CNT Experiment 3 Write-up

Course	Computer Networking Technologies (Lab)
Faculty Name	Prof. Pramod Kanjalkar
Branch	CSE(AIML)
Div	A
Year	TY
Academic Year	2024-25

Submitted by:		
Roll No	Name	PRN
22	Pratham Gadkari	12211451
25	Shubham Jadhav	12210400
30	Yash Kate	12211614
35	Medhaj Kulkarni	12211456
48	Prateek Buthale	12211667
49	Shankar Rakh	12211013
56	Om Suryawanshi	12210892

What is WebSocket?

A WebSocket is a **continuous two-way communication channel** between clients and servers. The client could be any web browser, and the server could be any backend system. Using the HTTP request/response connections, the WebSocket can transmit any number of protocols and can provide server-to-client information without polling. It allows two-way data transfer in which we can simultaneously transfer data from client to server and vice versa.

The advanced technology opens an interactive two-way communication between the client and server. By using the WebSocket API, we can send information to the server and receive the runtime response without polling the server for a response.

WebSocket **uses a TCP-based network protocol** to specify the data transmission between networks. It is a widely used network protocol that almost all clients support because it is dependable and efficient. TCP protocol establishes communication between two endpoints. Often it is known as sockets.

Two-way connection technologies such as WebSocket API **allow two-way data flow simultaneously**, which provides a quick way to transfer the data. Thus, WebSocket allows a web application to communicate with the WebSocket server without interruption to provide real-time data.

To understand how WebSocket works, first, we need to understand how a website works over the HTTP protocol and website access the data without using the WebSocket API. Usually, the web pages are served over the HTTP protocol via creating an HTTP connection. Here, the data is served by the HTTP protocol as per the client's request.

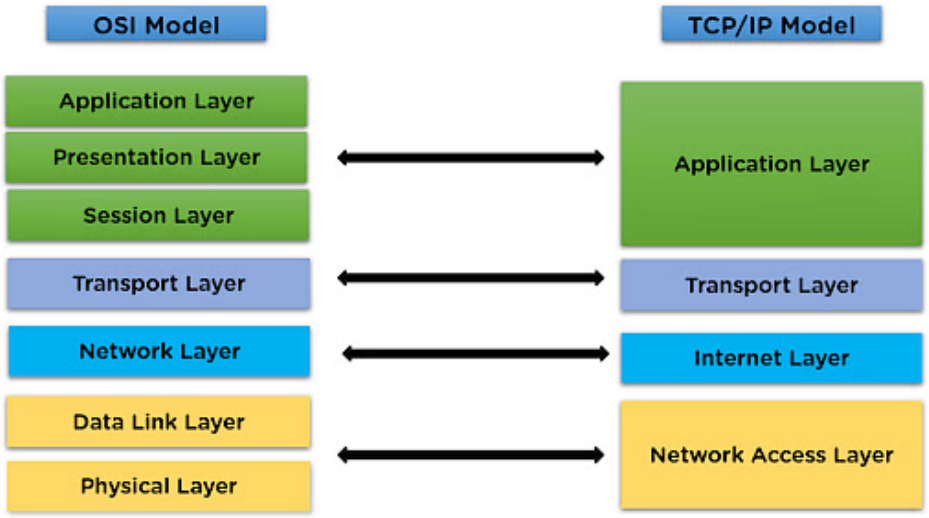
Each time the client requests the server, there is a specified response for every action, and the server sends the response accordingly. In a nutshell, HTTP protocol follows a request and response architecture which causes latency in the response.

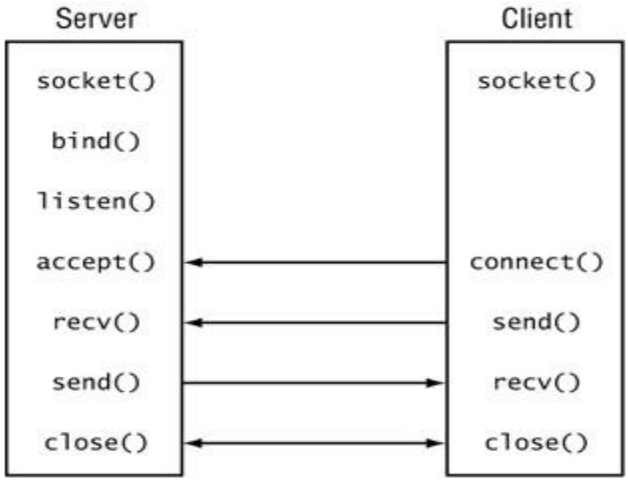
The WebSocket protocol works differently than the HTTP protocol. It can transfer data in real-time **utilizing a dynamic call-up method**. All we need to do is to establish a connection from the client to the server using the WebSocket protocol. The WebSocket protocol transmits the handshake to the client. It contains all the necessary information to identify the required data transmission information.

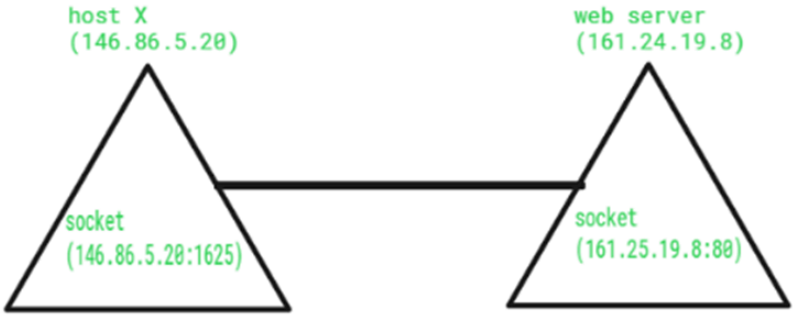
Once the connection is established, the channel is open, and it remains open after the handshake to allow for continuous communication. Thus, the **server can send the data to the client without requiring a request**. Hence, whenever the server receives new data, it transfers it to the client on the same channel without further request.

For starting a communication using the Socket, the client submits a request just like HTTP, and it opens the communication channel. After that, a TCP connection is maintained to transfer the data.

The socket approach is followed in web applications to send push notifications.







Most browsers support the protocol, including Google Chrome, Firefox, Microsoft Edge, Internet Explorer, Safari.

Comparison between Websocket and browser HTTP

Feature	WebSocket	HTTP
Overview	Full-duplex communication channels over a single TCP connection.	Stateless protocol for sending and receiving web pages and other data.
Key Characteristics		
Communication Model	Full-Duplex: Simultaneous two-way communication.	Request-Response: Client initiates requests; server sends back responses.
Connection Persistence	Persistent: Connection remains open, reducing overhead.	Stateless: Each request is independent; no state is retained between requests.
Latency	Lower latency: Ideal for real-time updates.	Higher overhead and latency due to headers in each request/response.
Data Transfer Efficiency	Efficient: Eliminates need for HTTP headers after initial handshake.	Higher overhead: Each request and response includes HTTP headers.
Data Formats	Handles both binary data and UTF-8 text.	Primarily text-based but can handle other file types.
Use Cases		
Real-time Applications	Chat applications, online gaming, collaborative tools.	Loading web pages, making API requests, transferring files and media.
Live Data Feeds	Stock market updates, sports scores.	Standard web browsing activities.
Collaborative Tools	Live document editing.	

Protocols

Protocols define the rules and conventions for data transmission and reception across a network, ensuring that communication between different systems or applications is standardized. Without protocols, effective communication between devices and software would be impossible.

- **Transmission Control Protocol (TCP):**
  TCP is a connection-oriented protocol that ensures reliable data transmission by establishing a connection between sender and receiver, guaranteeing data delivery in the correct order, and handling retransmission of lost packets.
  Use Cases: Web browsing (HTTP/HTTPS), email (SMTP, IMAP), file transfers (FTP).
- **User Datagram Protocol (UDP):**
  UDP is a connectionless protocol that sends data without establishing a connection. It does not guarantee data order or reliability, making it faster but less reliable than TCP.
  Use Cases: Streaming media, online gaming, DNS lookups.

System Calls

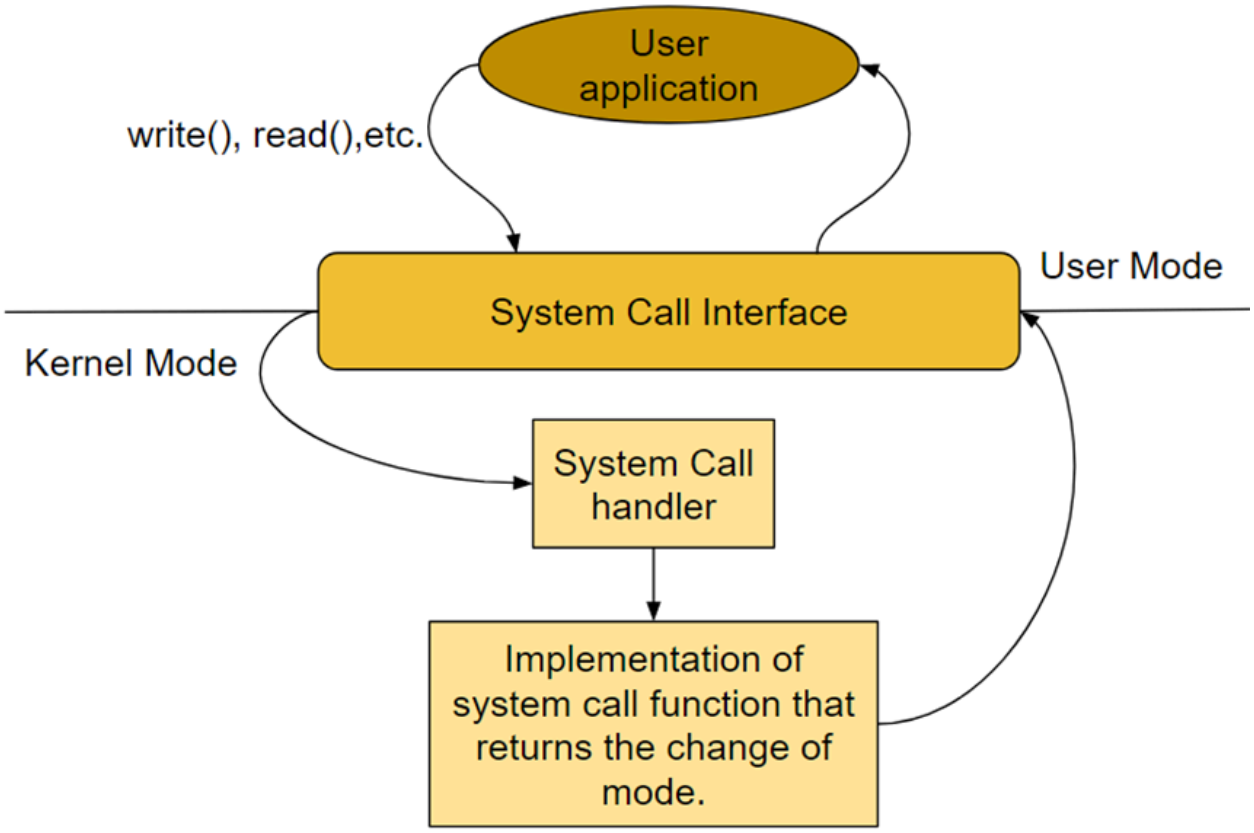
System calls are mechanisms that allow user applications to interact with the operating system (OS). They provide a controlled way to request OS services and resources, such as file operations, memory management, and process control. System calls facilitate communication between user-space programs and the OS kernel, which operates with direct hardware access.

How System Calls Work:

1. **User Space vs. Kernel Space:**
   - **User Space:** Where regular applications run without direct hardware access.
   - **Kernel Space:** Where the OS kernel operates with direct access to hardware and system resources.
2. **Invocation:**
   Applications invoke system calls to perform operations requiring OS intervention. This involves specifying a system call number (identifier) and passing necessary arguments.
3. **Transition:**
   Invoking a system call triggers a context switch from user mode to kernel mode, allowing the OS to execute the operation safely.

4**. Execution and Return:**

The kernel performs the requested operation and returns the result to the application, followed by a context switch back to user mode



Socket Programming Frameworks

In computer networks, a socket programming framework provides a set of APIs and tools to enable communication between devices over a network. Sockets are the endpoints of this communication, serving as an interface for sending and receiving data. Here's an overview of socket programming and some commonly used frameworks

1. **Socket API:**
    a. The Socket API defines a set of functions that can be used to create, configure, and manage sockets. Common operations include:
        i. **Creating a Socket:** Allocates resources for a new socket.
        ii. **Binding:** Associates a socket with a specific IP address and port number.
        iii. **Listening:** Prepares a socket to accept incoming connection requests (for server applications).
        iv. **Accepting:** Accepts a new incoming connection (for server applications).
        v. **Connecting:** Establishes a connection to a remote socket (for client applications).
        vi. **Sending/Receiving Data:** Transmits and receives data through the socket.
        vii. **Closing:** Terminates the socket connection and releases resources.
2. **Frameworks by Language:**
    a. **POSIX Sockets (C/C++):**
        i. **Overview:** Standard API for Unix-like operating systems, widely used in system-level and network programming.
        ii. **Usage:** Suitable for developing high-performance applications like web servers, database systems, and network utilities.
        iii. **Features:** Provides low-level control over socket operations and supports both IPv4 and IPv6.
    b. **Java Sockets:**
        i. **Overview:** Part of the `java.net` package, providing a higher-level abstraction for socket programming.
        ii. **Usage:** Commonly used for developing platform-independent network applications, such as chat servers, file transfer applications, and distributed systems.
        iii. **Features:** Supports both blocking and non-blocking I/O operations, and integrates well with Java's multithreading capabilities.
    c. **Python Sockets:**
        i. **Overview:** Part of the `socket` module in the Python standard library, known for its simplicity and ease of use.
        ii. **Usage:** Ideal for rapid development and prototyping of network applications, including web clients, servers, and IoT applications.
        iii. **Features:** Simplifies socket operations with a high-level API and supports both synchronous and asynchronous programming.
    d. **.NET Sockets (C#):**
        i. **Overview:** Part of the `System.Net.Sockets` namespace, providing robust support for network programming in the .NET ecosystem.
        ii. **Usage:** Frequently used in enterprise applications, web services, and cloud-based solutions.
        iii. **Features:** Offers a rich set of functionalities, including support for both TCP and UDP, asynchronous operations, and integration with the .NET framework.

e. **Boost Asio (C++):**
   i. **Overview:** Part of the Boost library, providing a modern and highly flexible framework for network programming.
   ii. **Usage:** Suitable for developing high-performance, cross-platform applications, including real-time systems, game servers, and financial trading platforms.
   iii. **Features:** Supports both synchronous and asynchronous operations, and provides a comprehensive set of I/O facilities.

Practical Applications

1. **Web Servers and Clients:**
   - Sockets are fundamental in the development of web servers and clients. They enable the exchange of HTTP requests and responses between browsers and web servers.
2. **Real-Time Communication:**
   - Applications like VoIP (Voice over IP), video conferencing, and online gaming rely on sockets for real-time data transmission.
3. **Distributed Systems:**
   - In distributed computing, sockets facilitate communication between different nodes in a network, enabling the coordination of tasks and data sharing.
4. **Internet of Things (IoT):**
   - IoT devices use sockets to communicate with each other and with central servers, enabling smart homes, industrial automation, and connected health systems.
5. **Remote Procedure Calls (RPC):**
   - Sockets are used in RPC systems to allow a program to execute procedures on a remote server as if they were local function calls.

.

Conclusion

Socket programming frameworks are essential for building robust and scalable networked applications. By abstracting the complexities of network communication, they enable developers to focus on application logic and functionality. Understanding the architecture and capabilities of different socket frameworks is crucial for choosing the right tools and technologies for your network programming needs.

Execution of Socket on a single computer (TCP)

Algorithm

Server-Side Algorithm:

1. Socket Creation:
    a. The server starts by creating a socket using the socket() system call.
    b. The AF_INET indicates that the socket will use IPv4, SOCK_STREAM specifies that it will be a TCP socket (as opposed to a UDP socket), and 0 allows the protocol to be chosen automatically.
2. Binding:
    a. The bind() system call binds the socket to an IP address and port number. In this case, the IP address is set to INADDR_ANY, which means the server will accept connections on any of the available network interfaces, and the port number is 8080.
3. Listening:
    a. The listen() system call puts the server in a passive mode where it waits for clients to connect. The number 3 specifies the maximum number of pending connections in the queue.
4. Accepting Connections:
    a. The accept() system call accepts an incoming connection from a client. It creates a new socket for the connection and returns a file descriptor for the connected socket.
5. Reading from the Client:
    a. The server reads data from the connected client using the read() system call, which reads up to BUFFER_SIZE bytes from the socket into the buffer.
6. Sending a Response:
    a. After receiving the message from the client, the server responds with a message ("Hello from server") using the send() system call.
7. Closing the Connection:
    a. The server closes the connection using the close() system call, first closing the socket associated with the client connection and then the main server socket.

Client-Side Algorithm:

1. Socket Creation:
    a. The client begins by creating a socket using the socket() system call with similar parameters as the server (AF_INET for IPv4, SOCK_STREAM for TCP).

2. Specifying Server Address:
    a. The sockaddr_in structure is filled in with the server's IP address and port number. The IP address is specified as 127.0.0.1, which refers to the localhost (i.e., the server is on the same machine as the client), and the port number is set to 8080.
3. Converting IP Address:
    a. The inet_pton() function converts the server IP address from a human-readable form (e.g., "127.0.0.1") to a format suitable for the sin_addr field.
4. Connecting to the Server:
    a. The connect() system call attempts to establish a connection to the server using the address and port specified in the serv_addr structure.
5. Sending a Message:
    a. After successfully connecting, the client sends a message ("Hello from client") to the server using the send() system call.
6. Receiving a Response:
    a. The client then waits for a response from the server. The read() system call reads up to BUFFER_SIZE bytes of data from the server and stores it in the buffer.
7. Closing the Connection:
    a. Finally, the client closes the socket and terminates.

Code

Server:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
   int server_fd, new_socket;
   struct sockaddr_in address;
   int addrlen = sizeof(address);
   char buffer[BUFFER_SIZE] = {0};

   if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
       perror("socket failed");
       exit(EXIT_FAILURE);
   }

   address.sin_family = AF_INET;
```

```
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))
< 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 3) < 0) {
        perror("listen");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen)) < 0) {
        perror("accept");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    read(new_socket, buffer, BUFFER_SIZE);
    printf("Message received: %s\n", buffer);
    send(new_socket, "Hello from server", strlen("Hello from server"),
0);

    close(new_socket);
    close(server_fd);

    return 0;
}
```

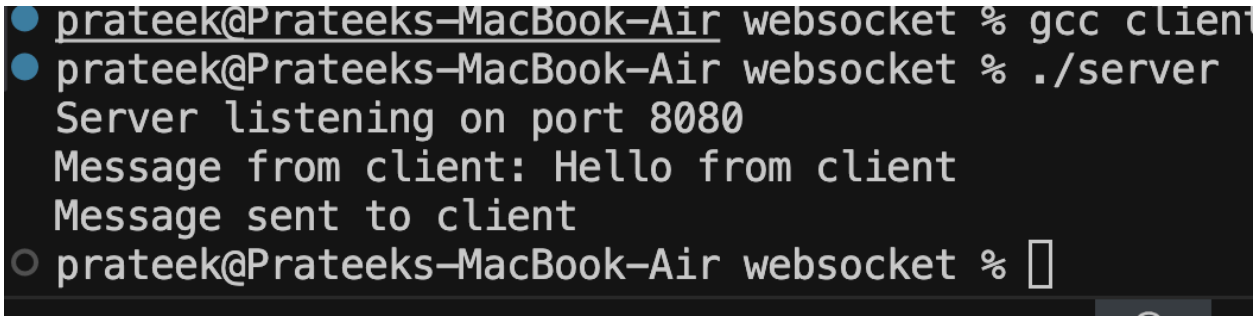
Client:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
```

```
    char buffer[BUFFER_SIZE] = {0};

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }

    send(sock, "Hello from client", strlen("Hello from client"), 0);
    printf("Hello message sent\n");
    read(sock, buffer, BUFFER_SIZE);
    printf("Message from server: %s\n", buffer);

    close(sock);

    return 0;
}
```

Results

```
prateek@Prateeks-MacBook-Air websocket % ./client
Message sent to server
Message from server: Hello from server
prateek@Prateeks-MacBook-Air websocket %
```



Execution of Socket on two computers (TCP)

Algorithm

1. Set up the server:
    a. Create a WebSocket server that listens for incoming connections.
    b. When a client connects, the server should be able to send and receive messages from the client.
2. Set up the client:
    a. Create a WebSocket client that connects to the server.
    b. The client should be able to send and receive messages to/from the server.
3. Enable message exchange via command line:
    a. Implement command-line interfaces (CLIs) for both the server and the client to send messages.
    b. Display incoming messages on the command line.

Code

Server:

```
import asyncio
import websockets

async def handle_client(websocket, path):
    async for message in websocket:
        print(f"Received message from client: {message}")
        response = input("Enter response to client: ")
        await websocket.send(response)

async def main():
    server = await websockets.serve(handle_client, "localhost", 8765)
```

```
    print("Server started at ws://localhost:8765")
    await server.wait_closed()

if __name__ == "__main__":
    asyncio.run(main())
```

Client:

```
import asyncio
import websockets

async def communicate():
    uri = "ws://localhost:8765"
    async with websockets.connect(uri) as websocket:
        while True:
            message = input("Enter message to send to server: ")
            await websocket.send(message)
            response = await websocket.recv()
            print(f"Received response from server: {response}")

if __name__ == "__main__":
    asyncio.run(communicate())
```

Execution of Socket on a single computer (UDP)

Algorithm for UDP Server (`server.c`)

1. **Initialize:**
    - Create a buffer and structures for server/client addresses.
2. **Create Socket:**

    - Use `socket()` to create a UDP socket.
3. **Set Address:**
    - Set server's IP address to `INADDR_ANY` and port to `PORT`.
4. **Bind Socket:**
    - Bind the socket to the server's address.
5. **Communication Loop:**
    - Receive client message using `recvfrom()`.
    - Print the message.
    - Send acknowledgement to the client using `sendto()`.
6. **Close Socket (on exit).**

Algorithm for UDP Client (`client.c`)

1. **Initialize:**
    - Create a buffer and structure for the server address.
2. **Create Socket:**
    - Use `socket()` to create a UDP socket.
3. **Set Address:**
    - Set server's IP address and port.
4. **Send Message:**
    - Get user input.
    - Send the message using `sendto()`.
5. **Receive Acknowledgment:**
    - Receive acknowledgment from the server using `recvfrom()` and print it.
6. **Close Socket.**

CODE:

Server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
   int sockfd;
   char buffer[BUFFER_SIZE];
   struct sockaddr_in server_addr, client_addr;

   // Create socket file descriptor
   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
       perror("Socket creation failed");
       exit(EXIT_FAILURE);
   }

   memset(&server_addr, 0, sizeof(server_addr));
   memset(&client_addr, 0, sizeof(client_addr));

   // Fill server information
   server_addr.sin_family = AF_INET; // IPv4
   server_addr.sin_addr.s_addr = INADDR_ANY;
   server_addr.sin_port = htons(PORT);

   // Bind the socket with the server address
   if (bind(sockfd, (const struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
       perror("Bind failed");
       close(sockfd);
       exit(EXIT_FAILURE);
   }

   int len, n;
   len = sizeof(client_addr); // Length of client address

   printf("UDP Server is up and waiting for messages...\n");

   while (1) {
       // Receive message from client
       n = recvfrom(sockfd, buffer, BUFFER_SIZE, 0, (struct sockaddr
*) &client_addr, &len);
       buffer[n] = '\0'; // Null terminate the string
       printf("Client: %s\n", buffer);

       // Send acknowledgment to client
       const char *ack = "Message received";
       sendto(sockfd, ack, strlen(ack), 0, (const struct sockaddr *)
&client_addr, len);
       printf("Acknowledgment sent to client.\n");
   }
```

```
    close(sockfd);
    return 0;
}
```

client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 9090
#define BUFFER_SIZE 1024

int main() {
    int sockfd;
    char buffer[BUFFER_SIZE];
    struct sockaddr_in server_addr;

    // Create socket file descriptor
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));

    // Fill server information
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    int n, len;

    // Get input message from user
    printf("Enter a message: ");
    fgets(buffer, BUFFER_SIZE, stdin);
```

```
    // Send message to server
    sendto(sockfd, buffer, strlen(buffer), 0, (const struct sockaddr *)
&server_addr, sizeof(server_addr));
    printf("Message sent to server.\n");

    // Receive acknowledgment from server
    len = sizeof(server_addr);
    n = recvfrom(sockfd, buffer, BUFFER_SIZE, 0, (struct sockaddr *)
&server_addr, &len);
    buffer[n] = '\0';
    printf("Server: %s\n", buffer);
    close(sockfd);
    return 0;
}
```

Output

server.c

```
prateek@Prateeks-MacBook-Air UDP % ./server
UDP Server is up and waiting for messages...
Client: Hello from CLIENT

Acknowledgment sent to client.
```

client.c

```
prateek@Prateeks-MacBook-Air UDP % ./client
Enter a message: Hello from CLIENT
Message sent to server.
Server: Message received
prateek@Prateeks-MacBook-Air UDP %
```