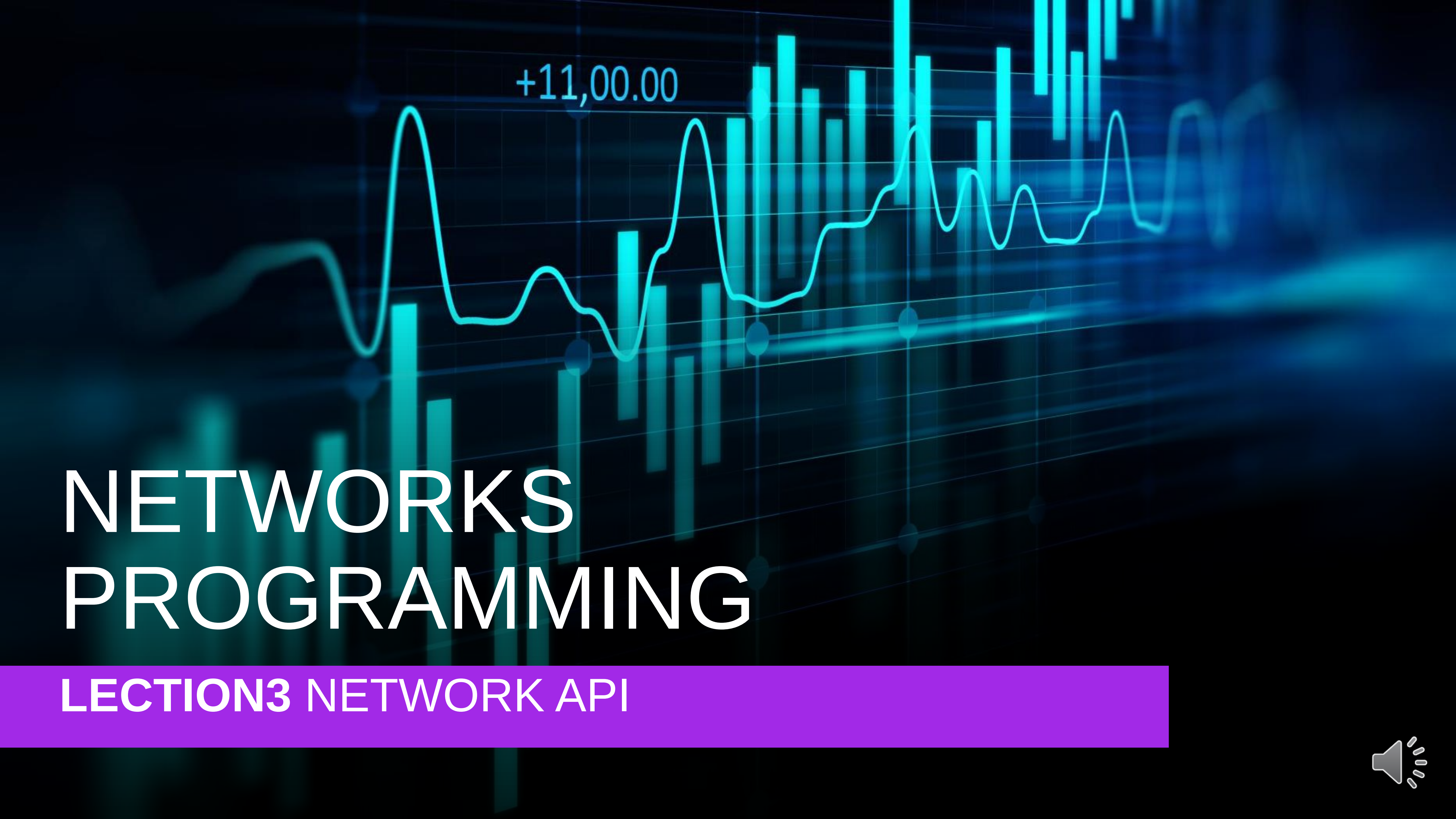


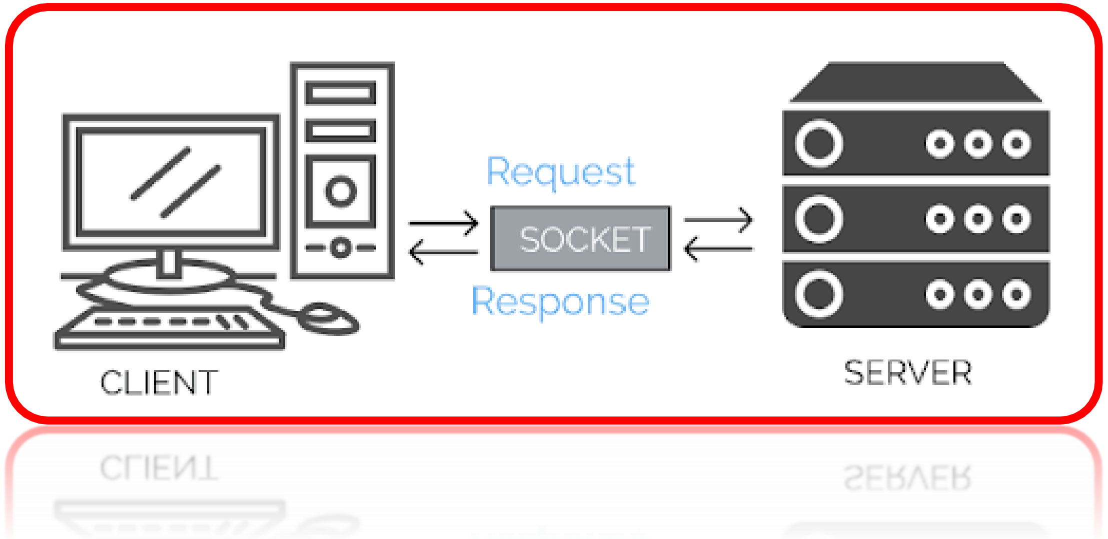
NETWORKS PROGRAMMING

LECTION3 NETWORK API

A **network socket** is an internal endpoint for sending or receiving data within a node on a computer network.

LECTION OBJECTIVES

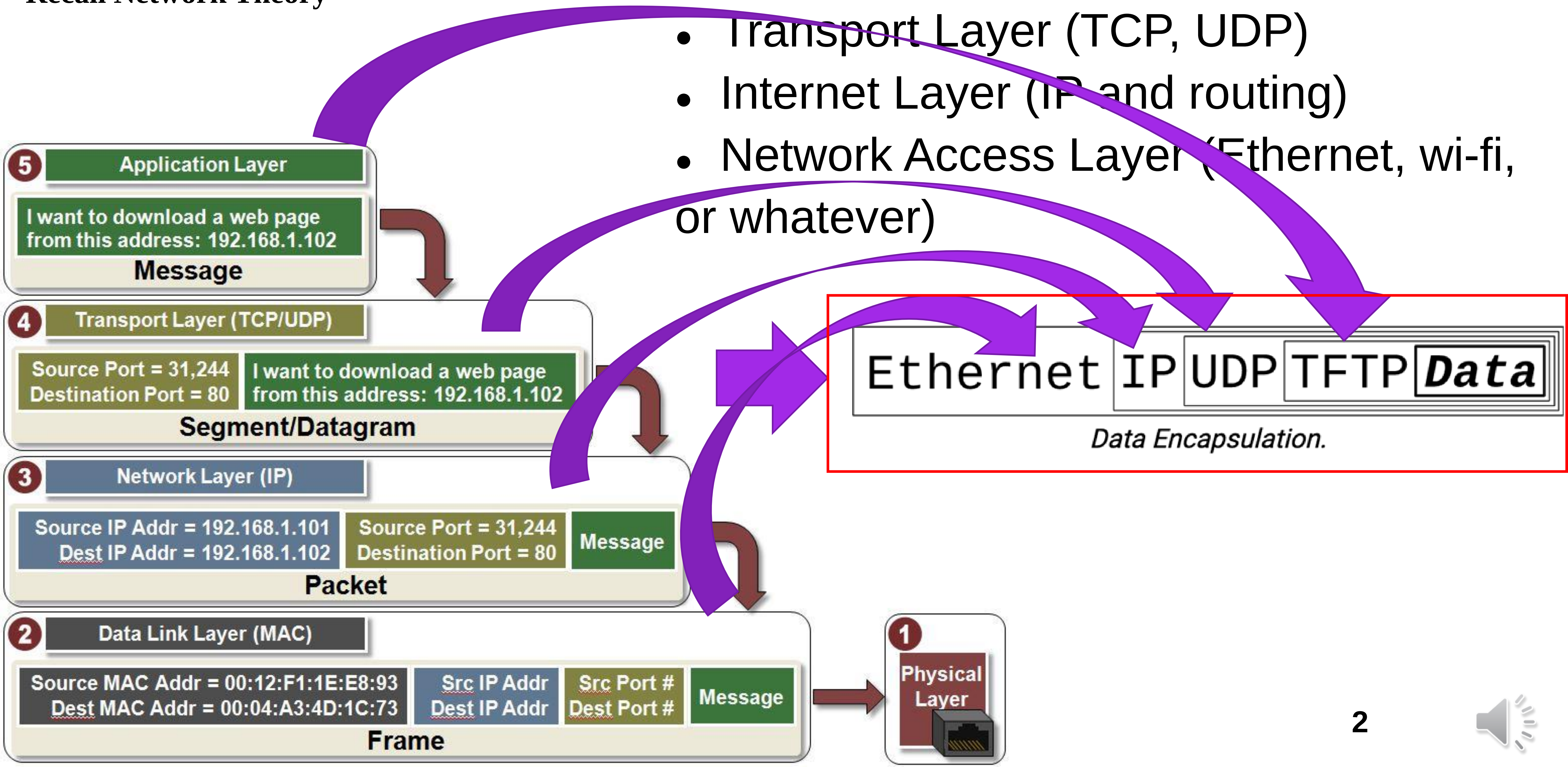
- What are Sockets in Network Programs
- IPv4 vs IPv6
- Byte Order
- Network API in details
- Client-Server Communication
- Practice



The term *socket* is analogous to physical female connectors, communication between two nodes through a channel being visualized as a cable with two male connectors plugging into sockets at each node.

Recall Network Theory

- Application Layer (telnet, ftp, etc.)
- Transport Layer (TCP, UDP)
- Internet Layer (IP and routing)
- Network Access Layer (Ethernet, wi-fi, or whatever)



5 Application Layer

I want to download a web page from this address: 192.168.1.102

Message

4 Transport Layer (TCP/UDP)

Source Port = 31,244
Destination Port = 80

I want to download a web page from this address: 192.168.1.102

Segment/Datagram

3 Network Layer (IP)

Source IP Addr = 192.168.1.101
Dest IP Addr = 192.168.1.102

Source Port = 31,244
Destination Port = 80

Message

Packet

2 Data Link Layer (MAC)

Source MAC Addr = 00:12:F1:1E:E8:93
Dest MAC Addr = 00:04:A3:4D:1C:73

Src IP Addr
Dest IP Addr

Src Port #
Dest Port #

Message

Frame

1 Physical Layer

Ethernet IP UDP TFTP ***Data***

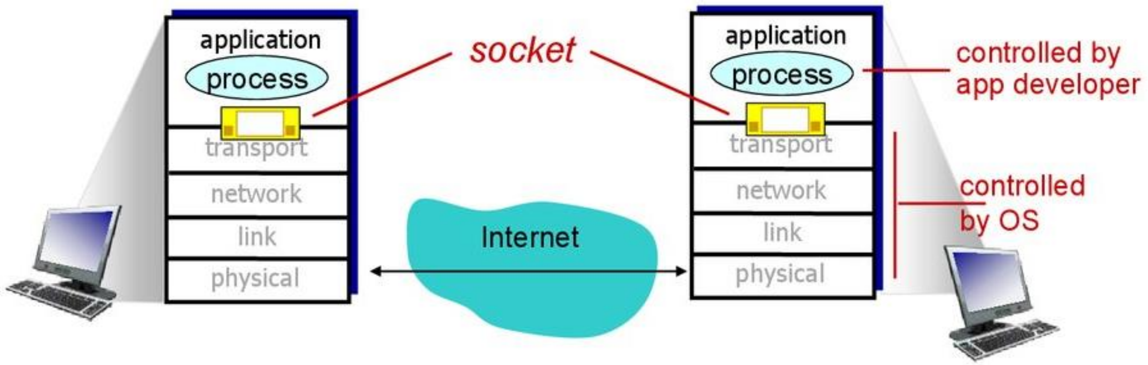
Data Encapsulation.

What is a socket!?

- **Socket**: a way to speak to other programs using standard Unix file descriptors
- **File descriptor** is simply an integer associated with an open file. But (and here's the catch), that file can be a network connection, a FIFO, a pipe, a terminal, a real on-the-disk file, or just about anything else. Everything in Unix is a file!
- Make a call to the **socket**() system routine. It returns the socket descriptor, and you communicate through it using the specialized send() and recv() (man send, man recv) socket calls.

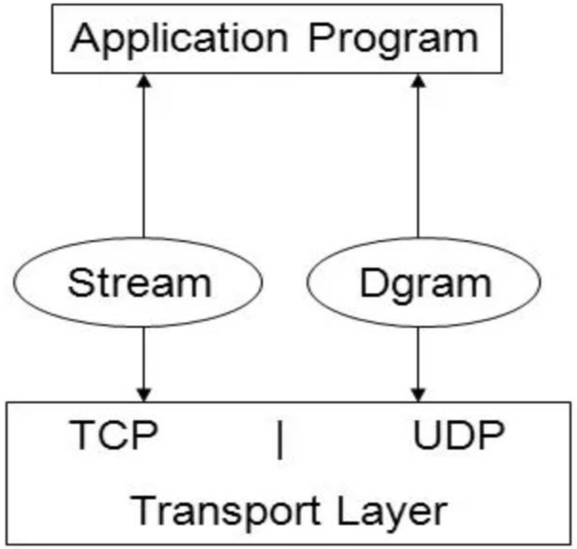
Command for use in the terminal get the documentation.

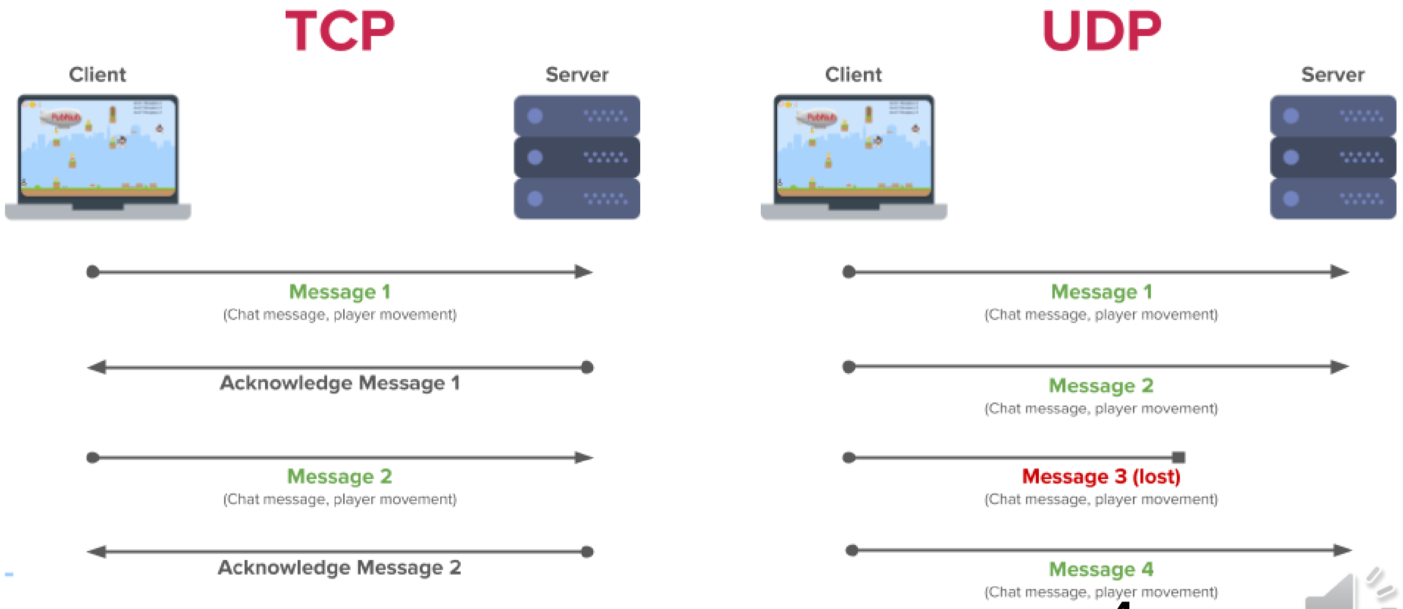
socket: door between application process and end-end-transport protocol



Two types of internet sockets

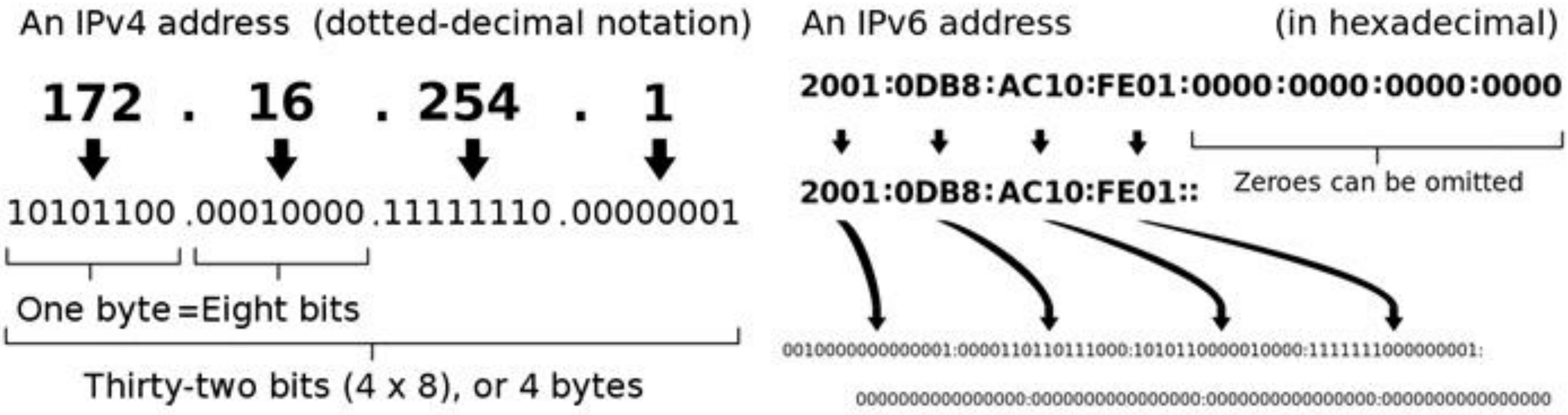
- **SOCK_STREAM** (TELNET, HTTP, MAIL, … over TCP) - Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported
- **SOCK_DGRAM** (TFTP, DHCP, video streaming, audio streaming, …) - supports datagrams (connectionless, unreliable messages of a fixed maximum length)
- …




TCP
Client
Server
Message 1
(Chat message, player movement)
Acknowledge Message 1
Message 2
(Chat message, player movement)
Acknowledge Message 2
UDP
Client
Server
Message 1
(Chat message, player movement)
Message 2
(Chat message, player movement)
Message 3 (lost)
(Chat message, player movement)
Message 4
(Chat message, player movement)

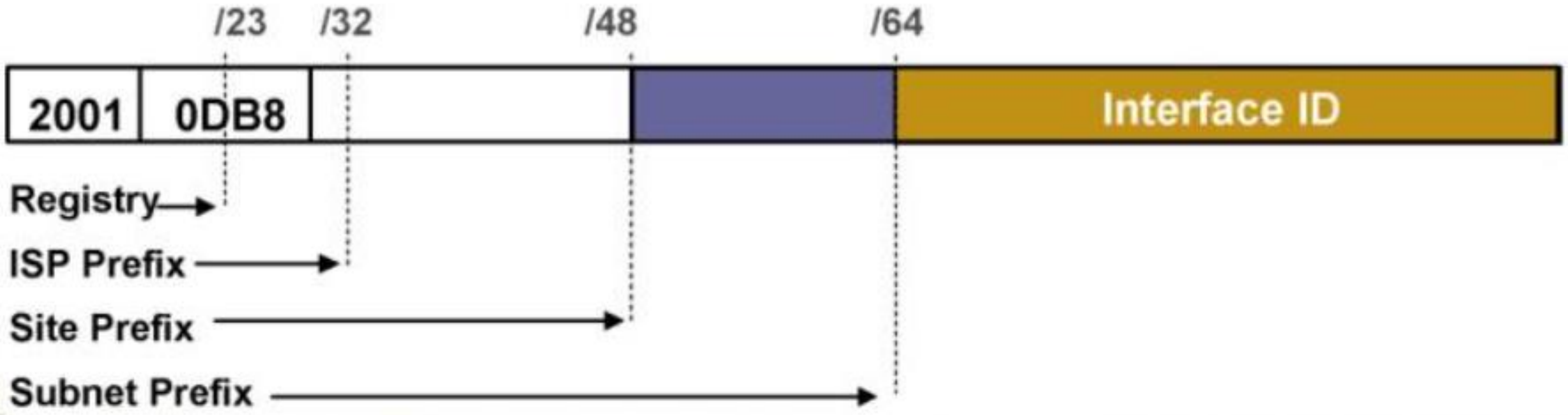
IP address v4/v6

- 192.0.2.111 - IPv4
- 2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551 - IPv6
- The address ::1 is the loopback address. It always means “this machine I’m running on now”. In IPv4, the loopback address is 127.0.0.1.



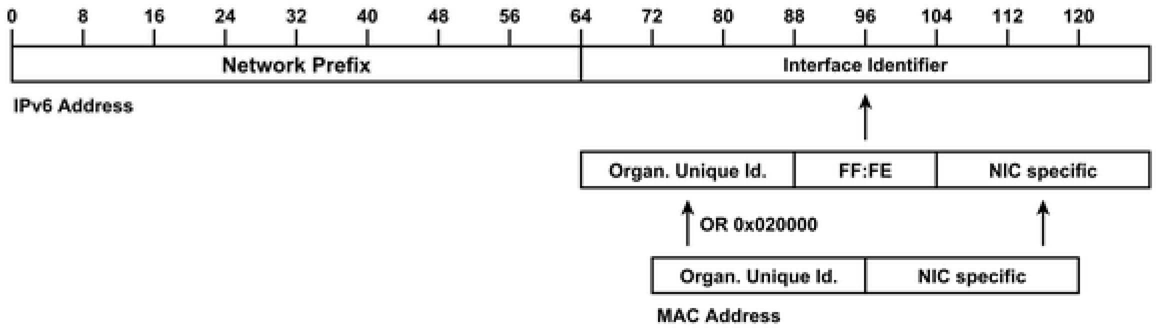
The IPv6 Address Space

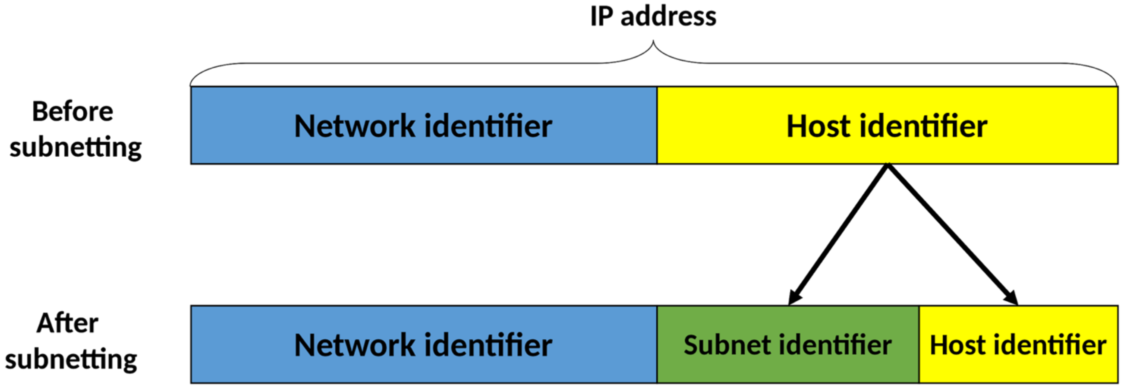
- 128-bit address space
- 128 bits were chosen to allow multiple levels of hierarchy and flexibility in designing hierarchical addressing and routing
- Global unicast and anycast addresses are defined by a global routing prefix, a subnet ID, and an interface ID

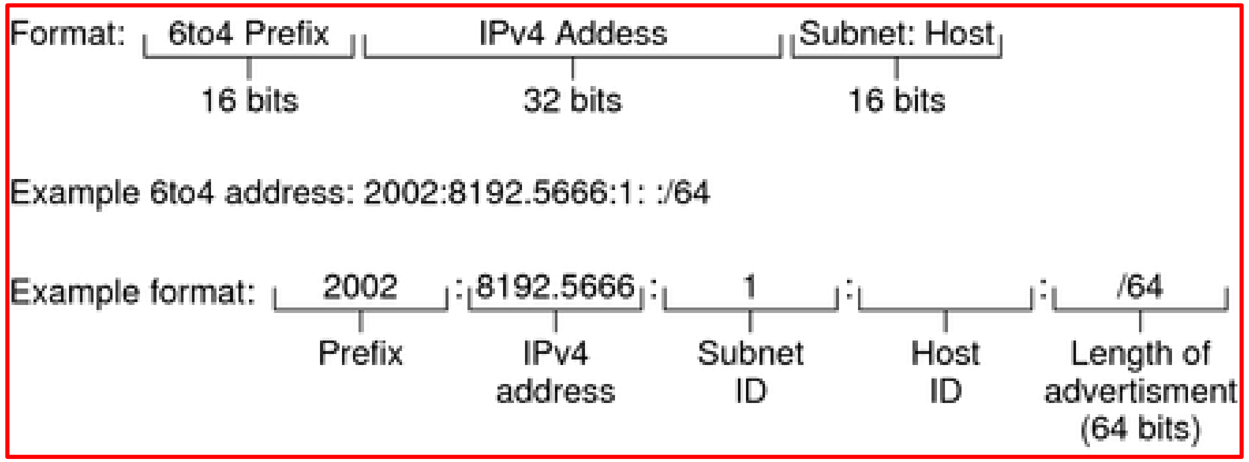


IP address v4/v6

- Subnets - Mask & Class





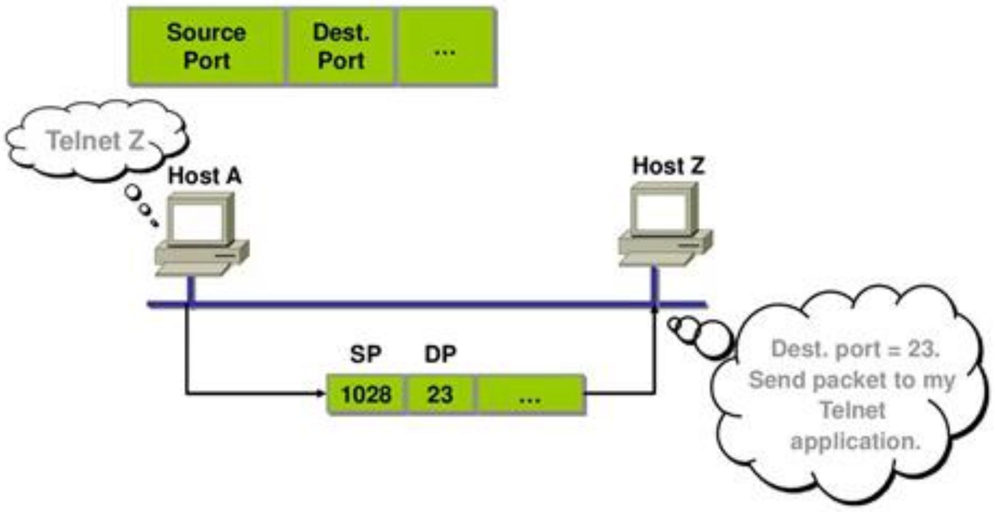


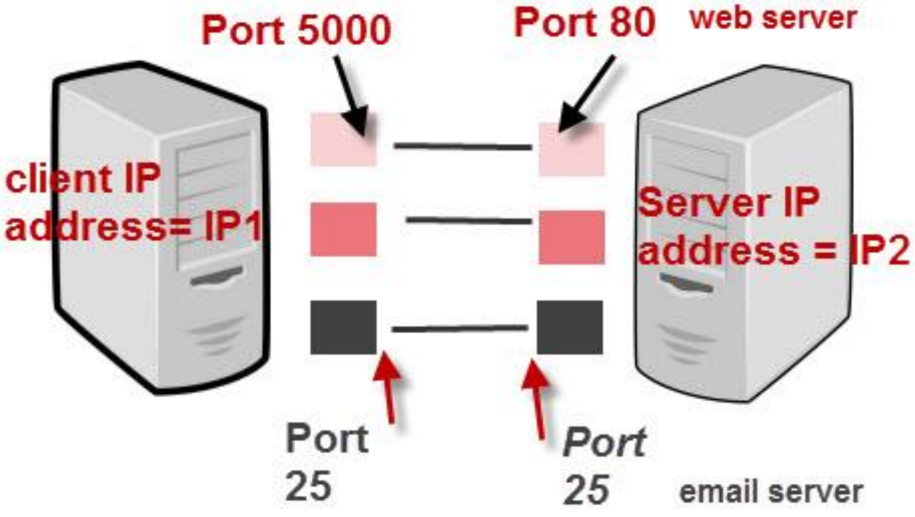
Example 6to4 address: 2002:8192.5666:1: :/64

Port numbers

- **TCP/UDP** (transport level)
- **Port number** is a 16-bit number that's like the local address for the connection
- HTTP 80, TELNET 23, SMTP 25, DOOM 666
- < 1024 reserved

TCP Port Numbers



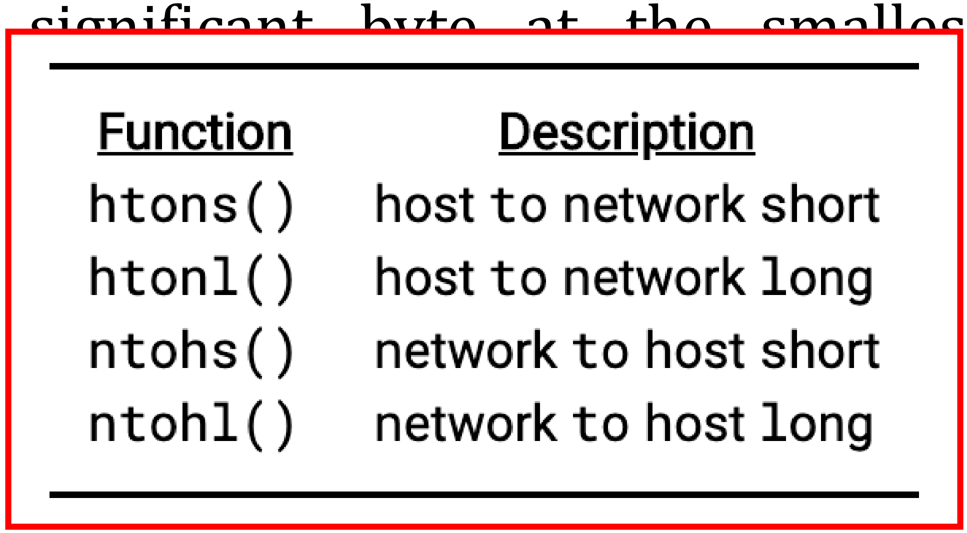


IP Address + Port number = Socket

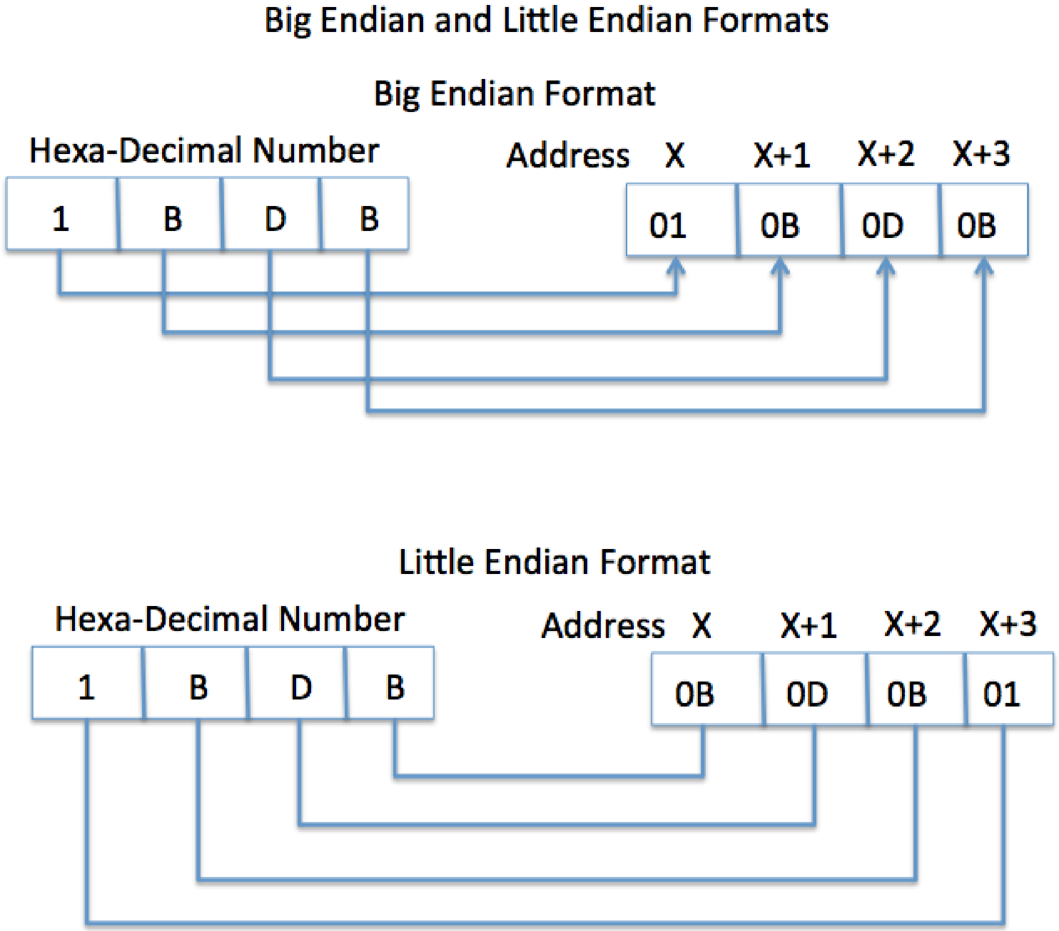
TCP/IP Ports And Sockets

Byte Order

- Computer might have been storing bytes in reverse order behind your back
- **Big-Endian** is also called Network Byte Order because that's the order us network types like
- **Little-Endian** stores the least-significant byte at the smallest



Function	Description
htons()	host to network short
htonl()	host to network long
ntohs()	network to host short
ntohl()	network to host long



Big Endian and Little Endian Formats

Structures

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

Structures

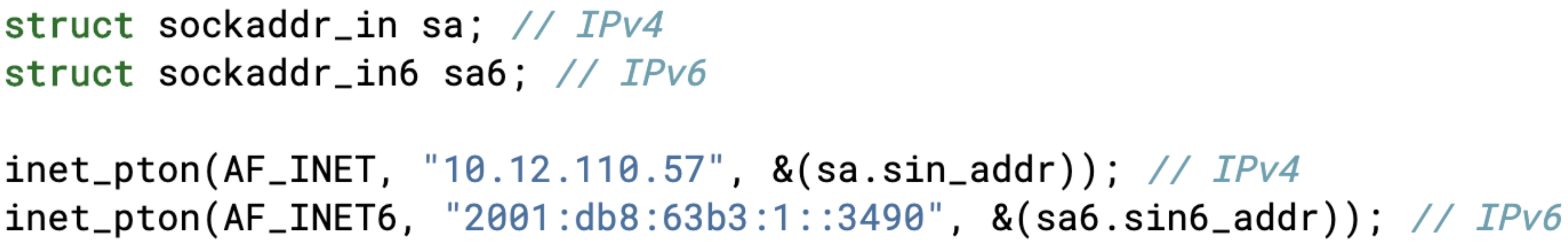
```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};

 struct sockaddr_in {
     short int          sin_family;  // Address family, AF_INET
     unsigned short int sin_port;    // Port number
     struct in_addr     sin_addr;    // Internet address
     unsigned char      sin_zero[8]; // Same size as struct sockaddr
 };
// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

Structures

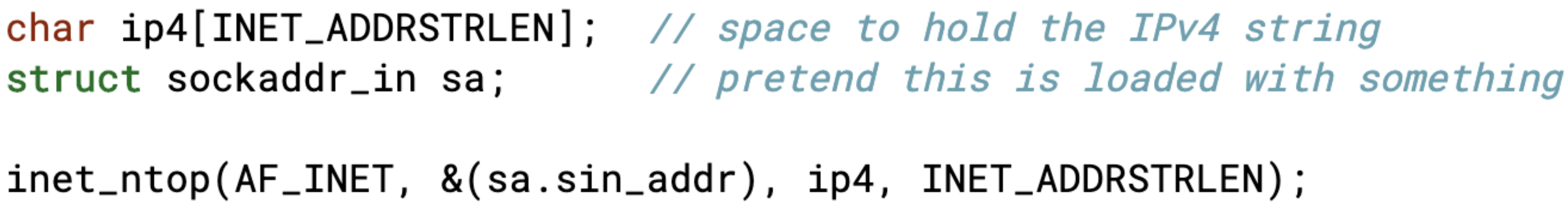
```
struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // IPv6 address
};
```

IP addresses convertion From IP4 to IP6

- No need to figure them out by hand and stuff them in a long with the in operator.
- **inet_pton**(), converts an IP address in numbers-and-dots notation into either a struct in_addr or a struct in6_addr depending on whether you specify AF_INET or AF_INET6
- **inet_ntop**(), reverse transformation



```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "10.12.110.57", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```



```
char ip4[INET_ADDRSTRLEN];  // space to hold the IPv4 string
struct sockaddr_in sa;      // pretend this is loaded with something

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);
```

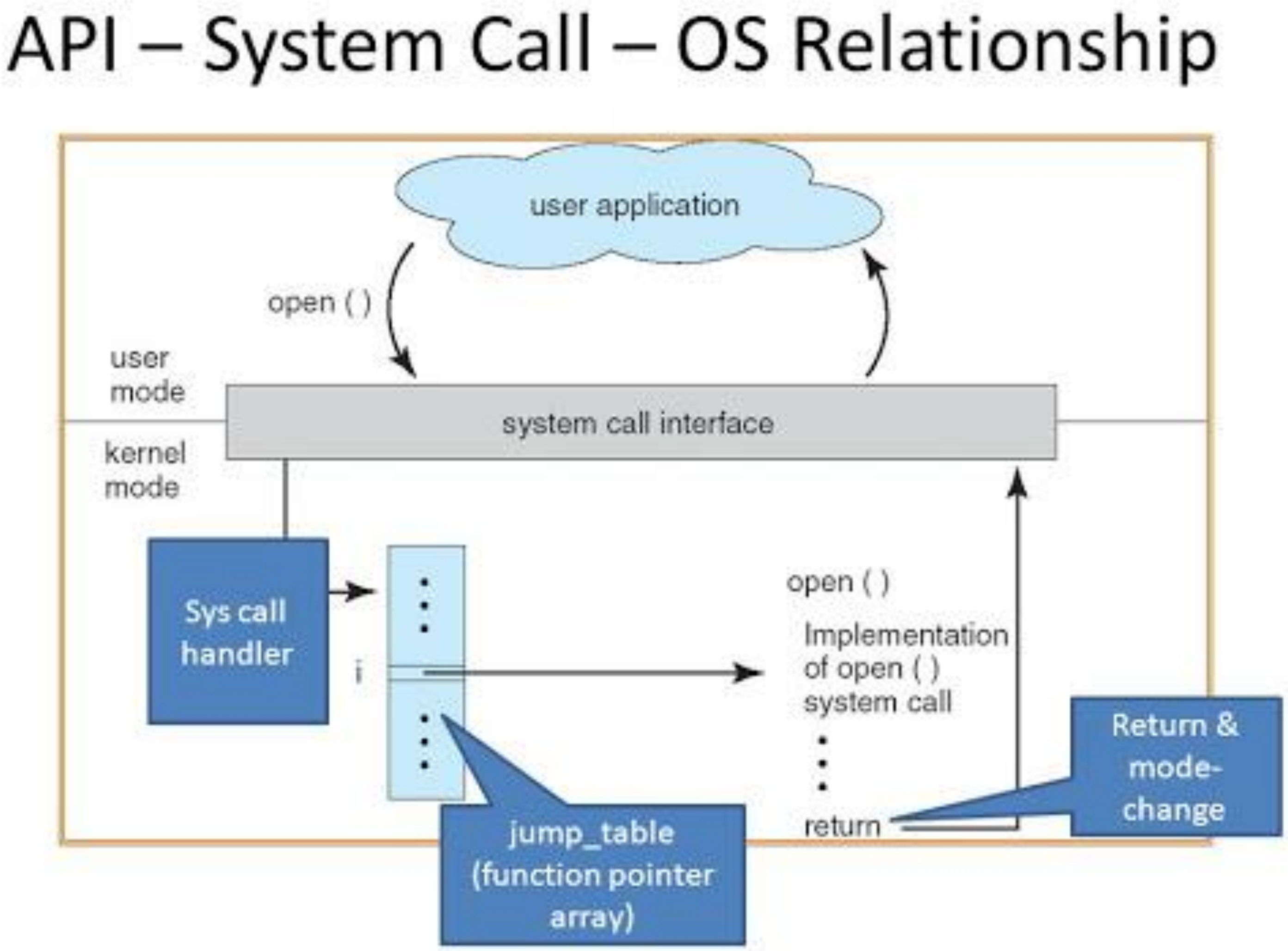
IP addresses convertion From IP4 to IP6

1. First of all, try to **use getaddrinfo()** to get all the struct sockaddr info, instead of packing the structures by hand.
2. Any place that you find you’re hard-coding anything related to the IP version, try to wrap up in a helper function.
3. Change **AF_INET to AF_INET6**.
4. Change **PF_INET to PF_INET6**.
5. Change the assignments of **INADDR_ANY to IN6ADDR_ANY**
6. Instead of struct sockaddr_in **use struct sockaddr_in6**
7. Instead of struct in_addr **use struct in6_addr**
8. Instead of inet_aton() or inet_addr(), **use inet_pton()**.
9. Instead of inet_ntoa(), **use inet_ntop()**.
10. Instead of gethostbyname(), **use** the superior **getaddrinfo()**.
11. Instead of gethostbyaddr(), **use** the superior **getnameinfo() (although gethostbyaddr() can still work with IPv6)**
12. INADDR_BROADCAST no longer works. **Use IPv6 multicast** instead.

System calls

System call provides the services of the operating system to the user programs via Application Program Interface(API).

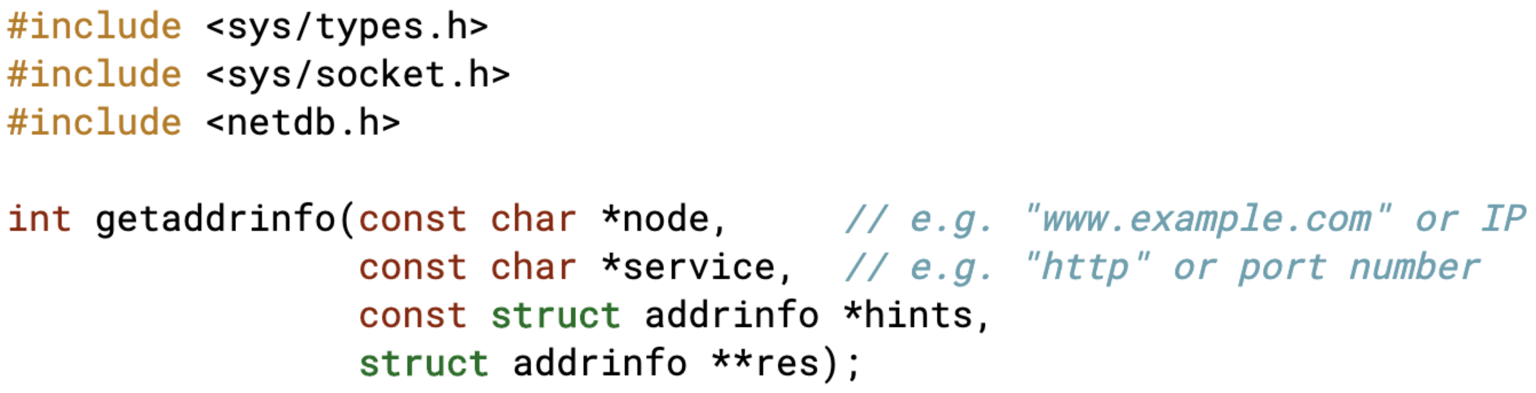
It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system.



API – System Call – OS Relationship

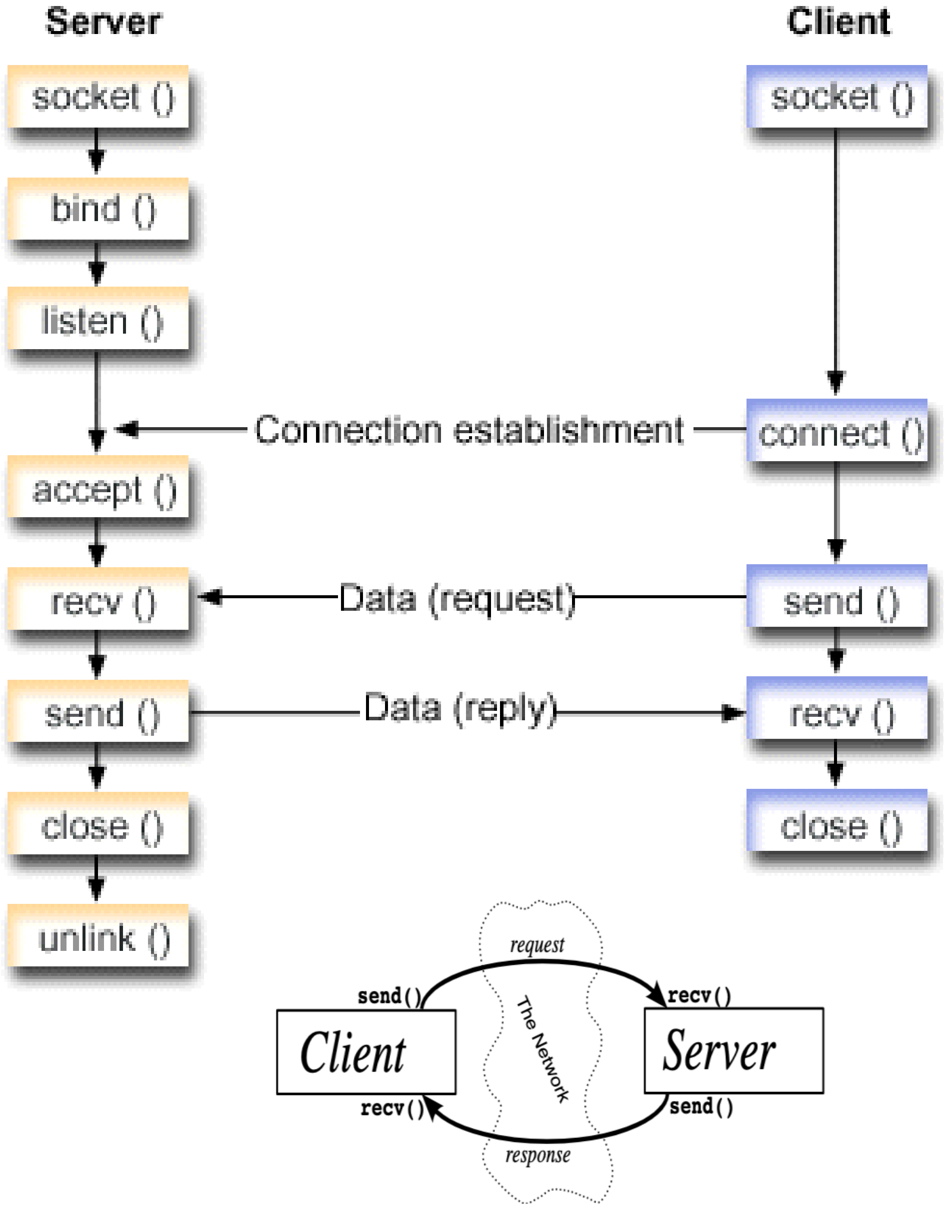
getaddrinfo

- DNS and service name lookups
- Give this function three input parameters, and it gives you a pointer to a linked-list, res, of results.
- The node parameter is the host name to connect to, or an IP address.
- service, which can be a port number, like “80”, or the name of a particular service like “http” or “ftp” or “telnet” or “smtp” or whatever.
- Finally, the hints parameter points to a struct addrinfo that you’ve already filled out with relevant information.



```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node,     // e.g. "www.example.com" or IP
                const char *service,  // e.g. "http" or port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

State Diagram for Server and Client Model



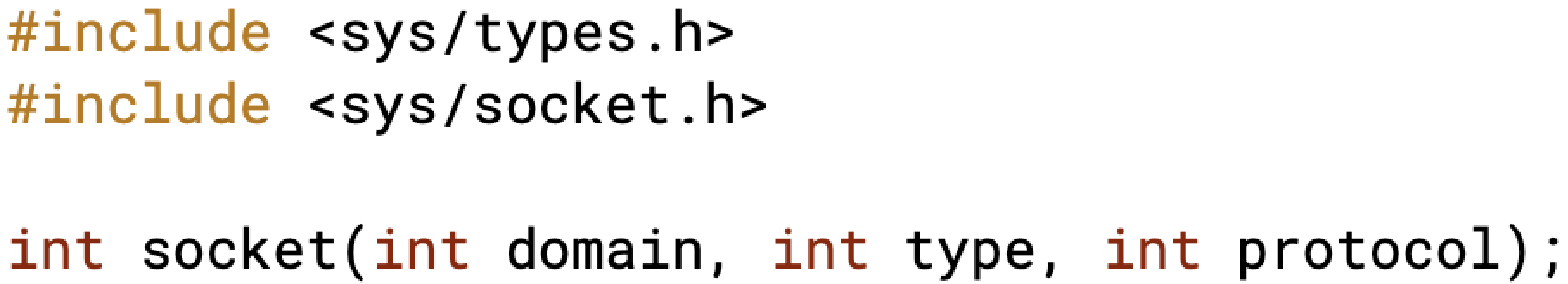
Client-Server Interaction.

Server program - a software component of a computing system that performs service (maintenance) functions at the request of a client, giving him access to certain resources or services.

The **client program** interacts with the server using a specific protocol. It can request any data from the server, manipulate data directly on the server, start new processes on the server, etc. The client program can provide data received from the server to the user or use it in some other way, depending on the purpose of the program.

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket (node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

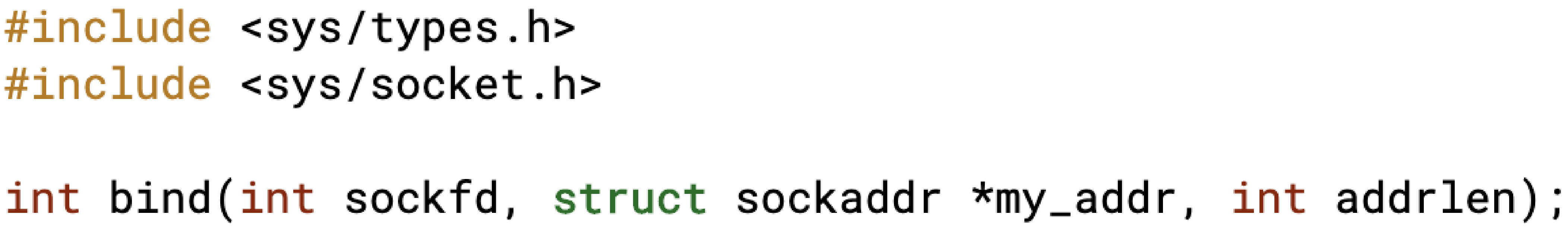
socket



```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

- Get the File Descriptor
- **domain** is PF_INET or PF_INET6
- **type** is SOCK_STREAM or SOCK_DGRAM
- **protocol** can be set to 0 to choose the proper protocol for the given type
- socket() simply returns to you a socket descriptor that you can use in later system calls, or -1 on error. The global variable errno is set to the error’s value.

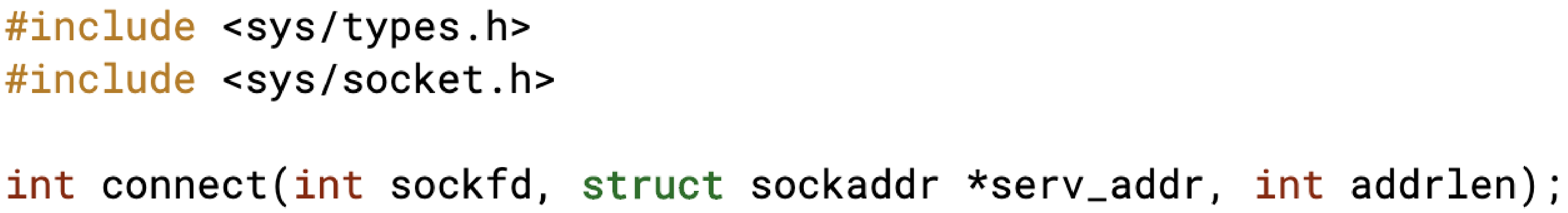
bind



```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- Once you have a socket, you might have to associate that socket with a port on your local machine.
- **sockfd** is the socket file descriptor returned by socket(). **my_addr** is a pointer to a struct sockaddr that contains information about your address, namely, port and IP address. **addrlen** is the length in bytes of that address.

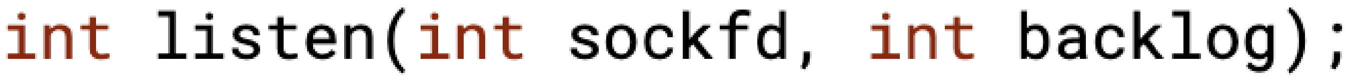
connect



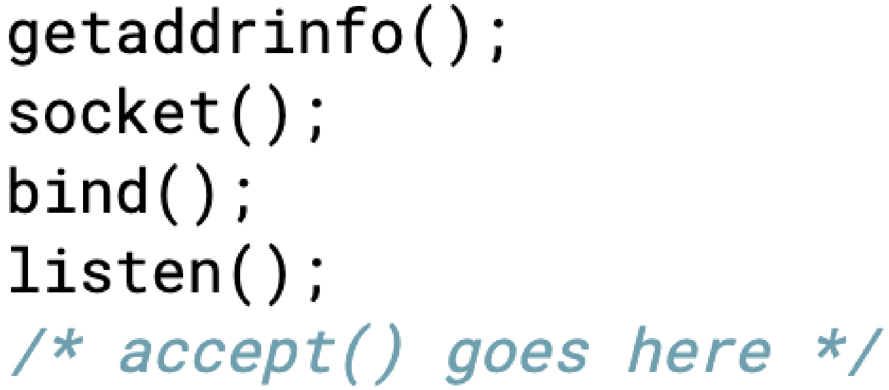
```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

- how to connect to a remote host
- **sockfd** is our friendly neighborhood socket file descriptor
- **serv_addr** is a struct sockaddr containing the destination port and IP address
- **addrlen** is the length in bytes of the server address structure.

listen



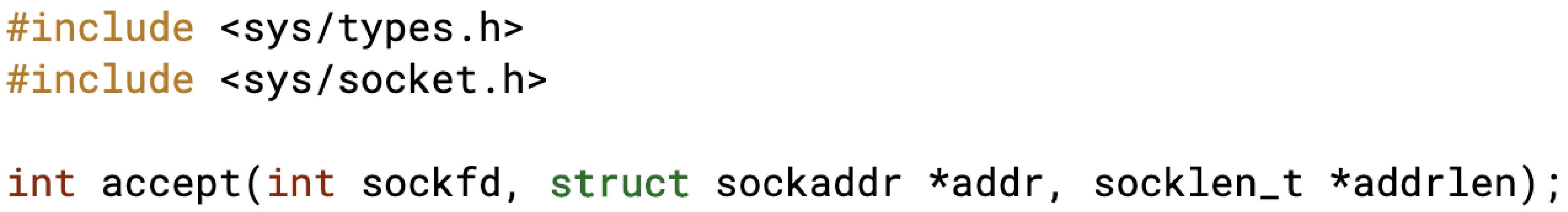
```
int listen(int sockfd, int backlog);
```



```
getaddrinfo();
socket();
bind();
listen();
/* accept() goes here */
```

- **sockfd** is the usual socket file descriptor from the socket() system call
- **backlog** is the number of connections allowed on the incoming queue
- probably get away with setting it to 5 or 10

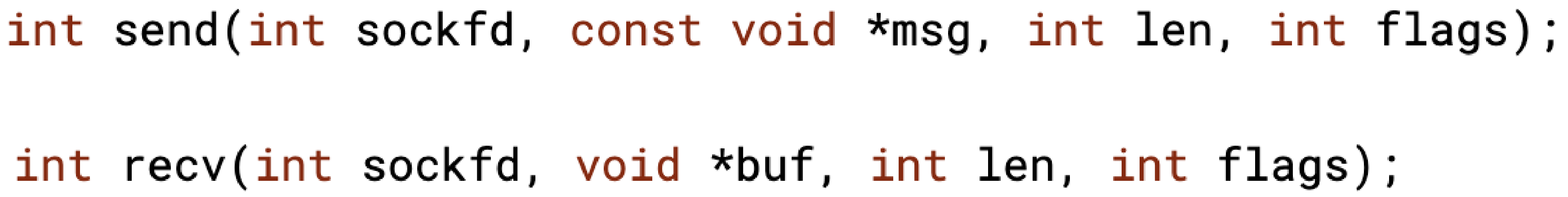
accept



```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- connection will be queued up waiting to be accept()ed
- **sockfd** is the listen()ing socket descriptor
- **addr** will usually be a pointer to a local struct sockaddr_storage. This is where the information about the incoming connection will go (and with it you can determine which host is calling you from which port)
- **addrlen** is a local integer variable that should be set to sizeof(struct sockaddr_storage) before its address is passed to accept()

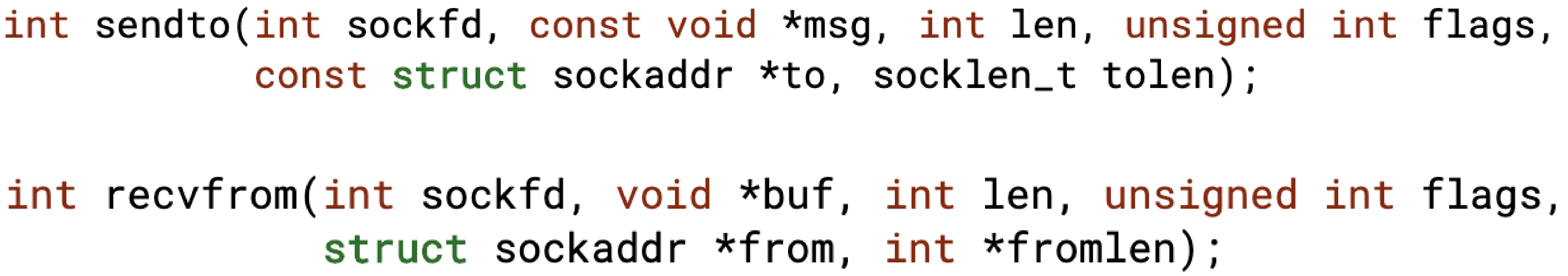
send/recv



```
int send(int sockfd, const void *msg, int len, int flags);

int recv(int sockfd, void *buf, int len, int flags);
```

- these two functions are for communicating over stream sockets or connected datagram sockets
- **sockfd** is the socket descriptor you want to send data
- **msg** is a pointer to the data you want to send
- **len** is the length of that data in bytes
- just set flags to 0
- **buf** is the buffer to read the information into
- **len** is the maximum length of the buffer

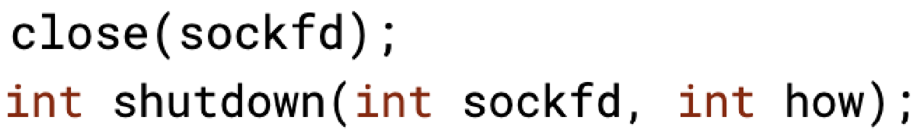
sendto/recvfrom



```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, socklen_t tolen);

int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

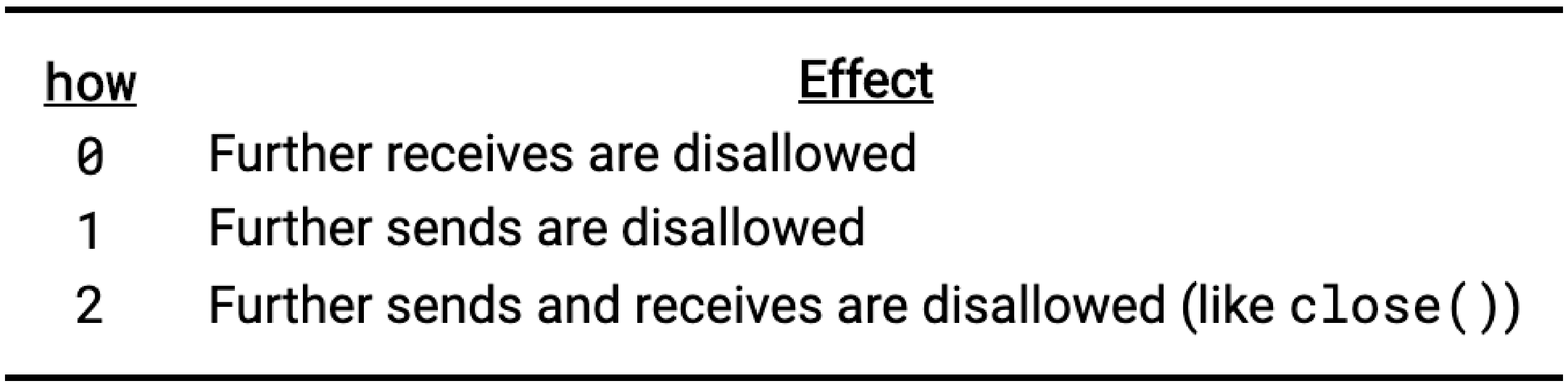
- this call is basically the same as the call to send() with the addition of two other pieces of information
- struct sockaddr (which will probably be another struct sockaddr_in or struct sockaddr_in6 or struct sockaddr_storage that you cast at the last minute) which contains the destination IP address and port

close/shutdown



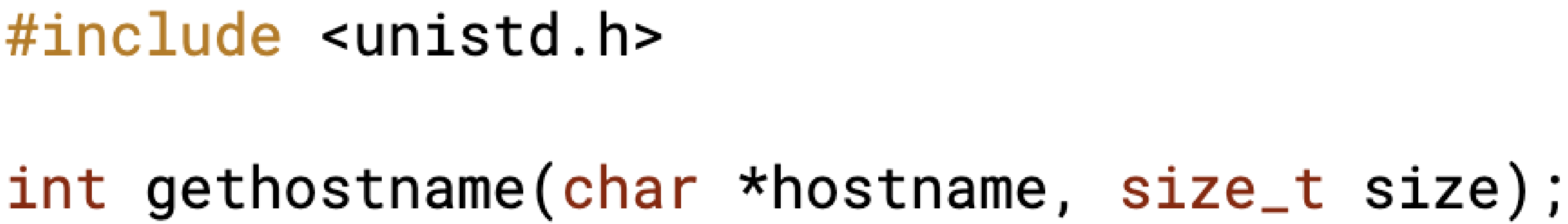
```
close(sockfd);
int shutdown(int sockfd, int how);
```

- close the connection on your socket descriptor
- Just in case you want a little more control over how the socket closes, you can use the shutdown
- **sockfd** is the socket file descriptor you want to shutdown, and **how** is one of the following



how	Effect
0	Further receives are disallowed
1	Further sends are disallowed
2	Further sends and receives are disallowed (like close())

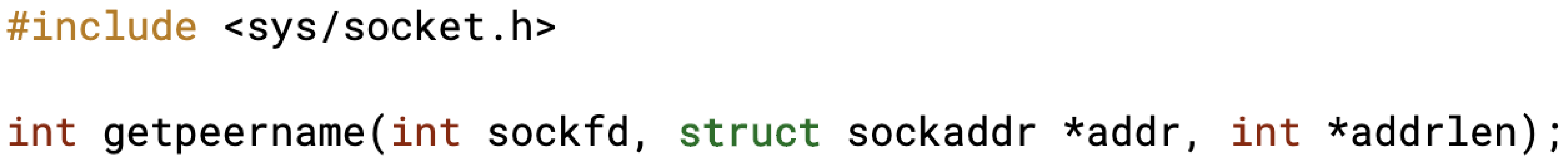
gethostname



```c
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

- It returns the name of the computer that your program is running on
- **hostname** is a pointer to an array of chars that will contain the hostname upon the function’s return
- **size** is the length in bytes of the hostname array

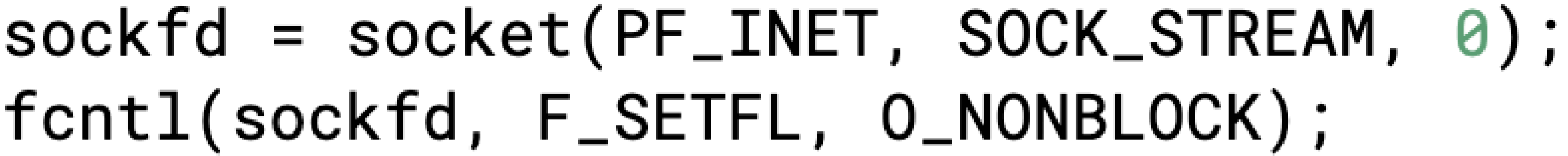
getpeername



```
#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

- the function **getpeername**() will tell you who is at the other end of a connected stream socket
- **sockfd** is the descriptor of the connected stream socket
- **addr** is a pointer to a struct sockaddr
- **addrlen** is a pointer to an int, that should be initialized to sizeof *addr or sizeof

Blocking



```
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

- In a nutshell, “block” is techie jargon for “sleep”
- Lots of functions block: accept, recv
- If you don’t want a socket to be blocking, you have to make a call to fcntl()
- If you put your program in a busy-wait looking for data on the socket, you’ll suck up CPU time like it was going out of style
- A more elegant solution for checking to see if there’s data waiting to be read comes in the following section on poll

select - synchronous I/O multiplexing

```c
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

struct timeval {
    int tv_sec;      // seconds
    int tv_usec;     // microseconds
};
```

Function	Description
`FD_SET(int fd, fd_set *set);`	Add fd to the set.
`FD_CLR(int fd, fd_set *set);`	Remove fd from the set.
`FD_ISSET(int fd, fd_set *set);`	Return true if fd is in the set.
`FD_ZERO(fd_set *set);`	Clear all entries from the set.

poll - synchronous I/O multiplexing

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);

struct pollfd {
    int fd;           // the socket descriptor
    short events;     // bitmap of events we're interested in
    short revents;    // when poll() returns, bitmap of events that occurred
};
```

Macro	Description
`POLLIN`	Alert me when data is ready to `recv()` on this socket.
`POLLOUT`	Alert me when I can `send()` data to this socket without blocking.