Bansilal Ramnath Agarwal Charitable Trust's
**VISHWAKARMA INSTITUTE OF TECHNOLOGY – PUNE**

# COMPUTER NETWORKS

| | |
|---|---|
| **Branch** | CS - AIML |
| **Division** | A |
| **Batch** | 2 |
| **GR-no** | 12311305 |
| **Roll no** | 04 |
| **Name** | Adwyte Karandikar |

## Experiment No. 3a:

**TITLE:** Implementation of Client-Server Communication using TCP Sockets (Hello Message Exchange and File Transfer)

### PROBLEM STATEMENT:

Write the client server programs using TCP Berkeley socket primitives for wired /wireless network for following:
**a. to say Hello to Each other**
**b. File transfer**

## THEORY:

In computer networks, socket programming enables communication between processes running on different devices over a network. A socket acts as an endpoint for sending and receiving data. The Berkeley socket API provides primitives to create, bind, listen, accept, connect, send, and receive messages.

**There are two common types of sockets:**

1. Stream sockets (TCP): Provide reliable, connection-oriented communication.
2. Datagram sockets (UDP): Provide connectionless, unreliable communication.

In this experiment, we use TCP sockets, which guarantee delivery, maintain order, and handle retransmissions in case of data loss.

**Two mini-programs are implemented:**

1. Hello Message Exchange: A client and server connect and exchange a greeting message.
2. File Transfer: A client requests a file, and the server sends it over a TCP connection, ensuring complete and ordered file transfer.

This demonstrates how real-world applications (like chat apps, FTP servers, or web servers) are built using TCP socket programming.

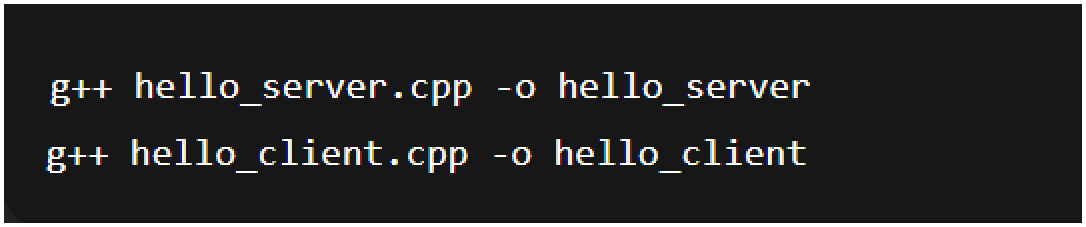
## STEP-BY-STEP PROCEDURE:

### Part A: Hello Message Exchange

1. Write and save the server program (hello_server.cpp) that:
    - Creates a socket using socket().
    - Binds the socket to an IP address and port.
    - Listens for incoming client connections using listen().
    - Accepts a connection with accept().
    - Reads the message sent by the client.
    - Sends back a "Hello from Server" message.

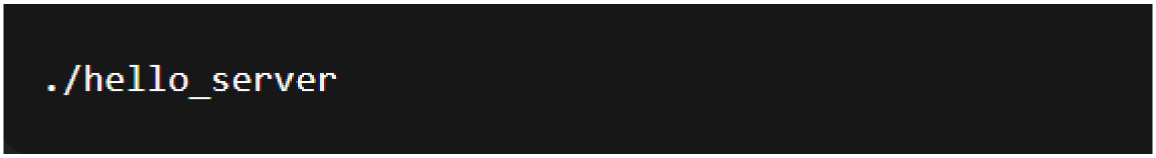
2. Write and save the client program (hello_client.cpp) that:
    - Creates a socket using socket().
    - Connects to the server using connect().
    - Sends a "Hello from Client" message.
    - Receives and prints the server’s response.

3. Compile both programs:



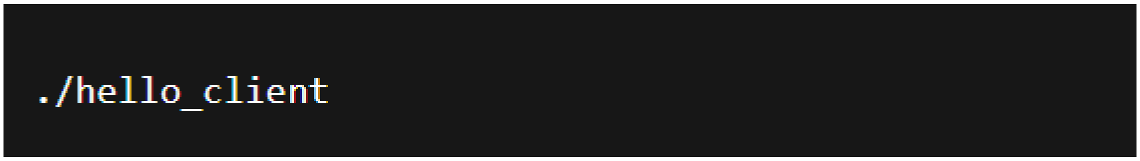
```
g++ hello_server.cpp -o hello_server
g++ hello_client.cpp -o hello_client
```

4. Run the server first in one terminal:



```
./hello_server
```

5. Run the client in another terminal:

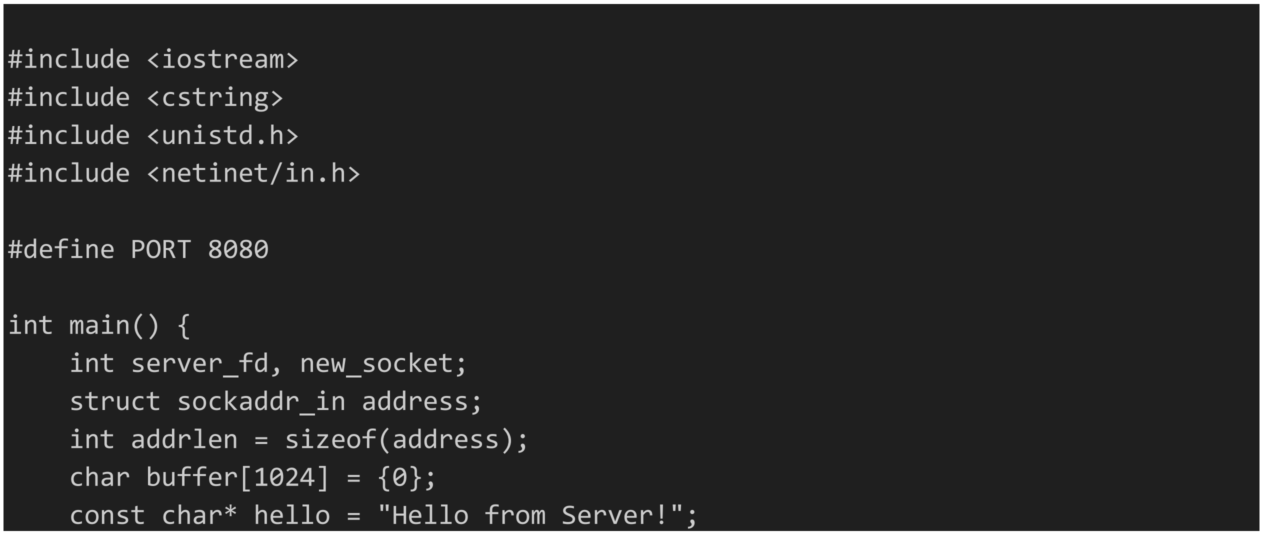


```
./hello_client
```

6. Observe the exchanged "Hello" messages on both terminals.

## A. Hello message exchange

### 1. hello_sever.cpp



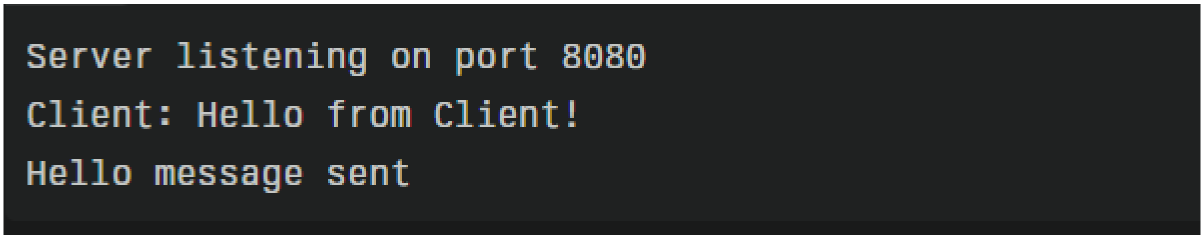
```
#include <iostream>
#include <cstring>
#include <unistd.h>
#include <netinet/in.h>

#define PORT 8080

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    const char* hello = "Hello from Server!";
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 0) {
        perror("Socket failed");
        return 1;
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY; // Listen on all interfaces
    address.sin_port = htons(PORT);

    bind(server_fd, (struct sockaddr*)&address, sizeof(address));
    listen(server_fd, 3);

    std::cout << "Server listening on port " << PORT << std::endl;

    new_socket = accept(server_fd, (struct sockaddr*)&address,
(socklen_t*)&addrlen);
    read(new_socket, buffer, 1024);
    std::cout << "Client: " << buffer << std::endl;

    send(new_socket, hello, strlen(hello), 0);
    std::cout << "Hello message sent" << std::endl;

    close(new_socket);
    close(server_fd);
    return 0;
}
```

### 2. hello_client.cpp

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

#define PORT 8080

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[1024] = {0};
    const char* hello = "Hello from Client!";

    sock = socket(AF_INET, SOCK_STREAM, 0);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr); // Replace with server
IP

    connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    send(sock, hello, strlen(hello), 0);
    std::cout << "Hello message sent" << std::endl;

    read(sock, buffer, 1024);
    std::cout << "Server: " << buffer << std::endl;

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

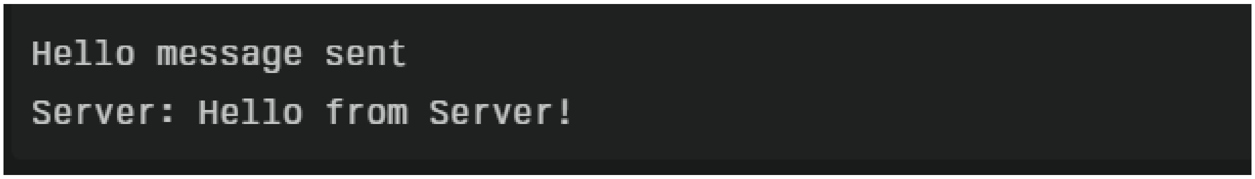
**Screenshots/Output:**

1) **Server Terminal (hello_server)**



```
Server listening on port 8080
Client: Hello from Client!
Hello message sent
```

2) **Client Terminal (hello_client)**



```
Hello message sent
Server: Hello from Server!
```

## B. Peer to Peer File Transfer

### 1) file_sever.cpp

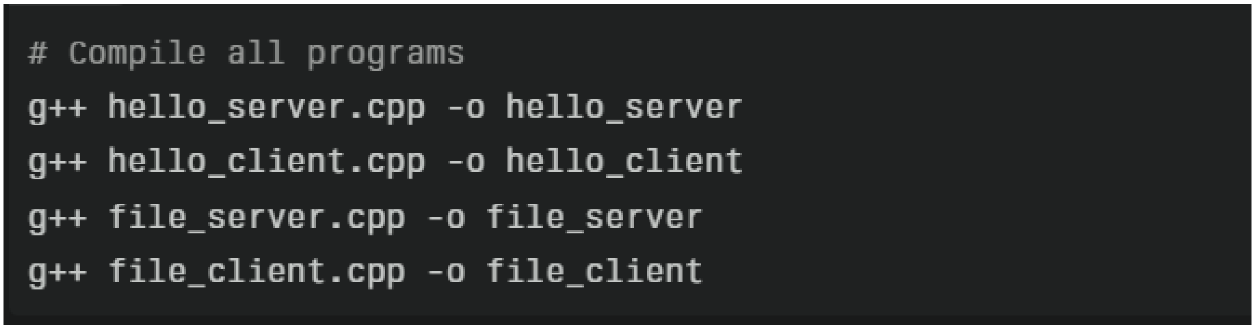
```
#include <iostream>
#include <fstream>
#include <unistd.h>
#include <netinet/in.h>
#include <cstring>

#define PORT 9090

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024];

    server_fd = socket(AF_INET, SOCK_STREAM, 0);

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

    bind(server_fd, (struct sockaddr*)&address, sizeof(address));
    listen(server_fd, 3);
    std::cout << "Waiting for file request on port " << PORT << std::endl;

    new_socket = accept(server_fd, (struct sockaddr*)&address,
(socklen_t*)&addrlen);

    // Receive requested filename
    read(new_socket, buffer, 1024);
    std::string filename = buffer;
    std::ifstream file(filename, std::ios::binary);

    if (!file) {
        std::cerr << "File not found: " << filename << std::endl;
        close(new_socket);
        close(server_fd);
        return 1;
    }
    while (!file.eof()) {
        file.read(buffer, sizeof(buffer));
        send(new_socket, buffer, file.gcount(), 0);
    }

    std::cout << "File sent successfully.\n";
    file.close();
    close(new_socket);
    close(server_fd);
    return 0;
}
```

### 2) file_client.cpp

```
#include <iostream>
#include <fstream>
#include <unistd.h>
#include <arpa/inet.h>
#include <cstring>

#define PORT 9090

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[1024];
    std::string filename = "sample.txt"; // File to request

    sock = socket(AF_INET, SOCK_STREAM, 0);

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr); // Replace with server
IP

    connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    send(sock, filename.c_str(), filename.length(), 0);

    std::ofstream outfile("received_" + filename, std::ios::binary);

    int bytesRead;
    while ((bytesRead = read(sock, buffer, 1024)) > 0) {
        outfile.write(buffer, bytesRead);
    }

    std::cout << "File received and saved as received_" << filename << std::endl;

    outfile.close();
    close(sock);
    return 0;
}
```

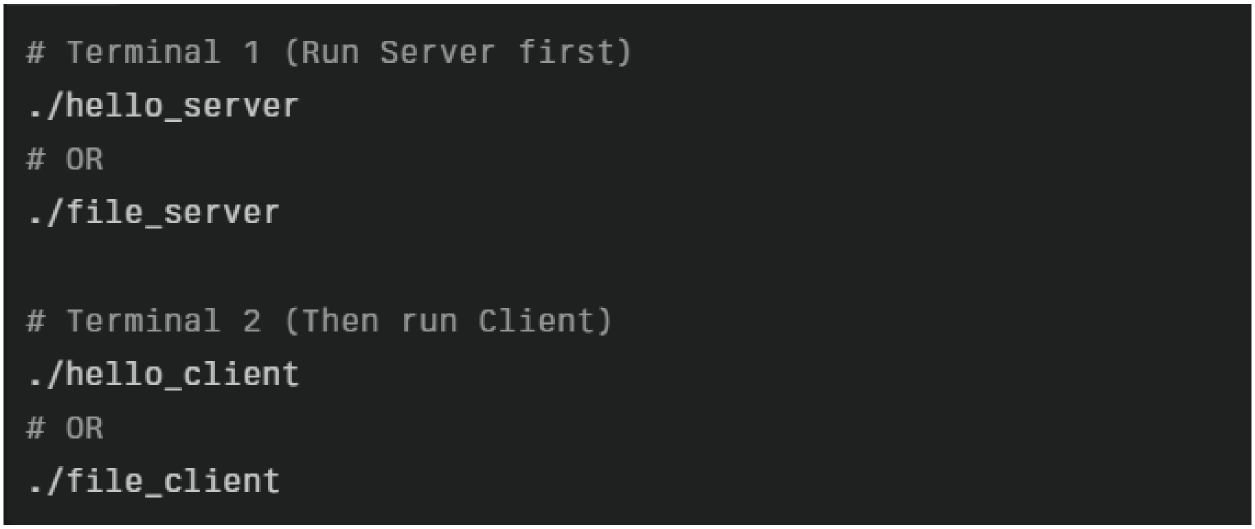
## Compile and Run Instruction:

### Compile:



```
# Compile all programs
g++ hello_server.cpp -o hello_server
g++ hello_client.cpp -o hello_client
g++ file_server.cpp -o file_server
g++ file_client.cpp -o file_client
```

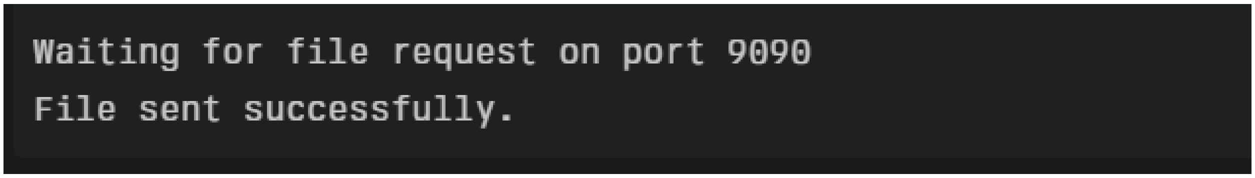
### Run:



```
# Terminal 1 (Run Server first)
./hello_server
# OR
./file_server

# Terminal 2 (Then run Client)
./hello_client
# OR
./file_client
```

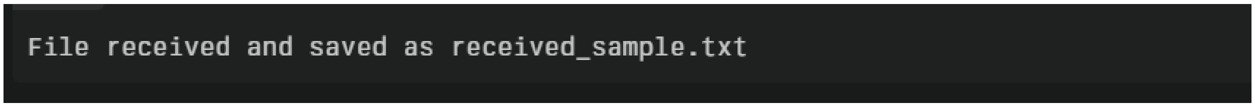
### Screenshots/Output:

1) **Server Terminal (file_server)**



```
Waiting for file request on port 9090
File sent successfully.
```

2) **Client Terminal (file_client)**



```
File received and saved as received_sample.txt
```

## CONCLUSION:

In this experiment, we successfully implemented client-server communication using TCP sockets. The first program demonstrated a simple Hello message exchange, while the second showed a file transfer mechanism between a client and a server.

Through this, we learned:

- How to use Berkeley socket primitives (socket(), bind(), listen(), accept(), connect(), send(), recv()).
- The working of TCP as a reliable, connection-oriented protocol.
- How to establish communication between two systems and transfer data securely.

This forms the foundation for building more advanced network applications like chat systems, FTP, and web servers.