

Week 7 Lecture 2

**NWEN 241**

**Systems Programming**

Jyoti Sahni

`jyoti.sahni@ecs.vuw.ac.nz`

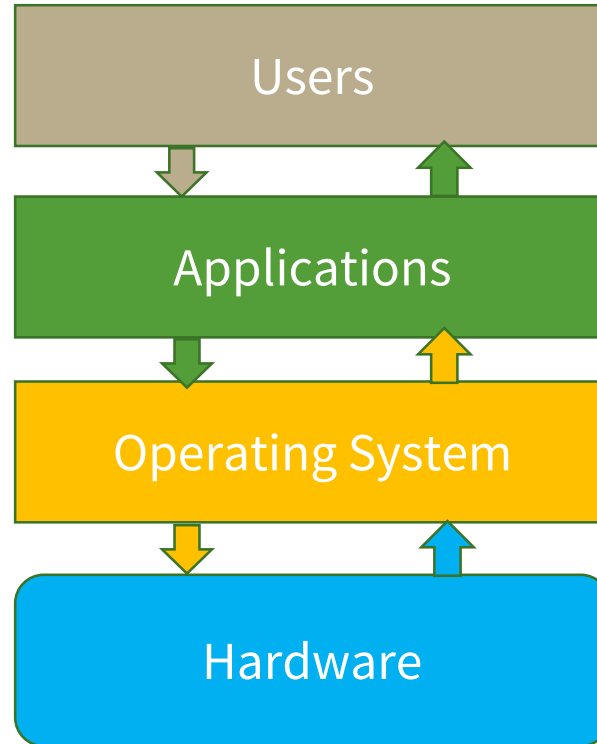
# Announcements

- **Exercise 3 is out, due on 6 May 2022 23:59**
  - Visit [https://ecs.wgtn.ac.nz/Courses/NWEN241\\_2022T1/Exercises](https://ecs.wgtn.ac.nz/Courses/NWEN241_2022T1/Exercises) for the handout
- **Assignment 3 is out, due on 9 May 2021 23:59**
  - Visit [https://ecs.wgtn.ac.nz/Courses/NWEN241\\_2022T1/Assignments](https://ecs.wgtn.ac.nz/Courses/NWEN241_2022T1/Assignments) for the handout

# Content

- System calls (Introduction here, details next week)
- Interprocess communication

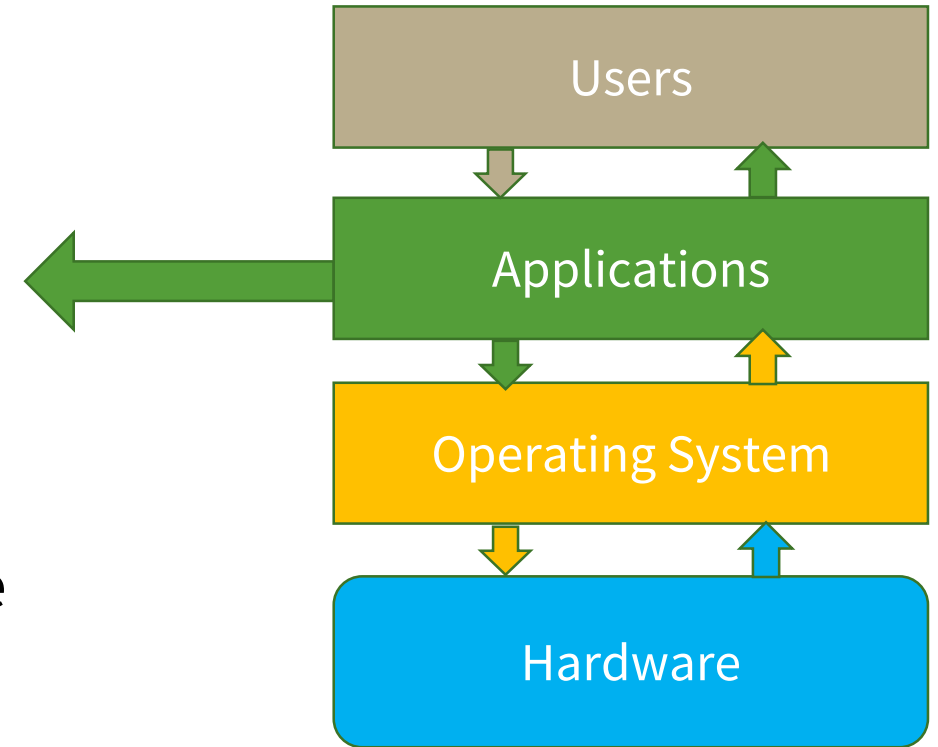
# System calls - What and Why?



**Conceptual View of a Computer System**

# System calls - What and Why?

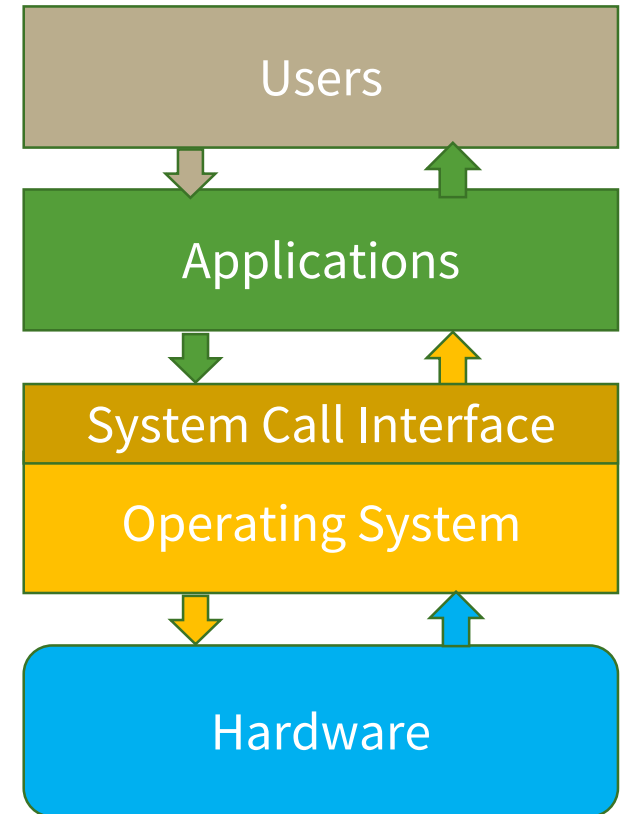
- Typically needs access to **system resources**.
- System resources can be:
  - a) **physical** – e.g. input devices, screen displays.
  - OR**
  - b) **Virtual** – e.g. files, network connections, threads.
- Applications need O.S. to enable them access these resources.



**Conceptual View of a Computer System**

# System calls - What and Why?

- Operating Systems **do not** allow application software to **access system resources directly** due to security and reliability issues.
- A program can **request** the services of system resources from O.S through **system calls**.
- **System calls** are **function invocations made from application into the OS** in order to request some service or resource from the operating system.
- Application developers often do not have direct access to system calls but can access them through a **system call API**, which in turn invokes the system call.



# An example of a system call usage

- Consider the following example:

```
#include<stdio.h>

int main()
{
    printf("Hello World");
    return 0;
}
```



C Library function **printf** "asks" the operating system to print for the calling program by using the system call API routines

# System call invocation – Example

```
#include <stdio.h>
void main(void)
{
    printf("Hello, world\n");
    exit(0);
}
```

printf("Hello, world\n");  
exit(0);

Standard C Library

write()

System Call Interface

User mode

Kernel mode

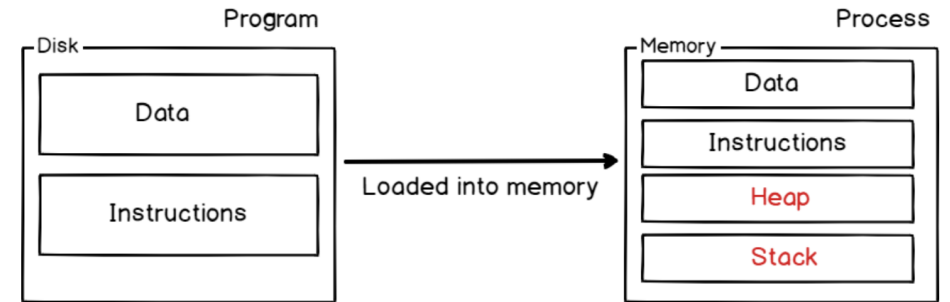
sys\_write()  
system call  
handler



# Interprocess Communication

# What is a process ?

- Program and process are related terms.



**Program** is a set of instructions to carry out a specified task

**Process** is a program in execution



Passive entity

Active entity

**Program** is stored in disk and does not require any other resource.

**Process** requires system resources such as CPU, memory, I/O etc.

Life span - Longer

Life span – limited

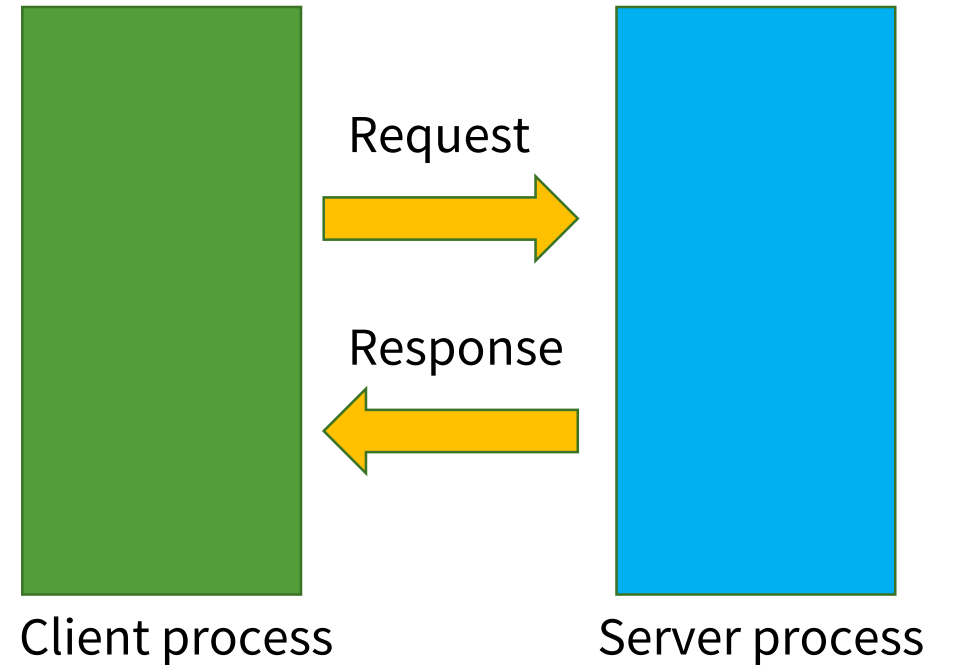
**Each time a program is run a new process is created.**

# Process - Independent Vs Cooperating

- **Independent** processes: processes that don't interact with other processes
- **Cooperating** processes: process can affect or be affected by other processes.
- In order to co-operate processes need to **communicate**
  - **Inter Process Communication**

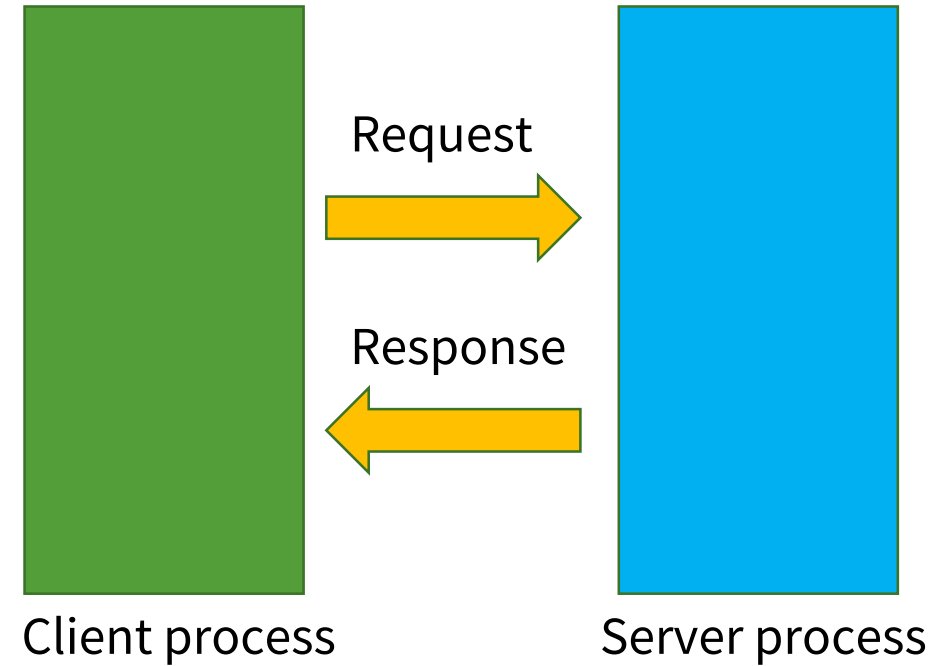
# Client-server model

- Most common IPC paradigm
- Based on the producer-consumer model of process cooperation
- Client makes the request for some resource or service to the server process
- Server process handles the request and sends the response (result) back to the client



# Client-server model

- **Client process** needs to know the existence and the address of the server
- However, the **Server** does not need to know the existence or address of the client prior to the connection
- **Once a connection** is established, both sides can send and receive information

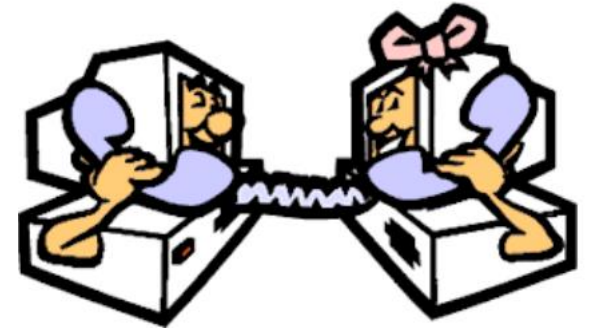


# Client-server communication

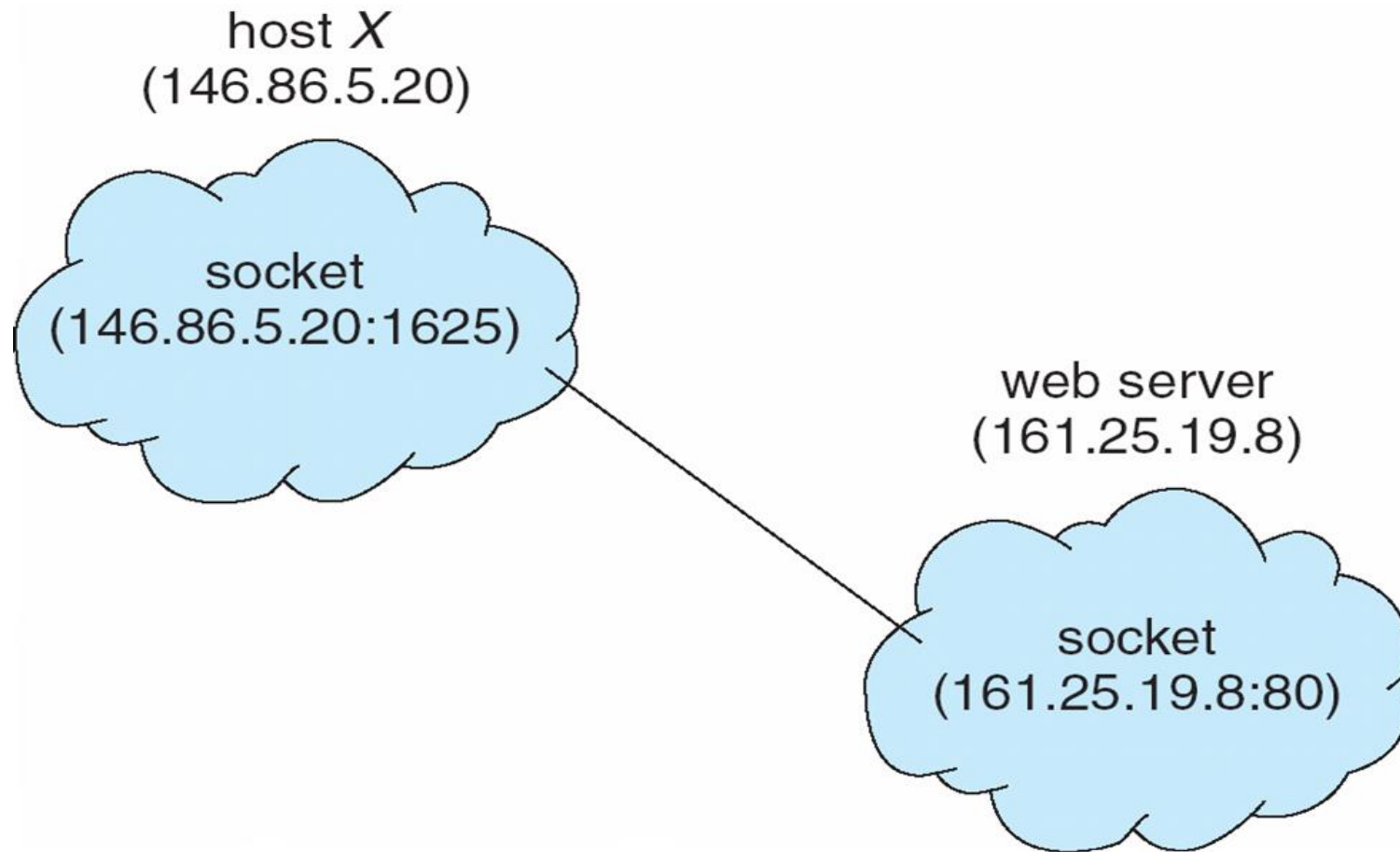
- Remote Procedure Calls
- Pipes
- **Sockets**

# What is socket?

- What do we need to know to allow two processes on a network to communicate?
  - **Identity of the communicating machines**
    - **IP Address**
  - **Identity of the communicating processes on these machines**
    - **Port**
- Concatenation of **IP address** and **port** defines a **socket** - A **socket** is defined as an endpoint for communication
- Example: The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**



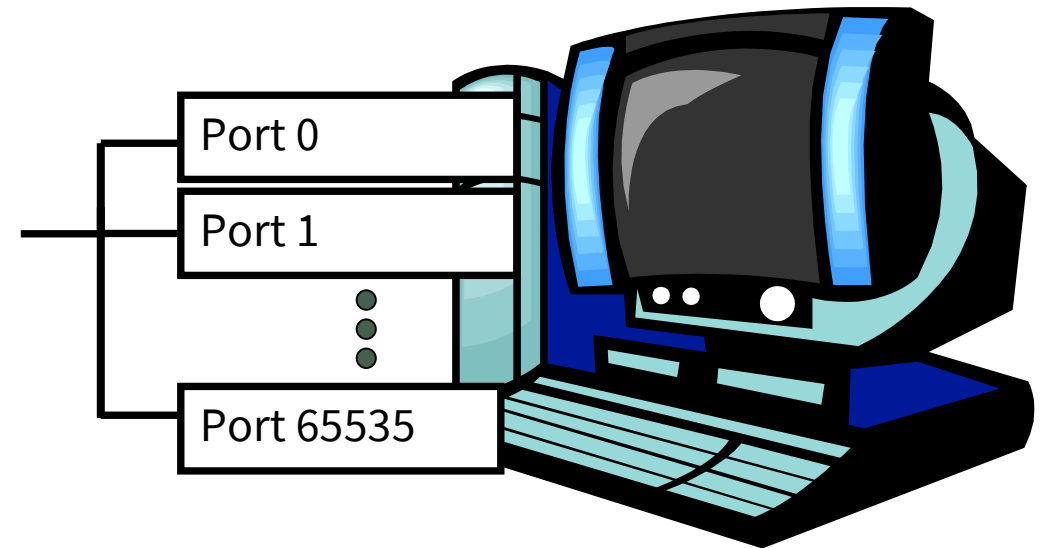
# Socket communication





# Port numbers

- Each host has **65,536** ports
- Use of ports 1-1023 requires privileges
- Some ports are reserved for specific apps
  - 20, 21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



# Sockets as programming interface

- An interface between application and network
  - The application creates a socket
  - The socket type dictates the style of communication
  - TCP VS UDP
    - reliable vs. best effort
    - connection-oriented vs. connectionless



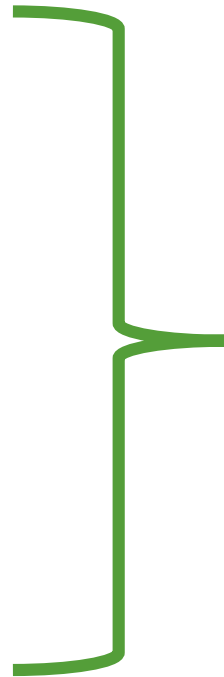
# Socket types

- SOCK\_STREAM
  - a.k.a. **TCP**
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional
- SOCK\_DGRAM
  - a.k.a. **UDP**
  - unreliable delivery
  - no order guarantees
  - no notion of “connection” – app indicates destination for each packet
  - can send or receive

We will focus on SOCK\_STREAM or TCP socket type

# System calls

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `send()` / `sendto()`
- `recv()` / `recvfrom()`



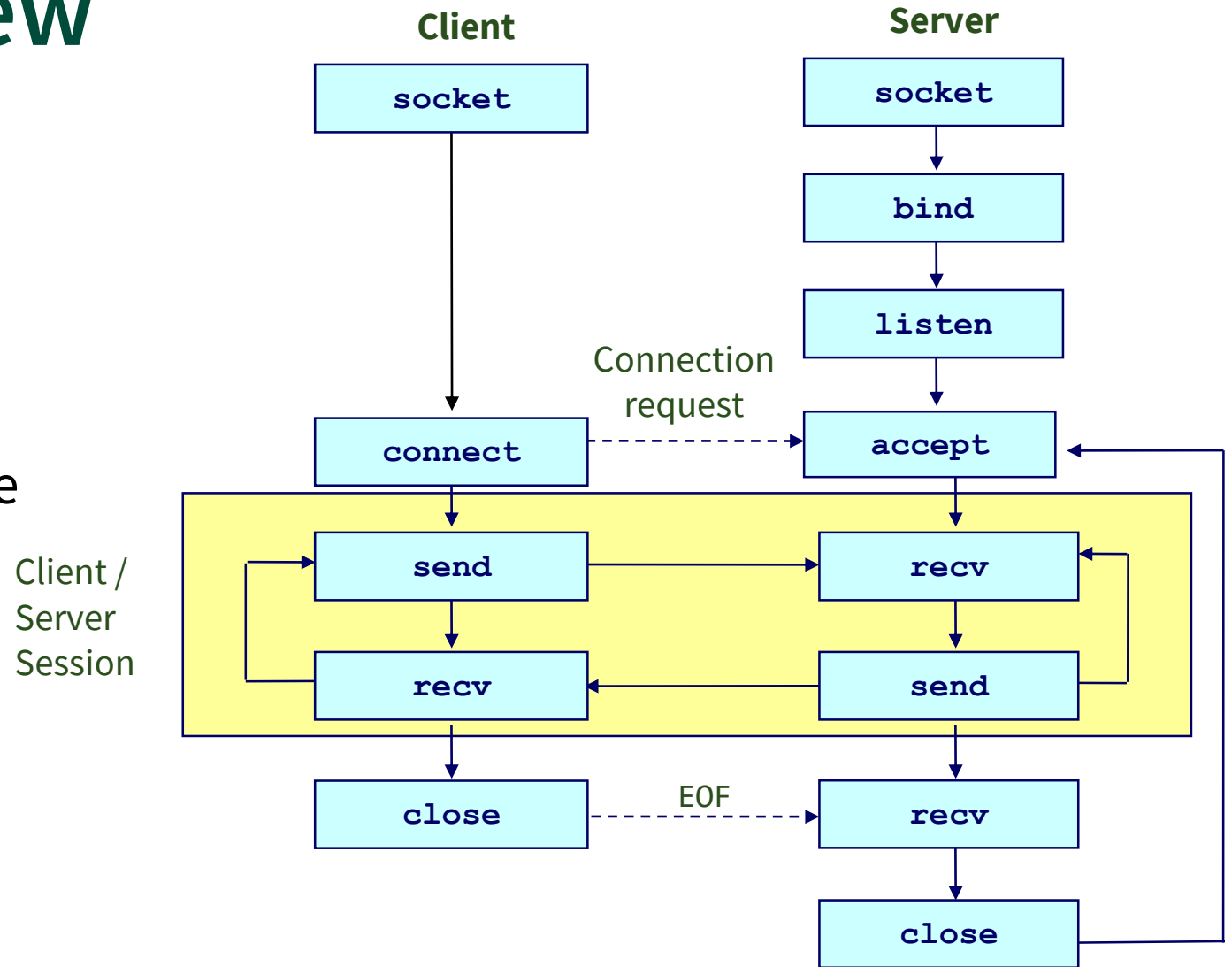
Include

`sys/types.h`

`sys/socket.h`

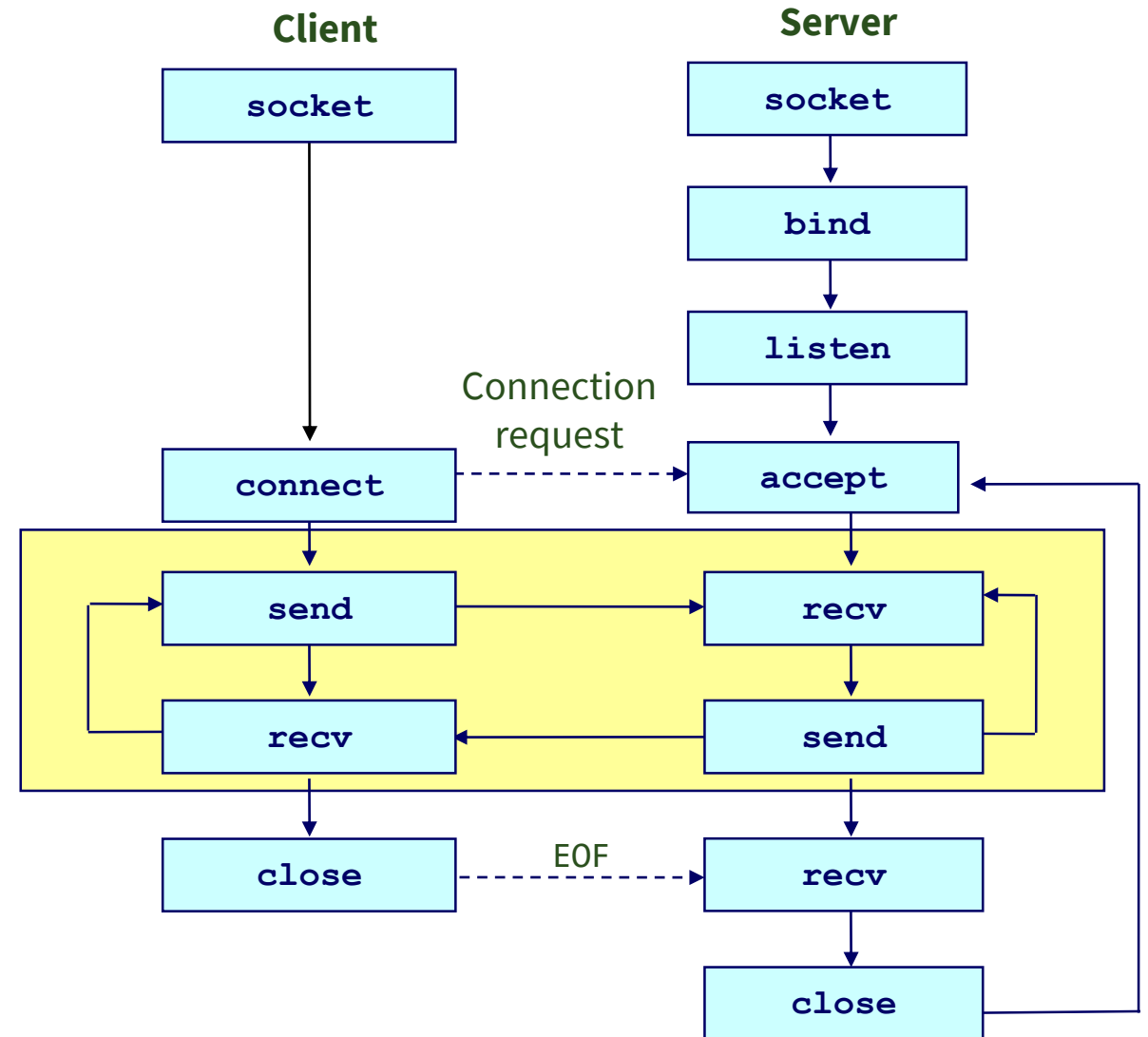
# TCP Client overview

- 1) Create a socket with the `socket()` system call
- 2) Connect the socket to the address of the server using the `connect()` system call
- 3) Send and receive data

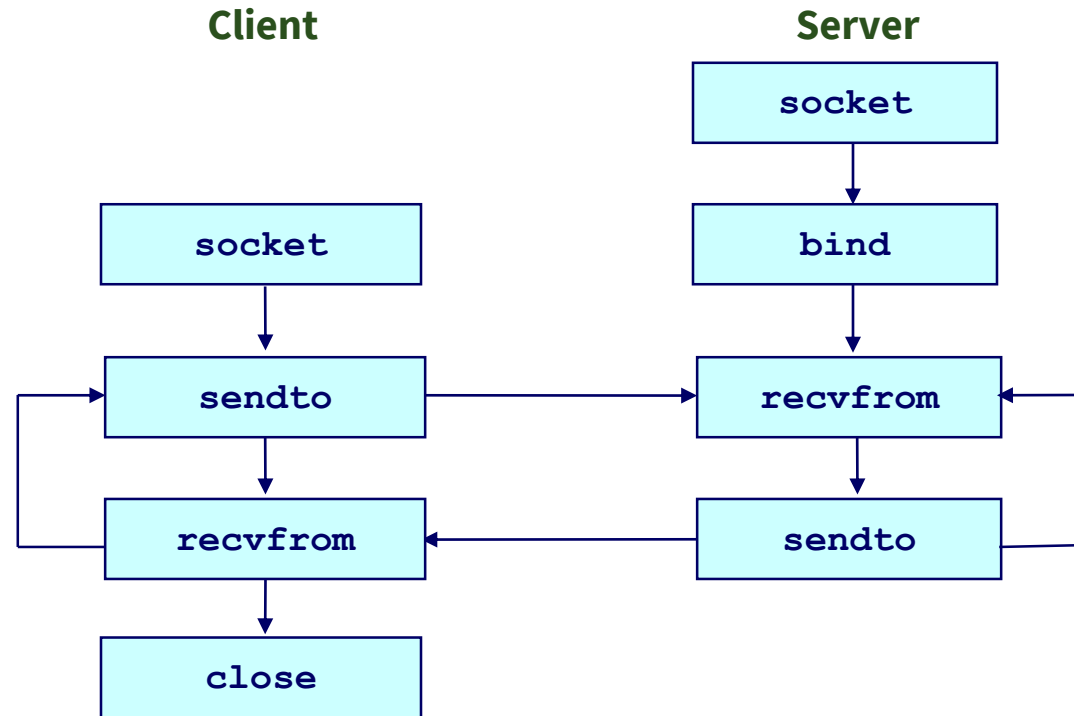


# TCP Server overview

- 1) Create a socket with the `socket()` system call
- 2) Bind the socket to an address using the `bind()` system call
- 3) Listen for connections with the `listen()` system call
- 4) Accept a connection with the `accept()` system call
- 5) Send and receive data



# Client-server communication overview - UDP



# Server: step 1

- Create a socket with the `socket()` system call

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

- *domain* – communication domain (protocol family) such as `AF_INET` (IPv4) or `AF_INET6` (IPv6)
  - *type* – communication semantics such as `SOCK_STREAM` (TCP) or `SOCK_DGRAM` (UDP)
  - *protocol* specifies the protocol. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0.
- Creates an endpoint of communication.
  - If successful, returns **socket file descriptor**, otherwise, returns -1



# Server: step 1 example

- Create TCP socket

```
int fd = socket(AF_INET, SOCK_STREAM, 0);  
if (fd == -1) {  
    printf("Error creating socket");  
    exit(0);  
}
```

- Create UDP socket

```
int fd = socket(AF_INET, SOCK_DGRAM, 0);  
if (fd == -1) {  
    printf("Error creating socket");  
    exit(0);  
}
```

# Server: step 2

- Bind the socket to an address using the `bind()` system call

```
int bind(int sockfd, const struct sockaddr *addr,  
         socklen_t addrLen);
```

- *sockfd* is the socket file descriptor (returned by `socket()`)
  - *addr* is a pointer to the structure `struct sockaddr` (generic data type for address) which contains the host IP address and port number to bind to
  - *addrLen* is the length of what *addr* points to
- 
- Binding means associating and reserving a port number for use by the socket
  - If successful, returns 0, otherwise, returns -1

# struct sockaddr

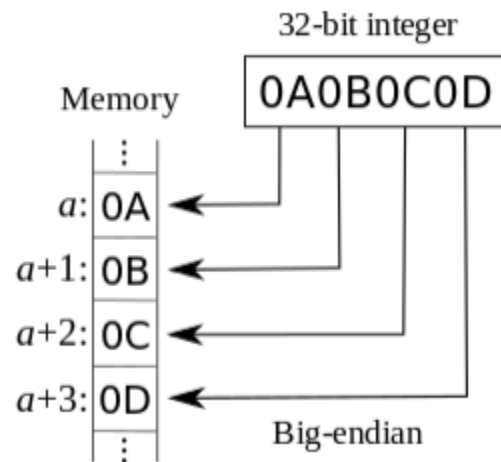
- `struct sockaddr_in` in IPv4 (included the `<netinet/in.h>` header)

```
struct sockaddr_in {
    short sin_family;           // AF_INET
    unsigned short sin_port;   // port number
    struct in_addr sin_addr;   // Internet address in
                               // network byte order
};

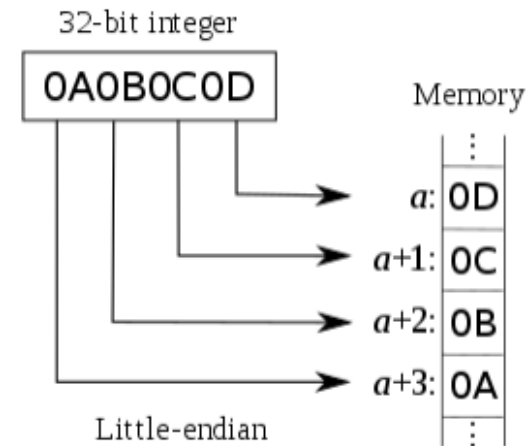
struct in_addr {
    unsigned long s_addr;     // IPv4 address in network
                               // byte order
};
```

# Host and network byte order

- Little-endian and big-endian issue?



A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest.



A little-endian system, in contrast, stores the least-significant byte at the smallest address.

# Host and network byte order

- Byte ordering also matters in network communication
  - Host and network may differ in byte ordering
  - Host byte order may be little-endian or big-endian
  - Network byte order is always big-endian
- Functions for converting between host and network byte order:

```
uint32_t htonl(uint32_t hostlong);  
uint16_t htons(uint16_t hostshort);  
uint32_t ntohl(uint32_t netlong);  
uint16_t ntohs(uint16_t netshort);
```

# Server: step 2 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
...

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; // any address
addr.sin_port = htons(1234); // port 1234

if (bind(fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
    printf("Error binding socket");
    exit(0);
}
```

```
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
};

struct in_addr {
    unsigned long s_addr;
};
```

# Server: step 3

- Listen for connections with the `listen()` system call

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

- *sockfd* is the socket file descriptor (returned by `socket()`)
- *backLog* is the maximum number of pending connections to allow for this socket
  - `SOMAXCONN` is defined as the number of maximum pending connections allowed by the operating system
- If successful, returns 0, otherwise, returns -1

# Server: step 3 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);  
...  
  
if(listen(fd, SOMAXCONN) < 0) {  
    printf("Error listening for connections");  
    exit(0);  
}
```



# Server: step 4

- Accept a connection with the `accept()` system call

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

- *sockfd* is the socket file descriptor (returned by `socket()`)
- *addr* is a pointer to the structure `struct sockaddr` which will contain the details of the peer socket
- *addrLen* is a pointer to the length of what *addr* points to
- If successful, returns non-negative **socket file descriptor**, otherwise, returns -1

# Server: step 4 example

```
int fd = socket(AF_INET, SOCK_STREAM, 0);  
...  
struct sockaddr_in client_addr;  
int addrlen = sizeof(client_addr);  
  
int client_fd = accept(fd, (struct sockaddr *)&client_addr,  
                      (socklen_t*)&addrlen);  
  
if(client_fd < 0) {  
    printf("Error accepting connection");  
    exit(0);  
}
```

# Server: step 5

- **Send** and receive data

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

- *sockfd* is the socket file descriptor (returned by `accept()`)
  - *buf* is a pointer to buffer to be sent
  - *len* is the length of buffer to be sent
  - *flags* is bitwise OR of zero or more options
- 
- Used in connection-oriented sockets (TCP)
  - If successful, returns number of characters sent, otherwise, returns -1
  - `send(sockfd, buf, len, 0);` is equivalent to `write(sockfd, buf, len);`

# Server: step 5

- **Send** and receive data

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,  
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

- *sockfd* is the socket file descriptor (returned by socket())
  - *buf* is a pointer to buffer to be sent
  - *len* is the length of buffer to be sent
  - *flags* is bitwise OR of zero or more options
  - *dest\_addr* is a pointer to the structure struct sockaddr which will contain the details of the peer socket
  - *addrlen* is a pointer to the length of what dest\_addr points to
- Used in non-connection-oriented sockets (UDP)
  - If successful, returns number of characters sent, otherwise, returns -1

# Server: step 5 example using send()

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
...
int client_fd = accept(fd, (struct sockaddr *)& client_addr,
                      (socklen_t *)& addrlen);
...

char msg[] = "hello, world";
ssize_t r = send(client_fd, msg, strlen(msg), 0);
if(r < 0) {
    printf("Error sending message");
    close(client_fd);
    exit(0);
}
```

# Server: step 5

- Send and **receive** data

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

- *sockfd* is the socket file descriptor (returned by `accept()`)
  - *buf* is a pointer to buffer to be received
  - *len* is the length of buffer to be received
  - *flags* is bitwise OR of zero or more options
- 
- Used in connection-oriented sockets (TCP)
  - If successful, returns number of characters received, otherwise, returns -1
  - If peer socket is shutdown/closed, will return 0
  - `recv(sockfd, buf, len, 0);` is equivalent to `read(sockfd, buf, len);`

# Server: step 5

- Send and **receive** data

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,  
                 struct sockaddr *src_addr, socklen_t *addrLen);
```

- *sockfd* is the socket file descriptor (returned by socket())
- *buf* is a pointer to buffer to be received
- *len* is the length of buffer to be received
- *flags* is bitwise OR of zero or more options
- *src\_addr* is a pointer to the structure struct sockaddr which will contain the details of the peer socket
- *addrLen* is a pointer to the length of what src\_addr points to
- Used in non-connection-oriented sockets (UDP)
- If successful, returns number of characters received, otherwise, returns -1
- If peer socket is shutdown/closed, will return 0

# Server: step 5 example using recv()

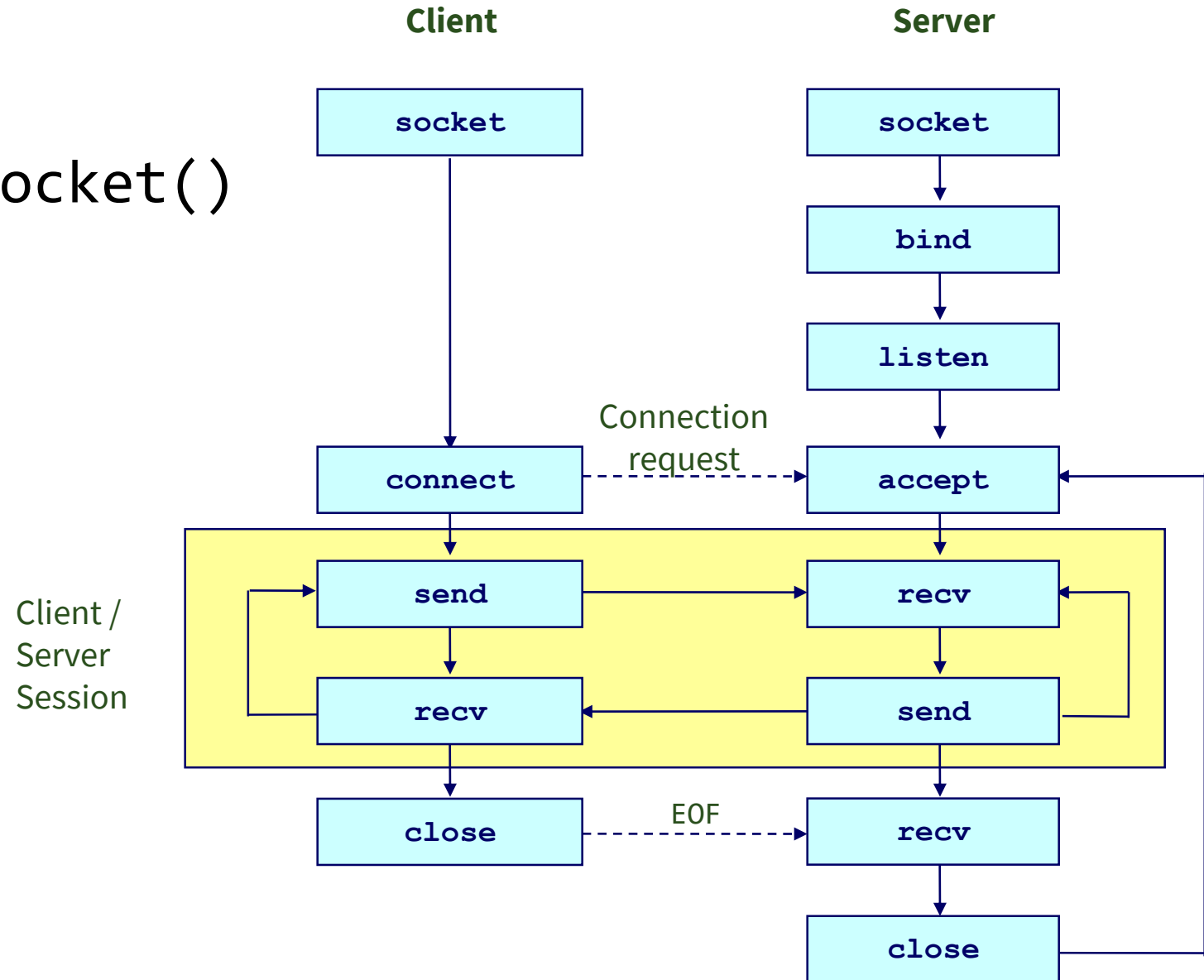
```
int fd = socket(AF_INET, SOCK_STREAM, 0);
...
int client_fd = accept(fd, (struct sockaddr *)& client_addr,
                      (socklen_t*)&addrlen);
...

char incoming[100];
ssize_t r = recv(client_fd, incoming, 100, 0);
if(r <= 0) {
    printf("Error receiving message");
    close(client_fd);
    exit(0);
}
// Do something with receiving message
printf("Received message: %s", incoming);
```



# Client: step 1

- Create a socket with the `socket()` system call
- Same as server step 1



# Client: step 2

- Connect the socket to the address of the server using the `connect()` system call
  - This step is only required for connection-oriented sockets (TCP)

```
int connect(int sockfd, const struct sockaddr *addr,  
socklen_t addrLen);
```

- *sockfd* is the socket file descriptor (returned by `socket()`)
- *addr* is a pointer to the structure `struct sockaddr` which will contain the details of the server socket
- *addrLen* is a pointer to the length of what *addr* points to
- If successful, returns 0, otherwise, returns -1

# Client: step 3

- Send and receive data
- Same as server step 5

# Closing a socket

- Socket must be closed after its use

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

```
int close(int sockfd);
```

- *sockfd* is the socket file descriptor (returned by `socket()`)
- *how* can either be `SHUT_RD` (further receptions disallowed), `SHUT_WR` (further transmissions disallowed), or `SHUT_RDWR` (further receptions and transmissions disallowed)
- Shutdown blocks communication without destroying the socket, close blocks the communication and destroys the socket.
- If successful, returns 0, otherwise, returns -1

# Next Lecture

- **This week** (Tutorial Style Lecture): Interprocess Communication (TCP)
- **Next Week** : System Calls and Process Management