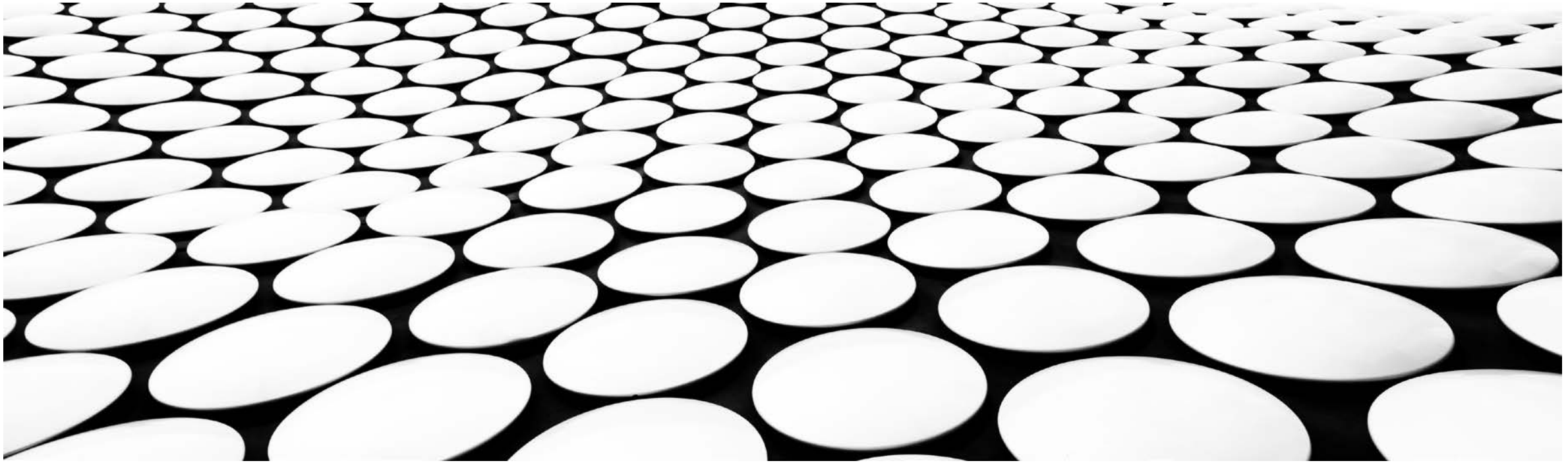
SOCKET PROGRAMMING



INTERPROCESS COMMUNICATION

Within a Single System:

These IPC mechanisms are used when processes communicate within the same operating system.

Pipes & FIFOs

- Pipes: A unidirectional communication channel between related processes (e.g., parent and child processes). One process writes data, and the other reads it.
- FIFOs (Named Pipes): Similar to pipes but with a name in the filesystem, allowing communication between unrelated processes.

Message Queues

- A structured way to send and receive messages between processes.
- The OS manages the queue, allowing multiple processes to read from and write to it asynchronously.
- Provides more flexibility than pipes since messages can be prioritized.

Semaphores & Shared Memory

- Semaphores: Used for synchronization between processes, ensuring controlled access to shared resources.
- Shared Memory: The fastest IPC method. Processes access a common memory region, eliminating the need for data copying. However, synchronization mechanisms (such as semaphores) are required to prevent conflicts.

Across Different Systems:

When processes communicate over a network, these mechanisms are used:

BSD Sockets

- A low-level networking API that allows communication between processes running on different machines.
- Supports both TCP (reliable, connection-based communication) and UDP (faster, connectionless communication).
- Used for client-server applications, such as web services.

Transport Layer Interface (TLI)

- An alternative to BSD sockets, used in older Unix systems.
- Provides a higher-level API for network communication, but is less commonly used today.

WHAT IS A SOCKET?

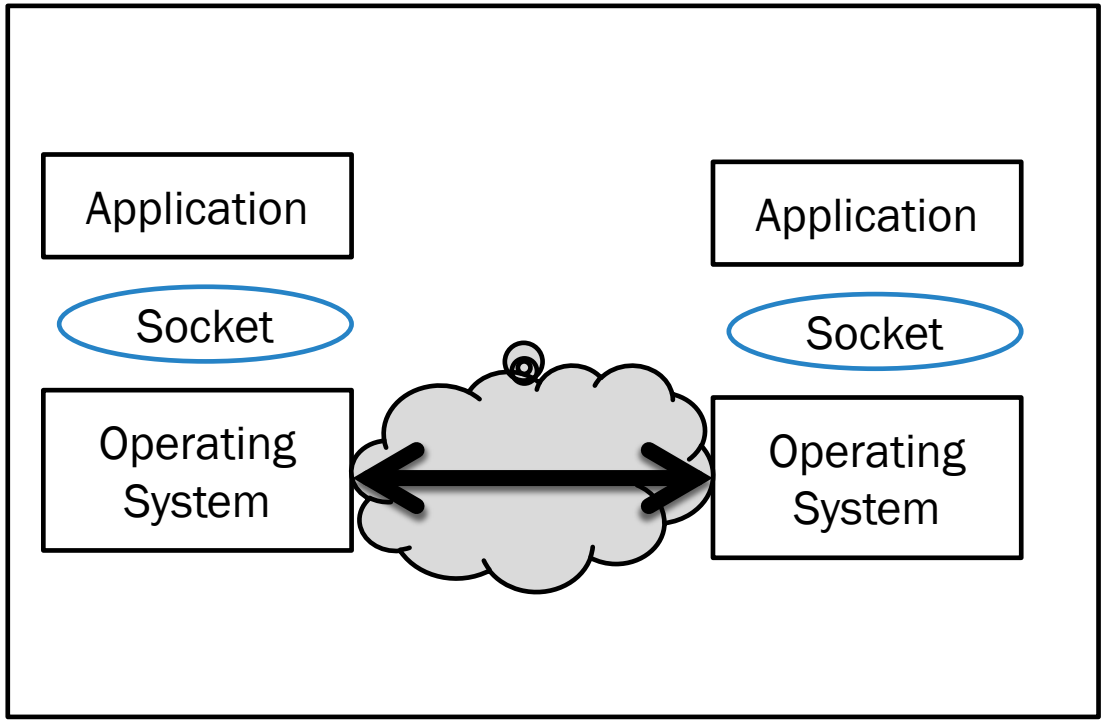
A **socket** is an **endpoint** for communication between two machines. It is a key concept in **network programming**, allowing processes to send and receive data over a network.

- History and Background

1. Introduced in Berkeley UNIX in the 1980s, later adopted widely.
2. Used for communication in client-server models.
3. In UNIX, everything is treated as a file, including network communication.

- How Sockets Work?

1. Process creates a socket using the socket() function.
2. The socket is bound to an address (IP + Port).
3. Clients and servers communicate through send() / recv() functions.
4. When communication ends, the socket is closed



WHAT IS A SOCKET?

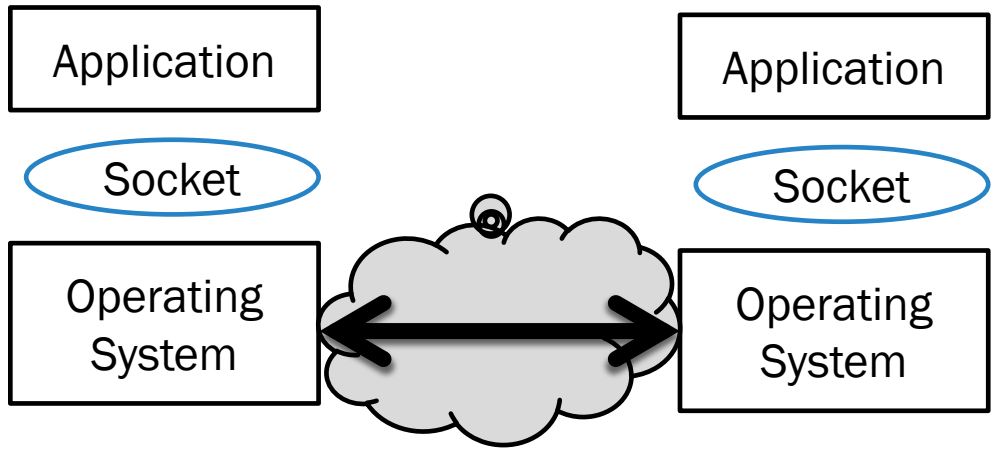
A **socket** is an **endpoint** for communication between two machines. It is a key concept in **network programming**, allowing processes to send and receive data over a network.

- Key Characteristics

1. Bidirectional communication: Data can flow both ways.
2. Abstracts network details: Applications do not need to manage low-level packet transmission.
3. Supports multiple protocols: TCP (reliable) and UDP (fast but unreliable).

- Practical Usage

1. Web browsing (HTTP, HTTPS)
2. Email (SMTP, IMAP, POP3)
3. File transfers (FTP, SCP)
4. Online gaming, VoIP, and more



CONNECTION-ORIENTED APPLICATION

A **connection-oriented application** establishes a reliable communication channel before exchanging data. **TCP (Transmission Control Protocol)** is commonly used for such applications because it ensures data is **delivered correctly and in order.**

Server: Preparing to Handle Clients

Before a server can communicate with clients, it must go through the following steps:

1. **Create a socket**
   - The server calls socket(AF_INET, SOCK_STREAM, 0) to create a TCP socket.
   - AF_INET: Specifies IPv4 addressing.
   - SOCK_STREAM: Indicates a stream-oriented (TCP) connection.
2. **Bind an address (IP + port) to the socket**
   - The bind() function assigns a specific IP address and port number to the socket.
   - This allows clients to locate the server.
3. **Call listen() to mark it as a passive socket**
   - listen() tells the OS that this socket will be used to accept incoming connections.
   - The backlog parameter determines how many pending connections can be queued.
4. **Accept incoming connections from clients (accept())**
   - The server waits for a connection request.
   - When a client connects, accept() returns a new socket specifically for communication with that client.

CONNECTION-ORIENTED APPLICATION

A **connection-oriented application** establishes a reliable communication channel before exchanging data. **TCP (Transmission Control Protocol)** is commonly used for such applications because it ensures data is **delivered correctly and in order**.

🔷 Client: Connecting to the Server

To initiate communication, the client follows these steps:

1. **Create a socket**
   - Just like the server, the client creates a TCP socket using socket(AF_INET, SOCK_STREAM, 0).
2. **Connect to the server**
   - The client calls connect(), specifying the server's IP address and port number.
   - If the server accepts the connection, a communication channel is established.
3. **Exchange data (send() and recv())**
   - Once connected, the client can send data to the server using send().
   - The server responds, and the client reads the data using recv().

🔷 Further Communication

Once the connection is established, further communication depends on the application protocol.

For example:

- Web applications use HTTP over TCP.
- File transfers use FTP over TCP.
- Chat applications establish a continuous TCP session.

WHY TCP FOR CONNECT 4?

- **Reliable & Ordered**: We cannot afford to lose or reorder moves.
- **Stream-based**: No need to handle packet boundaries manually.
- **Easier** for a turn-based game: No re-transmission logic required in user space.

Feature	TCP	UDP
Reliability	Ensures all moves are received	Packets can be lost
Ordering	Moves arrive in order	Moves may arrive out of sequence
Data Handling	Continuous stream, no manual reassembly	Must handle packet boundaries manually

TYPICAL CLIENT\ SERVER PROGRAM

Client

- Create a socket.
- Determine server address and port number.
- Initiate the connection to the server.
- Write data to the socket.
- Read data from the socket.
- Close the socket.

Server

- Create a socket.
- Associate local address and port with the socket.
- Wait to hear from a client (passive).
- Accept an incoming connection from a client.
- Write data to the socket.
- Read data from the socket.
- Close the socket.
- Repeat with the next connection request.

CREATE SOCKET

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

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

In socket programming, the **socket()** function is used to **create a socket** for communication.

- Returns: A file descriptor (integer) for the newly created socket.
- On failure: Returns -1 and sets errno to indicate the error.

Parameters Explanation

1. Domain (Protocol Family)

Specifies the communication domain (or protocol family).
Common values:

Domain	Description
AF_INET	IPv4 Internet protocols
AF_INET6	IPv6 Internet protocols
AF_UNIX	Local interprocess communication (IPC)
AF_IPX	Novell IPX (rarely used)

Most common choice: AF_INET for IPv4 networking.

2. Type (Communication Semantics)

Defines the type of communication the socket will use:

Type	Description
SOCK_STREAM	Reliable, connection-oriented (TCP)
SOCK_DGRAM	Fast, connectionless (UDP)
SOCK_SEQPACKET	Fixed-length, ordered message exchange

Most common choices:

SOCK_STREAM for **TCP** (e.g., web services, file transfer).

SOCK_DGRAM for **UDP** (e.g., video streaming, DNS).

CREATE SOCKET

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

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

Parameters Explanation

3. **Protocol**

Defines the **specific protocol** to use within the selected type.

Commonly set to 0, meaning the system selects the default protocol for the given type:

SOCK_STREAM → **TCP**

SOCK_DGRAM → **UDP**

If multiple protocols exist, an explicit protocol number can be provided.

Example: Creating a TCP Socket

```c
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
    perror("Socket creation failed");
    exit(EXIT_FAILURE);
}
```

Creates a TCP socket using IPv4 (AF_INET).

Returns a file descriptor (sockfd) if successful.

Returns -1 if an error occurs, with details stored in errno.

STREAMS AND DATAGRAMS

In socket programming, communication can be categorized into **two types**:

1.Connection-oriented reliable byte streams (TCP - SOCK_STREAM)

2.Connectionless unreliable datagrams (UDP - SOCK_DGRAM)

◆ Connection-Oriented Reliable Byte Stream (TCP - SOCK_STREAM)

- Uses TCP, which ensures reliable, ordered, and error-free data transfer.
- No message boundaries:
  - Data is treated as a continuous stream.
  - The receiver may receive multiple writes() in a single read() call.
- Suitable for:
  - Web browsing (HTTP, HTTPS)
  - File transfers (FTP)
  - Email services (SMTP, IMAP, POP3)

◆ Connectionless Unreliable Datagram (UDP - SOCK_DGRAM)

- Uses UDP, which is faster but unreliable (no error checking, no retransmission).
- Message boundaries are preserved:
  - Each sendto() corresponds exactly to one recvfrom().
  - Data is received as distinct packets.
- Suitable for:
  - Video streaming, VoIP, gaming (low latency)
  - DNS lookups (quick, lightweight requests)
  - IoT devices with minimal overhead

CONNECTING TO THE SERVER

In socket programming, the connect() function is used by a client to establish a connection with a remote server.

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

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

- ◆ Parameters Explanation
- 1. sockfd (Socket Descriptor)
  - A file descriptor returned by the socket() function.
  - Identifies the client's socket.
- 2. addr (Server Address)
  - A pointer to a struct sockaddr that specifies the server's IP address and port.
  - Typically cast from struct sockaddr_in for IPv4:

struct sockaddr_in serv_addr;

serv_addr.sin_family = AF_INET; // IPv4

serv_addr.sin_port = htons(8080); // Port 8080

serv_addr.sin_addr.s_addr = inet_addr("192.168.1.1"); // Server IP

- 3. addrlen (Address Size)
  - Specifies the size of the addr structure, typically sizeof(struct sockaddr_in).

USEFULL STRUCTS

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

Purpose:

A generic structure used by system calls like bind(), connect(), accept(), etc.
Holds any type of socket address (IPv4, IPv6, etc.).

Limitations:

sa_data is not structured, so it is not directly used in most cases.
Instead, more specific structures like sockaddr_in are used.

USEFULL STRUCTS

```c
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
};
```

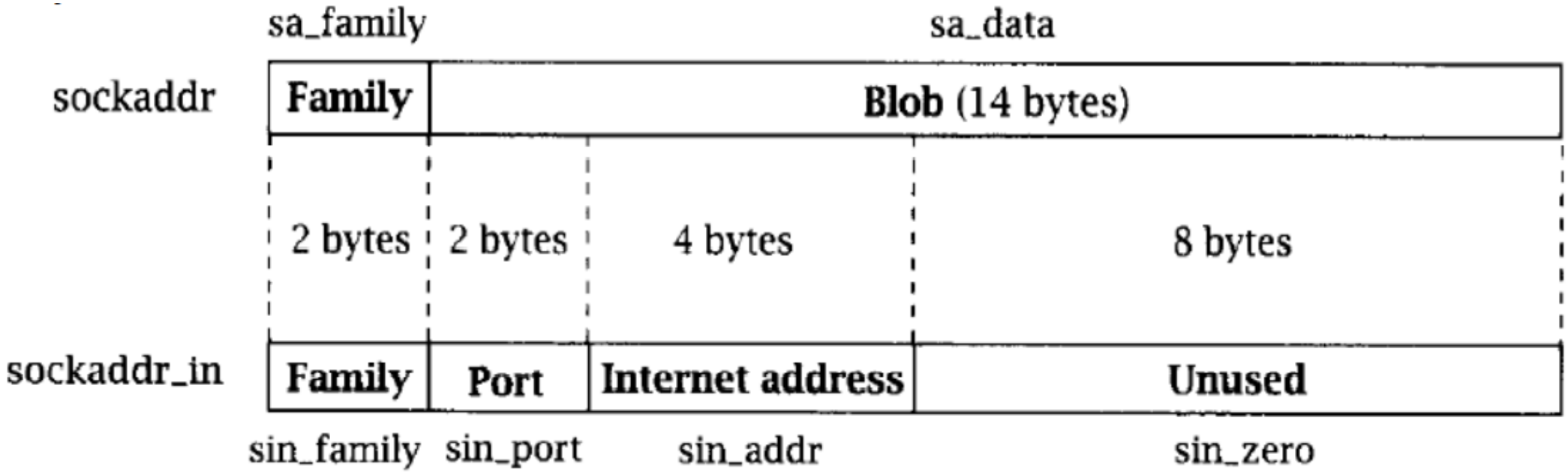
Purpose:

Specifically designed for IPv4 addresses.
Used when working with TCP/IP or UDP/IP networking.

Key Fields:

Field	Description
sin_family	Set to AF_INET for IPv4 addresses
sin_port	Holds the port number (must use htons())
sin_addr	Stores the IPv4 address (use inet_addr() or inet_pton())
sin_zero	Padding to keep structure size consistent

USEFULL STRUCTS (2)



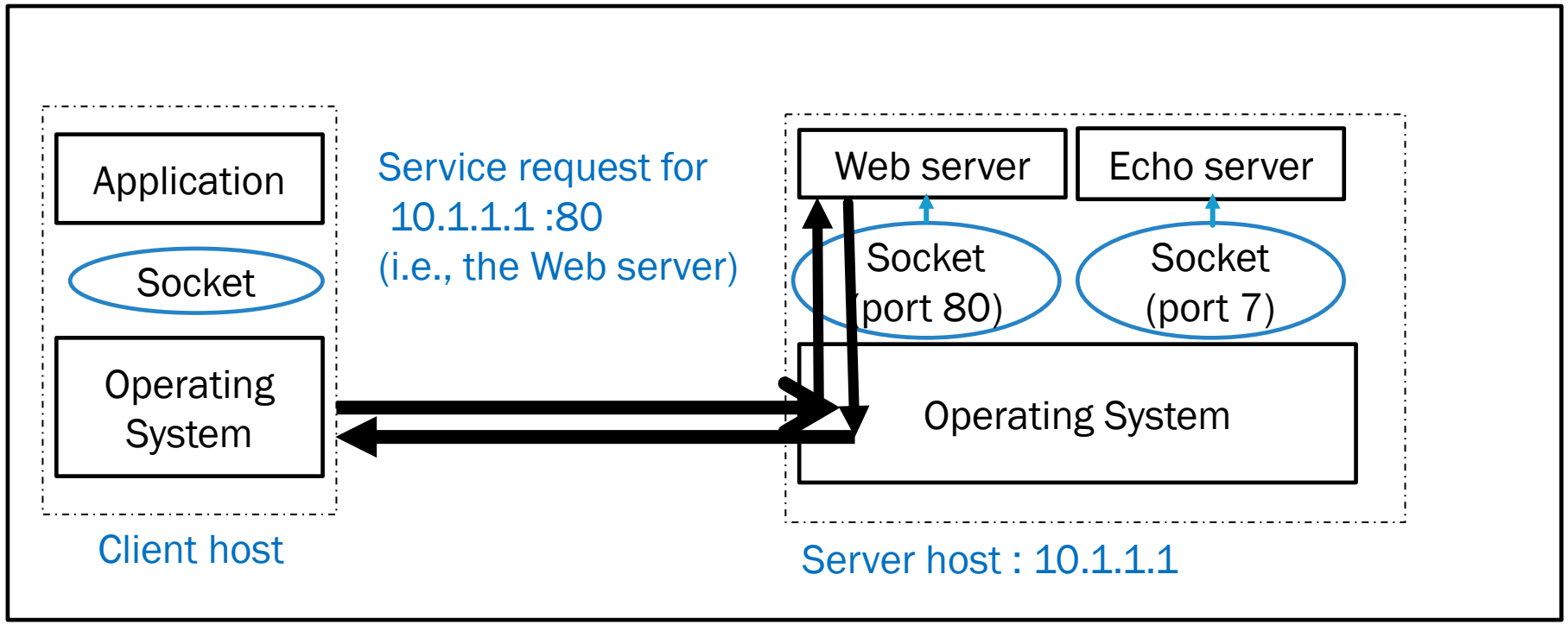
SENDING AND RECEIVING DATA

- write(int sockfd, void *buf, size_t len).
  - Parameters:
    - **sockfd**: The socket descriptor.
    - **buf**: A pointer to the buffer containing data to send.
    - **len**: The size of the buffer in bytes..
  - Returns the number of characters written, and -1 on error.

- read(int sockfd, void *buf, size_t len)
  - Parameters:
    - **sockfd**: The socket descriptor.
    - **buf**: A pointer to a buffer to store received data.
    - **len**: The maximum number of bytes to read..
  - Returns the number of characters read, and -1 on error.

- int close(int sockfd)
  - Closes the socket and releases resources.
  - No further read/write operations can be performed after closing the socket.
  - Returns 0 on success, -1 on failure.

TYPICAL CLIENT PROGRAM



TYPICAL CLIENT\ SERVER PROGRAM

Client

- ✓ Create a socket.
- ✓ Determine server address and port number.
- ✓ Initiate the connection to the server.
- ✓ Write data to the socket.
- ✓ Read data from the socket.
- ✓ Close the socket.

Server

- Create a socket.
- Associate local address and port with the socket.
- Wait to hear from a client (passive).
- Accept an incoming connection from a client.
- Write data to the socket.
- Read data from the socket.
- Close the socket.
- Repeat with the next connection request.

BIND SOCKET TO THE LOCAL ADDRESS AND PORT

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

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

- Returns 0 on success ⟶ Binding is successful.
- Returns -1 on failure ⟶ Binding failed.

Parameters Explanation

1. sockfd (Socket Descriptor)

- A file descriptor returned by socket().
- Identifies the socket that will be bound to an address.

2. addr (Socket Address)

- A pointer to a sockaddr structure that contains the IP address and port.
- Typically cast from struct sockaddr_in for IPv4:

struct sockaddr_in server_addr;

server_addr.sin_family = AF_INET;

server_addr.sin_port = htons(8080);

server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

- AF_INET ⟶ IPv4 protocol.
- htons(8080) ⟶ Converts port 8080 to network byte order.
- htonl(INADDR_ANY) ⟶ Allows the socket to accept connections on any available network interface.

3. addrlen (Address Length)

The size of the address structure (e.g., sizeof(struct sockaddr_in)).

LISTEN FOR CONNECTIONS ON A SOCKET

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

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

- Returns 0 on success ⟶ The socket is ready to accept connections.
- Returns -1 on failure ⟶ Check errno for details.

Parameters Explanation

1. sockfd (Socket Descriptor)

- The socket file descriptor returned by socket().
- Must be bound to an address using bind() before calling listen().

2. backlog (Connection Queue Limit)

- Defines the maximum number of pending connections that can be queued before being accepted.
- If a new connection request arrives when the queue is full, the client may receive an error (ECONNREFUSED).

CONNECTION ACCEPTANCE BY SERVER

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

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

- Returns a new socket descriptor (client_sockfd) ⟶ Used for communication with the client.
- Returns -1 on failure ⟶ Check errno for details.

Parameters Explanation

1. sockfd (Listening Socket)

- The server's listening socket descriptor created with socket() and bind(), then marked as passive using listen().
- Must be in listening mode (listen() called first).

2. addr (Client Address)

- A pointer to a sockaddr structure that will store the client's IP address and port number.
- Typically cast from struct sockaddr_in for IPv4:

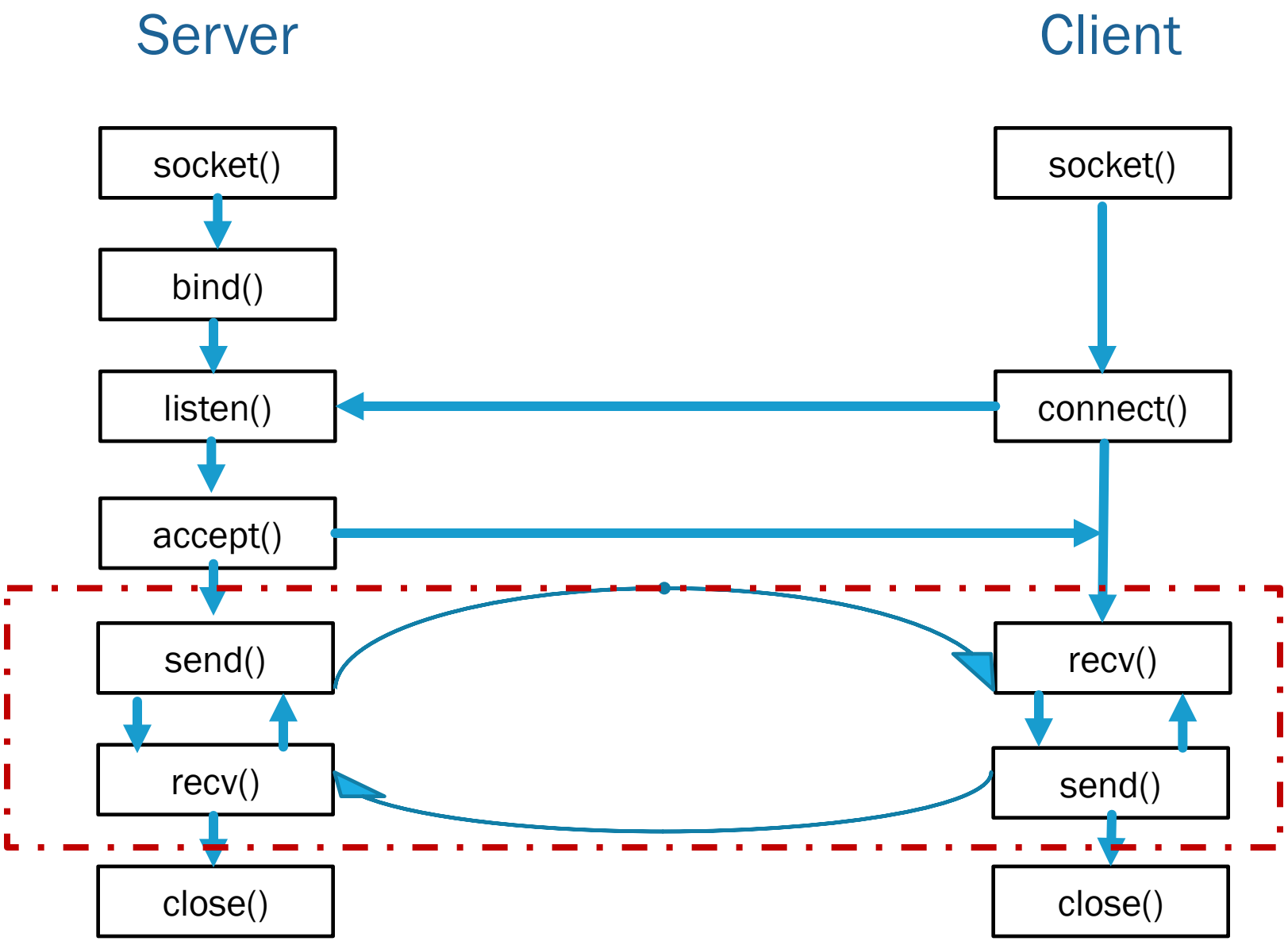
struct sockaddr_in client_addr;

socklen_t addr_size = sizeof(client_addr);

- If NULL, the server won't retrieve client address details.

3. addrlen (Address Size)

- A pointer to a variable holding the size of the addr structure.
- Must be initialized before calling accept().


Server
Client
socket()
bind()
listen()
accept()
send()
recv()
close()
socket()
connect()
recv()
send()
close()

```c
// Main function
int main() {
    struct sockaddr_in servaddr, clientaddr; // Server and client address structures
    char buff[1500]; // Buffer for storing received data
    int client_len;  // Length of client address structure

    // Step 1: Create a UDP socket
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1) {// Check if socket creation failed
        perror("Socket creation failed...\n");
        exit(EXIT_FAILURE);
    }

    // Step 2: Clear the server address structure
    bzero(&servaddr, sizeof(servaddr));

    // Step 3: Assign IP and PORT
    servaddr.sin_family = AF_INET;          // IPv4 address family
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY); // Accept connections from any IP
    servaddr.sin_port = htons(9090);        // Port number 9090 (convert to network byte order)

    // Step 4: Bind the socket to the IP and Port
    if (bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
        perror("Socket bind failed...\n");
        exit(EXIT_FAILURE);
    }

    // Step 5: Define client address length
    int clientLen = sizeof(clientaddr);

    // Step 6: Server listens indefinitely for incoming data
    while (1) {
        bzero(buff, 1500); // Clear buffer before receiving new data

        // Step 7: Receive data from client
        recvfrom(sockfd, buff, 1500 - 1, 0, (struct sockaddr *)&clientaddr, &client_len);

        // Step 8: Print the received message
        printf("From client: %s\t To client: ", buff);
        printf("%s \n ", buff);
    }

    // Step 9: Close the socket (this will never execute because of the infinite loop)
    close(sockfd);

    return 0;
}
```

UDP server

```c
// Main function
int main() {
    struct sockaddr_in servaddr; // Server address structure
    char buff[1500]; // Buffer for storing data
    int sockfd;

    // Step 1: Create a UDP socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd == -1) {
        perror("Socket creation failed...\n");
        exit(EXIT_FAILURE);
    }

    // Step 2: Define server address
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(9090); // Server port
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server IP address

    // Step 3: Send a message to the server
    strcpy(buff, "Hello, Server!");
    sendto(sockfd, buff, strlen(buff), 0, (struct sockaddr*)&servaddr, sizeof(servaddr));

    // Step 4: Receive response from the server
    int len = sizeof(servaddr);
    recvfrom(sockfd, buff, sizeof(buff), 0, (struct sockaddr*)&servaddr, &len);
    printf("From server: %s\n", buff);

    // Step 5: Close the socket
    close(sockfd);

    return 0;
}
```

UDP client

```
/ Main function
int main() {
    struct sockaddr_in servaddr, clientaddr; // Server and client address structures
    char buff[1500]; // Buffer for storing received data
    int sockfd, client_sockfd, client_len;

    // Step 1: Create a TCP socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("Socket creation failed...\n");
        exit(EXIT_FAILURE);
    }

    // Step 2: Define server address
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(9090); // Port number
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY); // Accept connections from any IP

    // Step 3: Bind the socket
    if (bind(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0) {
        perror("Socket bind failed...\n");
        exit(EXIT_FAILURE);
    }

    // Step 4: Listen for incoming connections
    if (listen(sockfd, 5) < 0) {
        perror("Listen failed...\n");
        exit(EXIT_FAILURE);
    }
    printf("Server listening on port 9090...\n");

    // Step 5: Accept a client connection
    client_len = sizeof(clientaddr);
    client_sockfd = accept(sockfd, (struct sockaddr*)&clientaddr, &client_len);
    if (client_sockfd < 0) {
        perror("Client connection failed...\n");
        exit(EXIT_FAILURE);
    }
    printf("Client connected!\n");

    // Step 6: Receive data from the client
    bzero(buff, 1500);
    read(client_sockfd, buff, sizeof(buff));
    printf("From client: %s\n", buff);

    // Step 7: Send response to client
    write(client_sockfd, buff, strlen(buff));

    // Step 8: Close the sockets
    close(client_sockfd);
    close(sockfd);

    return 0;
}
```

TCP server

TCP client

```c
// Main function
int main() {
    struct sockaddr_in servaddr; // Server address structure
    char buff[1500]; // Buffer for storing data
    int sockfd;

    // Step 1: Create a TCP socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("Socket creation failed...\n");
        exit(EXIT_FAILURE);
    }

    // Step 2: Define server address
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(9090); // Server port
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server IP address

    // Step 3: Connect to the server
    if (connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0) {
        perror("Connection to server failed...\n");
        exit(EXIT_FAILURE);
    }
    printf("Connected to server!\n");

    // Step 4: Send data to server
    strcpy(buff, "Hello, Server!");
    write(sockfd, buff, strlen(buff));

    // Step 5: Receive response from server
    bzero(buff, sizeof(buff));
    read(sockfd, buff, sizeof(buff));
    printf("From server: %s\n", buff);

    // Step 6: Close the socket
    close(sockfd);

    return 0;
}
```

NON-BLOCKING SOCKET

- A non-blocking socket allows a program to perform other tasks while waiting for network data, preventing the process from getting stuck indefinitely.

What is a Non-Blocking Socket?

- By default, sockets are blocking.
- If you try to read from a blocking socket but there's no data, the program waits indefinitely.
- Non-blocking sockets allow read(), recv(), or recvfrom() to return immediately even if no data is available.
- If no data is available, the function returns -1 and sets errno to EWOULDBLOCK or EAGAIN.

```
#include <unistd.h>
#include <fcntl.h>
.
.
.
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

This **modifies the socket file descriptor (sockfd)** to **non-blocking mode**.

Now, recv(), send(), or accept() will **return immediately** if no data is available.

MULTIPLEXING: SELECT() FUNCTION

- Multiplexing allows monitoring multiple file descriptors (sockets, pipes, files, etc.) to check which ones are ready for I/O operations without blocking execution.

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

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

- **readfds**: The file descriptors in this set are watched to see if the are ready for reading.
- **writefds**: The file descriptors in this set are watched to see if the are ready for writing.
- **exceptfds**: The file descriptors in this set are watched for "exceptional conditions“.
- **Timeout**: select() should block waiting for a file descriptor to become ready.
- **nfds** : file descriptors in each set are checked, up to this limit.

MULTIPLEXING (2) – WORKING WITH FD_SET IN SELECT()

fd_set readfds;

- Creates a **set of file descriptors** to monitor for readability.

// Clear an fd_set bofore use

FD_ZERO(&readfds)

- Initializes the **file descriptor set** by **clearing all bits**.
- This is necessary before adding any file descriptors.

// Add a descriptor to an fd_set

FD_SET(master_sock, &readfds);

- Adds **master_sock (server listening socket)** to the set.
- select() will now **monitor** this socket for incoming connections

// Remove a descriptor from an fd_set

FD_CLR(master_sock, &readfds);

- Removes **master_sock** from the set.
- This is useful when closing a connection.

//If something happened on the master socket , then its an incoming connection

FD_ISSET(master_sock, &readfds);

- Checks if master_sock is **ready** (e.g., has an incoming connection).
- FD_ISSET() returns **non-zero** if data is available.

SELECT EXAMPLE

```
fd_set   readset;
FD_ZERO(&readset);

FD_SET(0, &readset);
FD_SET(4, &readset);

select(5, &readset, NULL, NULL, NULL);

if  (FD_ISSET(0, &readset) {
/* something to be read from  0 */}
if  (FD_ISSET(4, &readset) {
   /* something to be read from  4 */
}
```

FD_ZERO(&readset);

•Clears the fd_set to ensure no previous descriptors remain.

FD_SET(0, &readset);

•Adds file descriptor **0** (typically **stdin**) to the monitoring set.

FD_SET(4, &readset);

•Adds file descriptor **4** (which could be a **network socket** or a **file**) to the monitoring set.

select(5, &readset, NULL, NULL, NULL);

•Blocks **until one of the file descriptors is ready** for reading.

•5 is nfds (highest file descriptor +1, i.e., max(0,4) + 1 = 5).

•**FD_ISSET(fd, &readset)** Returns **true** if fd is ready for reading.

TIMEOUT IN SELECT

Timeout Behavior in select()

- Immediate Timeout (tv_sec = 0, tv_usec = 0)
  - select() returns immediately, even if no file descriptors are ready.
  - This makes it non-blocking.
- Indefinite Blocking (timeout = NULL)
  - select() waits forever until at least one file descriptor is ready.

- Fixed Timeout (tv_sec > 0 or tv_usec > 0)
  - select() waits for the specified time before returning.
  - If no event occurs within the time, it returns 0 (timeout).

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

EXAMPLE

```c
#define STDIN 0  // Standard input file descriptor
int main() {
    struct timeval t_out;  // Timeout structure
    fd_set readfds;  // Set of file descriptors for reading

    // Step 1: Set timeout values
    t_out.tv_sec = 2;  // 2 seconds timeout
    t_out.tv_usec = 500000;  // 500,000 microseconds (0.5 seconds)

    // Step 2: Initialize the file descriptor set
    FD_ZERO(&readfds);  // Clear the set before adding file descriptors
    FD_SET(STDIN, &readfds);  // Add standard input (fd 0) to the read set

    // Step 3: Call select() to wait for input on stdin
    // We are only interested in readfds, so writefds and exceptfds are NULL
    int activity = select(STDIN + 1, &readfds, NULL, NULL, &t_out);

    // Step 4: Check if select() detected activity
    if (activity == -1) {
        perror("select() failed");
        exit(EXIT_FAILURE);
    } else if (FD_ISSET(STDIN, &readfds)) {  // Check if stdin is ready for reading
        printf("A key was pressed!\n");
    } else {  // No input detected within the timeout period
        printf("Timed out.\n");
    }
    return 0;
}
```

USEFUL FUNCTIONS

- Mapping between names and addresses (DNS)
  - Host name to address: gethostbyname() "www.google.com"
  - Host address to name: gethostbyaddr() "142.250.190.78"
- Mapping between different byte ordering schemes
  - Host to Network Short: htons()
  - Host to Network Long: htonl()
  - Network to Host Short: ntohs()
  - Network to Host Long: ntohl()

TCP ECHO SERVER EXAMPLE

```c
int main() {
    // Step 1: Create a socket
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Step 2: Define server address
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;  // Listen on all interfaces
    server_addr.sin_port = htons(8080);  // Port number (convert to network byte order)

    // Step 3: Bind the socket to the address and port
    if (bind(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    // Step 4: Listen for incoming connections
    if (listen(sockfd, 5) < 0) {  // Allow up to 5 pending connections
        perror("Listen failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }
    printf("Server listening on port 8080...\n");

    // Step 5: Accept and handle client connections
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);

    while (1) {  // Infinite loop to handle multiple clients
        int clientfd = accept(sockfd, (struct sockaddr*)&client_addr, &client_len);
        if (clientfd < 0) {
            perror("Accept failed");
            continue;
        }
        printf("Client connected!\n");

        // Step 6: Receive and send (Echo back)
        char buffer[1024];
        ssize_t n = recv(clientfd, buffer, sizeof(buffer) - 1, 0);
        if (n > 0) {
            buffer[n] = '\0';  // Null-terminate the received string
            printf("Received: %s\n", buffer);
            send(clientfd, buffer, n, 0);  // Echo back to client
        }

        // Step 7: Close the client socket
        close(clientfd);
    }

    // Step 8: Close the server socket (unreachable in an infinite loop)
    close(sockfd);

    return 0;
}
```

REFERENCES

- https://man7.org/linux/man-pages/
- TCP/ IP Sockets in C, Second edition, by Mickael J. Donahoo and Kenneth L. Calvert.
- Socket Programming presentation by Umit Karabiyik.