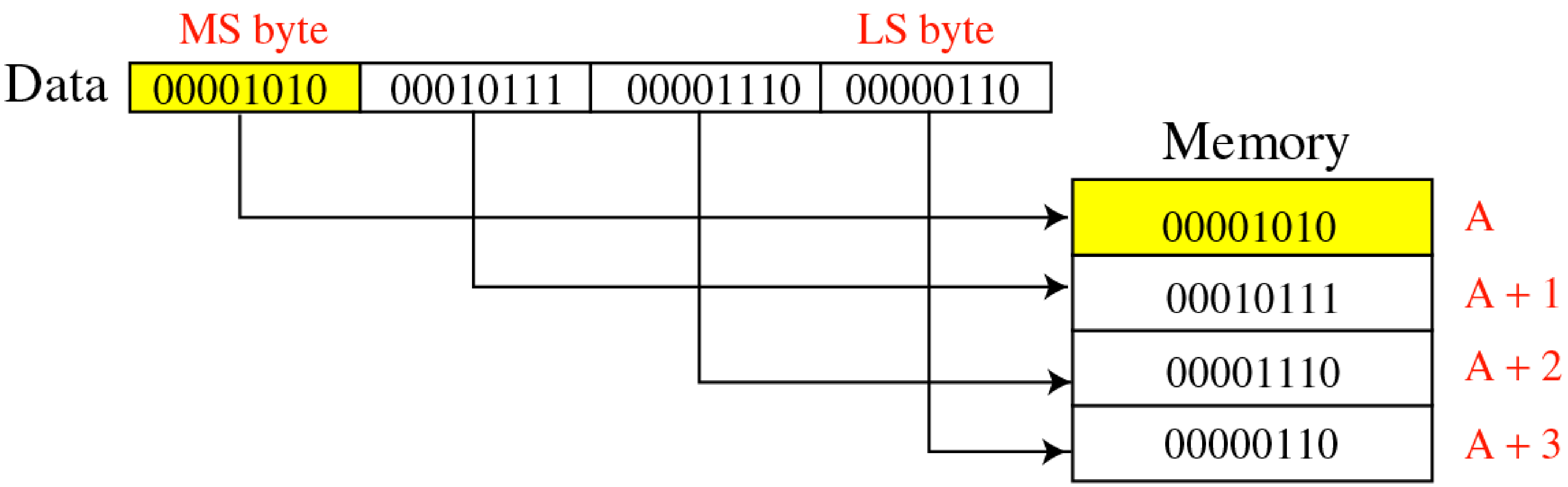


# Socket Types



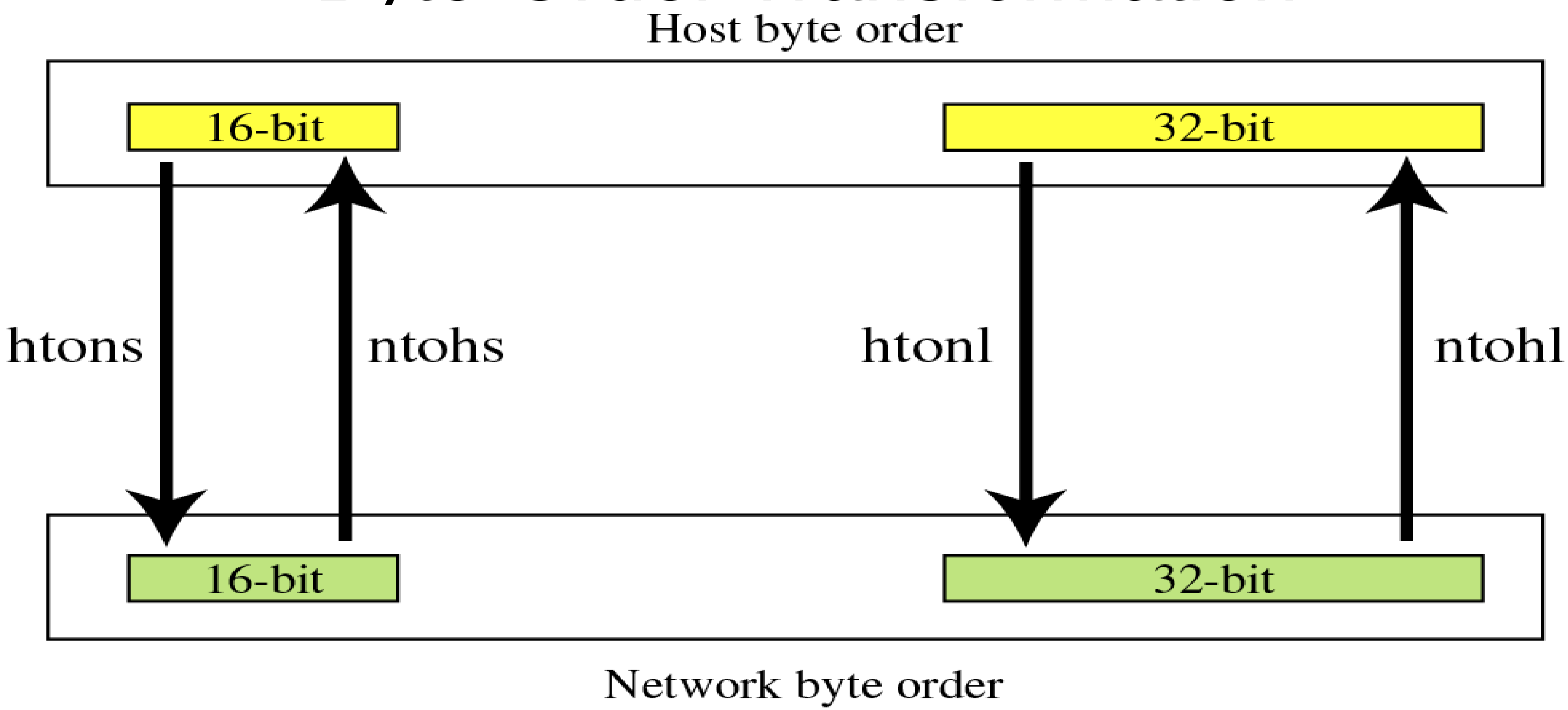
# Byte ordering

- Big Endian byte-order




The byte order for the TCP/IP protocol suite is big endian.

# Byte-Order Transformation




u_short **htons** ( u_short ***host_short*** ) ;

u_short **ntohs** ( u_short ***network_short*** ) ;

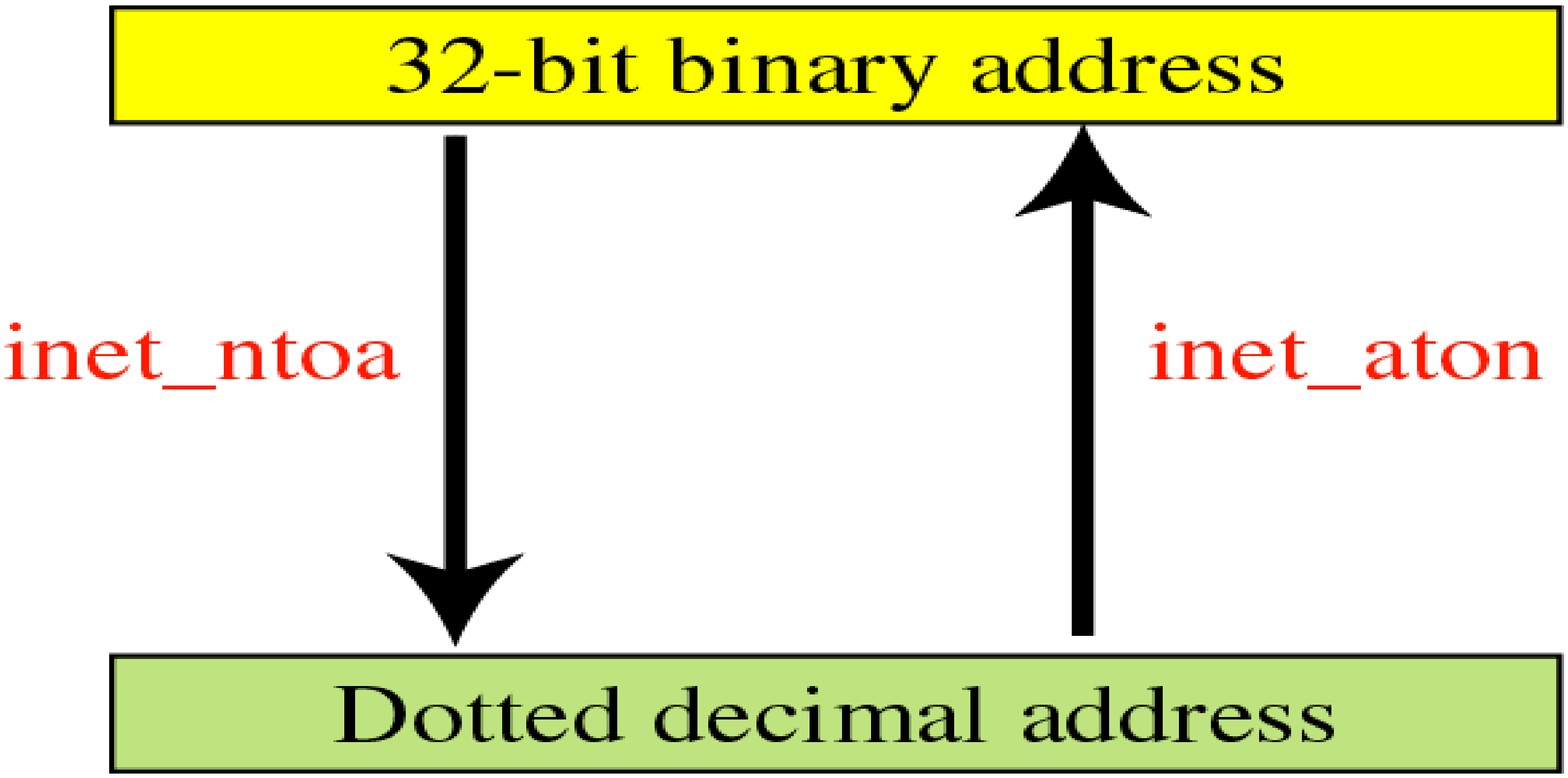
u_long **htonl** ( u_long ***host_long*** ) ;

u_long **ntohl** ( u_long ***network_long*** ) ;

# Address Transformation

int inet_aton ( const char *strptr , struct in_addr *addrptr ) ;

char *inet_ntoa (struct in_addr inaddr ) ;



# Byte-Manipulation Functions

- In network programming, we often need to initialize a field, copy the contents of one field to another, or compare the contents of two fields.
  - Cannot use string functions (strcpy, strcmp, …) which assume null character termination.

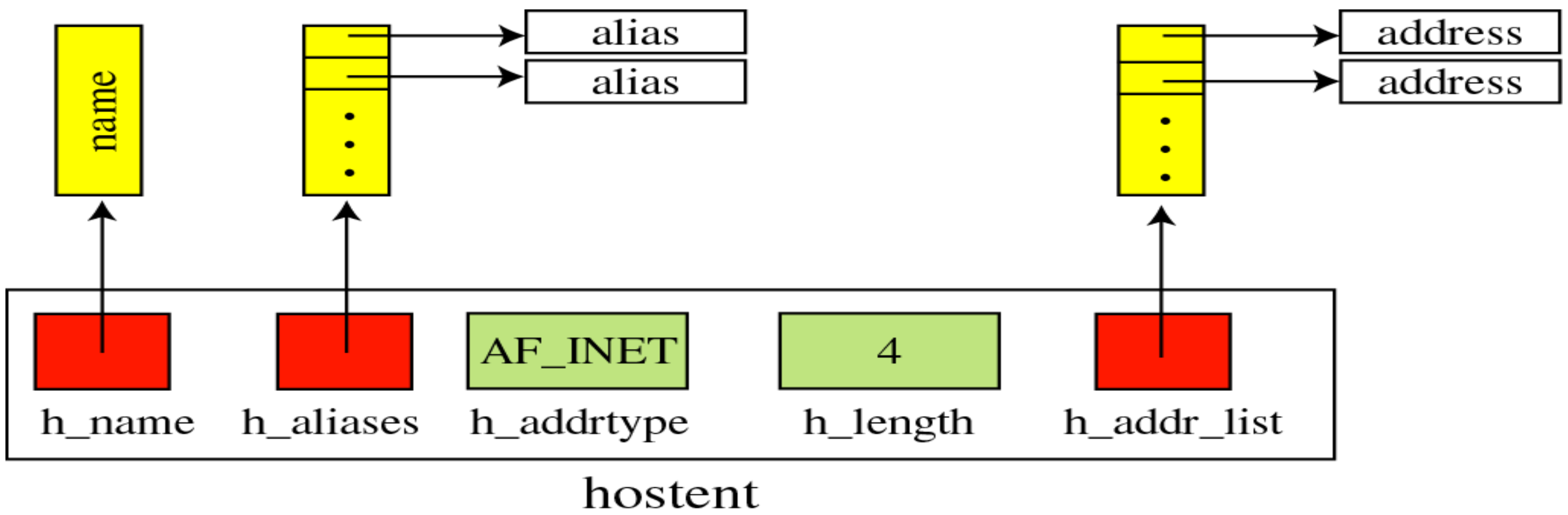
void ***memset** ( void ***dest* , int *chr* , int *len* ) ;

void ***memcpy** ( void ***dest* , const void ***src* , int *len* ) ;

int **memcmp** ( const void ***first* , const void ***second* , int *len* ) ;

# Information about remote host

struct hostent ***gethostbyname** ( const char ****hostname*** ) ;




```
struct      hostent
{
            char        *h_name ;
            char        **h_aliases ;
            int         h_addrtype ;
            int         h_length ;
            char        **h_addr_list ;
} ;
```

# Procedures That Implement The Socket API

## Creating and Deleting Sockets

- `fd=`**`socket`**`(protofamily, type, protocol)`
  Creates a new socket. Returns a file descriptor (fd). Must specify:
    - the protocol family (e.g. TCP/IP)
    - the type of service (e.g. STREAM or DGRAM)
    - the protocol (e.g. TCP or UDP)
- **`close`**(fd)
  Deletes socket.
  For connected STREAM sockets, sends EOF to close connection.

# Procedures That Implement The Socket API

## Putting Servers "on the Air"

- **`bind`**(fd)
  Used by server to establish port to listen on.
  When server has >1 IP addrs, can specify "ANY", or a specific one
- **`listen`** (fd, queuesize)
  Used by connection-oriented servers only, to put server "on the air"
  Queuesize parameter: how many pending connections can be waiting
- afd = **`accept`** (lfd, caddress, caddresslen)
  Used by connection-oriented servers to accept one new connection
  - There must already be a listening socket (lfd)
  - Returns afd, a new socket for the new connection, and
  - The address of the caller (e.g. for security, log keeping. etc.)

# How Clients Communicate with Servers

- **`connect`** (fd, saddress, saddreslen)

  Used by connection-oriented clients to connect to server

  - There must already be a socket bound to a connection-oriented service on the fd
  - There must already be a listening socket on the server
  - You pass in the address (IP address, and port number) of the server.

  Used by connectionless clients to specify a "default send to address"

  - Subsequent "writes" or "sends" don't have to specify a destination address
  - BUT, there really ISN'T any connection established… this is a bad choice of names!

# How Clients Communicate with Servers

- **send** (fd, data, length, flags)
  **sendto** (fd, data, length, flags, destaddress, addresslen)
  **sendmsg** (fd, msgstruct, flags)
  **write** (fd, data, length)

  Used to send data.
  - **send** requires a connection (or for UDP, default send address) be already established
  - **sendto** used when we need to specify the dest address (for UDP only)
  - sendmsg is an alternative version of sendto that uses a struct to pass parameters
  - write is the “normal” write function; can be used with both files and sockets
- **recv** (…) **recvfrom** (…) **recvmsg** (…) **read** (…)

  Used to receive data… parameters are similar, but in reverse

  (destination => source, etc…)

# Connectionless Service (UDP)

**Server**

1. Create transport endpoint: **socket()**
2. Assign transport endpoint an address: **bind()**
3. Wait for a packet to arrive: **recvfrom()**
4. Formulate reply (if any) and send: **sendto()**
5. Release transport endpoint: **close()**

Client

1. Create transport endpoint: **socket()**
2. Assign transport endpoint an address (optional): **bind()**
3. Determine address of server
4. Formulate message and send: **sendto()**
5. Wait for packet to arrive: **recvfrom()**
6. Release transport endpoint: **close()**

**Server**

1. Create transport endpoint for incoming connection request: **socket()**

2. Assign transport endpoint an address: **bind( )**

3. Announce willing to accept connections: **listen( )**

4. Block and Wait for incoming request: **accept( )**

5. Wait for a packet to arrive: **recv ( )**

6. Formulate reply (if any) and send: **send( )**

7. Release transport endpoint: **close( )**

**Client**

1. Create transport endpoint: **socket( )**

2. Assign transport endpoint an address (optional): **bind( )**

3. Determine address of server

4. Connect to server: **connect( )**

4. Formulate message and send: **send ( )**

5. Wait for packet to arrive: **recv( )**

6. Release transport endpoint: **close( )**

CONNECTION-ORIENTED SERVICE

# Lab this week

- Grab the simple client-server demo in C from my github
- If you run the server code on one local terminal and the client code on another, it transmits a Hello World message and quits
- Lab tasks
  - Make the server persistent, such that it doesn't quit after one message transfer
  - Redesign the server such that it accepts inputs from the client, transforms them into lowercase and transmits back
  - Redesign the server and client such that
    - The client requests an HTML file from the server that is displaying some headers and a simple table
    - The server reads the file stored in the server's computer and transmits it as text to the client
    - Write a shell script that will display the corresponding webpage on the client's computer
  - Create a version of the server that will allow connectionless (UDP) transfer
    - Client code will have to change too
    - See if you can get the HTML transfer to work for this UDP connection also
  - Create a version of the TCP server that will permit concurrent connections from multiple clients