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

COMPUTER NETWORKS

Branch	CS - AIML
Division	A
Batch	2
GR-no	12311493
Roll no	54
Name	Atharva Kangralkar

Experiment No. 3a:

TITLE: Socket programming

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

LINUX SOCKET PROGRAMMING

The Berkeley socket interface, an API, allows communications between hosts or between processes on one computer, using the concept of a socket. It can work with many different I/O devices and drivers, although support for these depends on the operating- system implementation. This interface implementation is implicit for TCP/IP, and it is therefore one of the fundamental technologies

underlying the Internet. It was first developed at the University of California, Berkeley for use on UNIX systems. All modern operating systems now have some implementation of the Berkeley socket interface, as it has become the standard interface for connecting to the Internet.
Programmers can make the socket interfaces accessible at three different levels, most powerfully and fundamentally at the RAW socket level. Very few applications need the degree of control over outgoing communications that this provides, so RAW sockets support was intended to be available only on computers used for developing Internet- related technologies. In recent years, most operating systems have implemented support for it anyway, including Windows XP.

The header files

The Berkeley socket development library has many associated header files. They include:
<sys/socket.h>
Definitions for the most basic of socket structures with the BSD socket API
<sys/types.h>
Basic data types associated with structures within the BSD socket API
<netinet/in.h>
Definitions for the socketaddr_in{} and other base data structures.
<sys/un.h>
Definitions and data type declarations for SOCK_UNIX streams

TCP provides the concept of a **connection**. A process creates a TCP socket by calling the `socket()` function with the parameters `PF_INET` or `PF_INET6` and `SOCK_STREAM`.

Server

Setting up a simple TCP server involves the following steps:

1. **Creating a TCP socket**
   Call `socket()` to create the socket.
2. **Binding the socket to a listening port**
   - Declare a `sockaddr_in` structure.
   - Clear it using `bzero()` or `memset()`.
   - Set the `sin_family` to `AF_INET` or `AF_INET6`.
   - Set the `sin_port` to the listening port (in network byte order).
     - Use `htons()` to convert a short int from host to network byte order.
3. **Preparing the socket to listen**
   Call `listen()` to make the socket a listening socket.
4. **Accepting incoming connections**
   Call `accept()`.
   - This blocks until an incoming connection is received.
   - Returns a socket descriptor for the accepted connection.
   - The original socket descriptor remains as the listening descriptor and can be used for further `accept()` calls until closed.

5. **Communicating with the remote host**
   Use `send()` and `recv()` for data transmission.
6. **Closing sockets**
   Call `close()` on each socket once it is no longer needed.
   - If `fork()` is used, each process must close the sockets it knows about.
   - Two processes should not use the same socket simultaneously

Client

Setting up a TCP client involves the following steps:

1. **Creating a TCP socket**
   Call `socket()` to create the socket.
2. **Connecting to the server**
   Call `connect()`, passing a `sockaddr_in` structure with:
   - `sin_family` set to `AF_INET` or `AF_INET6`
   - `sin_port` set to the server's listening port (in network byte order)
   - `sin_addr` set to the server's IPv4 or IPv6 address (in network byte order)
3. **Communicating with the server**
   Use `send()` and `recv()` for data transmission.
4. **Terminating the connection**
   Call `close()` to clean up.
   - If `fork()` is used, each process must close the socket it knows about.

Berkeley Socket Programming (TCP)

The **Berkeley socket interface** is an API that allows communication between hosts across a network or between processes on the same machine. It was first developed at the University of California, Berkeley, for UNIX systems and has since become the **standard interface for Internet communication**.

It supports multiple communication protocols (e.g., TCP, UDP, RAW sockets) and is available in all modern operating systems.

Important Functions

- socket(domain, type, protocol) → creates a socket.
- bind(sockfd, addr, addrlen) → assigns IP and port to a socket.
- listen(sockfd, backlog) → prepares socket to accept connections.
- accept(sockfd, cliaddr, addrlen) → accepts a client connection.
- connect(sockfd, serv_addr, addrlen) → client connects to server.
- send(sockfd, buffer, length, flags) → sends data.
- recv(sockfd, buffer, length, flags) → receives data.
- close(sockfd) → closes the socket.

Blocking vs Non-Blocking

- **Blocking**: Functions like accept(), recv() block until data/connection is available.
- **Non-blocking**: Socket functions return immediately, even if no data is available. (Configured via fcntl() or ioctl().)

Cleaning Up

Every socket created with socket() must be closed using close().
This releases system resources and avoids memory leaks.

Applications in Assignment

a) Hello Client-Server

- Server waits for connection.
- Client connects and sends "Hello Server".
- Server replies "Hello Client".

b) File Transfer

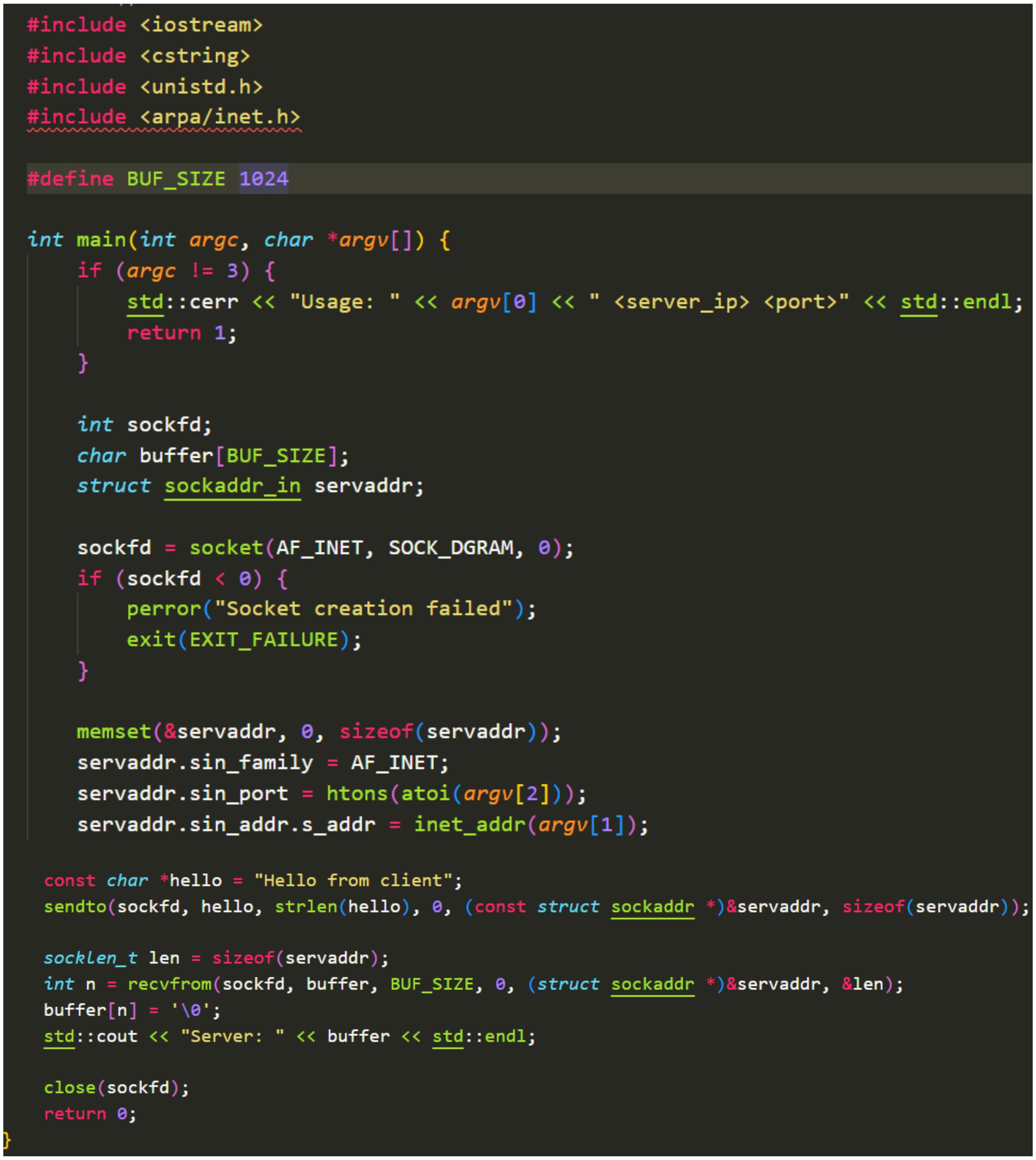
Server:

1. Create, bind, listen, and accept.
2. Open a file, read data, and send() it to the client.
3. Close file and socket.

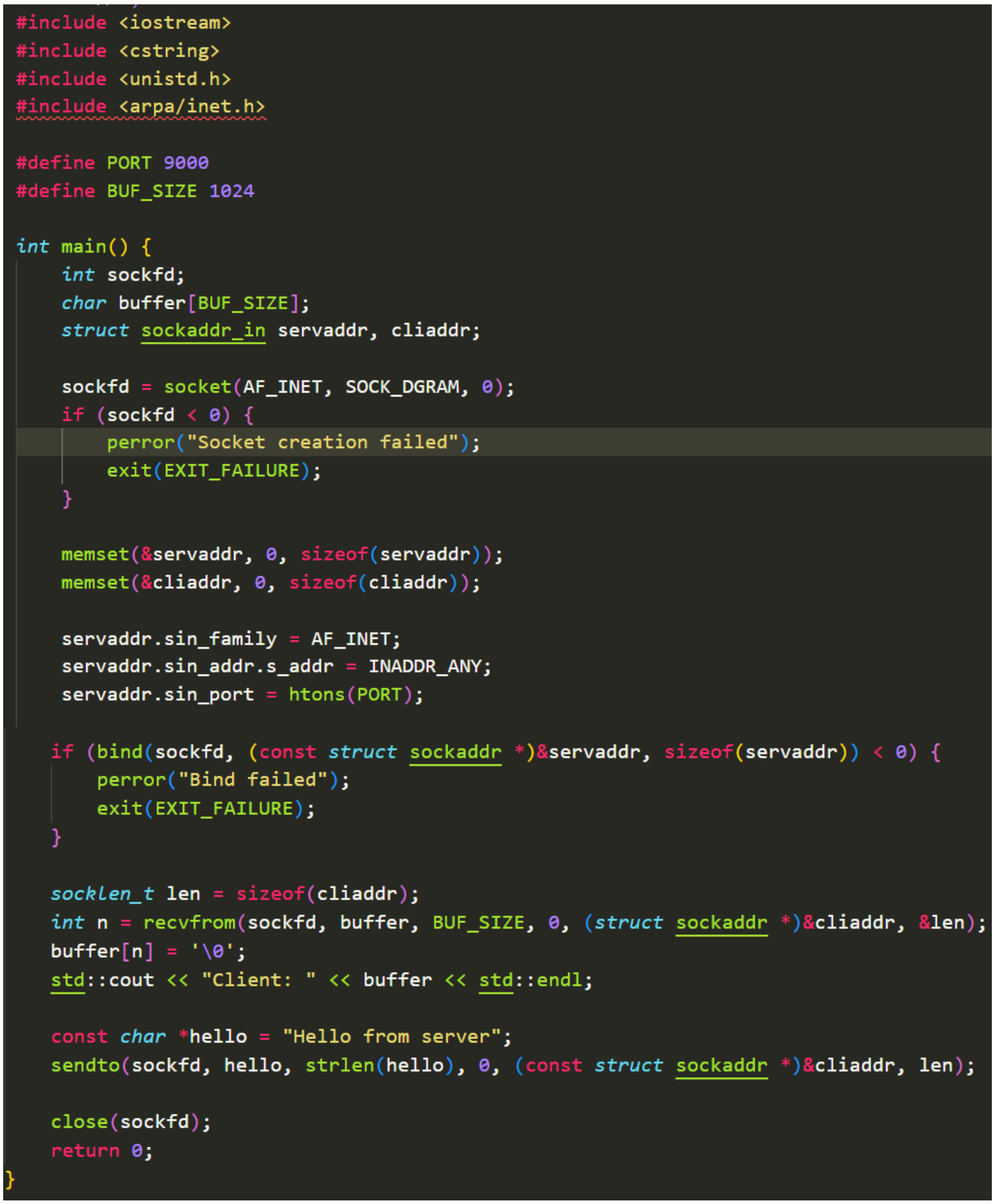
Client:

1. Create and connect to server.
2. Receive file data using recv().
3. Save received data into a file.
4. Close socket.

Code:

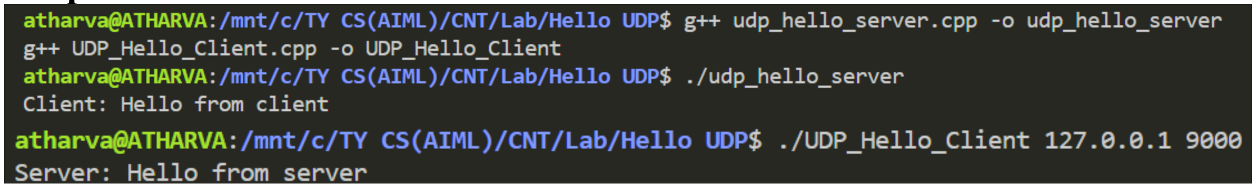


```cpp
#include <iostream>
#include <cstring>
#include <unistd.h>
#include <arpa/inet.h>

#define BUF_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <server_ip> <port>" << std::endl;
        return 1;
    }

    int sockfd;
    char buffer[BUF_SIZE];
    struct sockaddr_in servaddr;

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

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(atoi(argv[2]));
    servaddr.sin_addr.s_addr = inet_addr(argv[1]);

    const char *hello = "Hello from client";
    sendto(sockfd, hello, strlen(hello), 0, (const struct sockaddr *)&servaddr, sizeof(servaddr));

    socklen_t len = sizeof(servaddr);
    int n = recvfrom(sockfd, buffer, BUF_SIZE, 0, (struct sockaddr *)&servaddr, &len);
    buffer[n] = '\0';
    std::cout << "Server: " << buffer << std::endl;

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



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

#define PORT 9000
#define BUF_SIZE 1024

int main() {
    int sockfd;
    char buffer[BUF_SIZE];
    struct sockaddr_in servaddr, cliaddr;

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

    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));

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

   if (bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
       perror("Bind failed");
       exit(EXIT_FAILURE);
   }

   socklen_t len = sizeof(cliaddr);
   int n = recvfrom(sockfd, buffer, BUF_SIZE, 0, (struct sockaddr *)&cliaddr, &len);
   buffer[n] = '\0';
   std::cout << "Client: " << buffer << std::endl;

   const char *hello = "Hello from server";
   sendto(sockfd, hello, strlen(hello), 0, (const struct sockaddr *)&cliaddr, len);

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

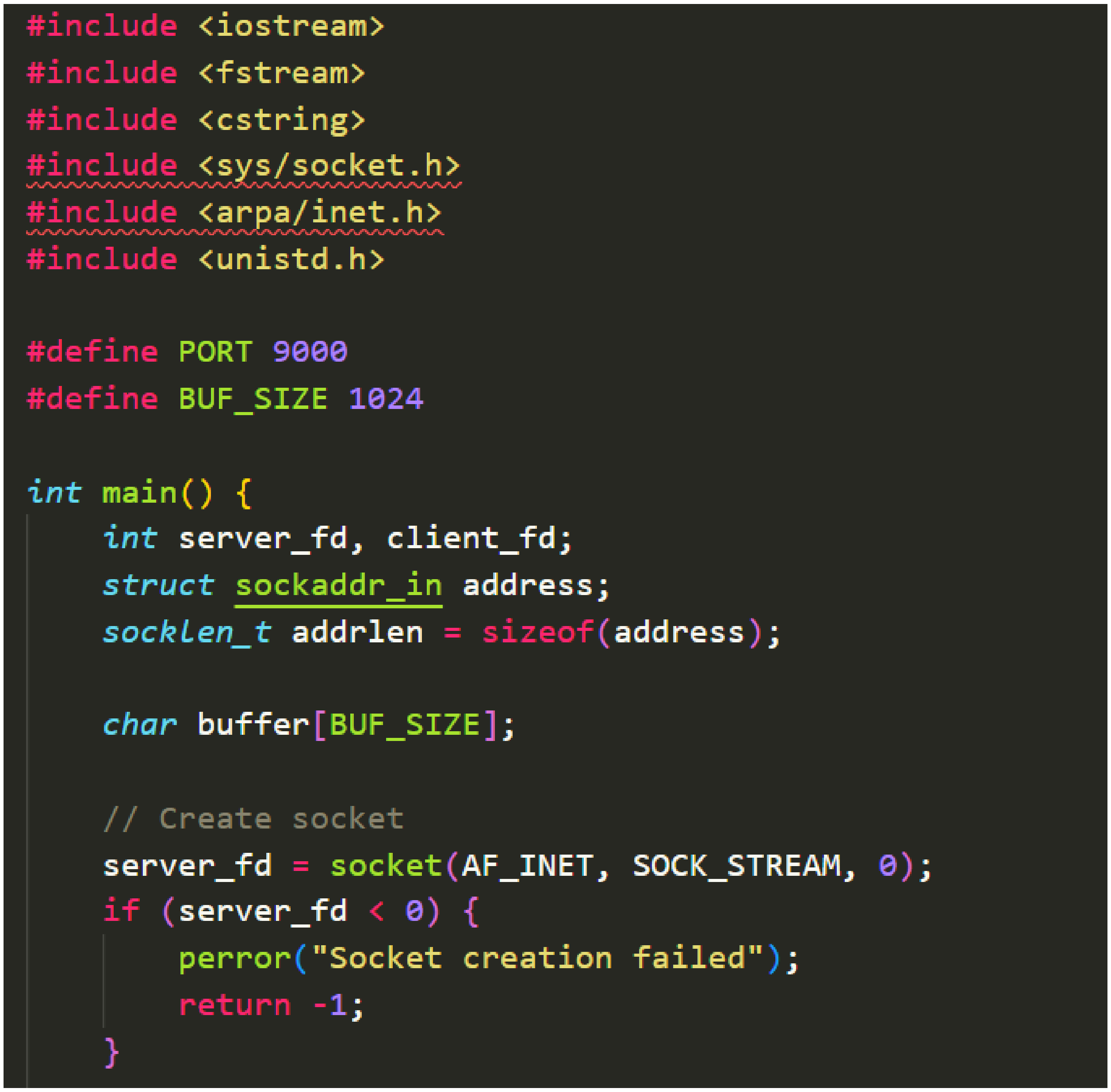
Output:



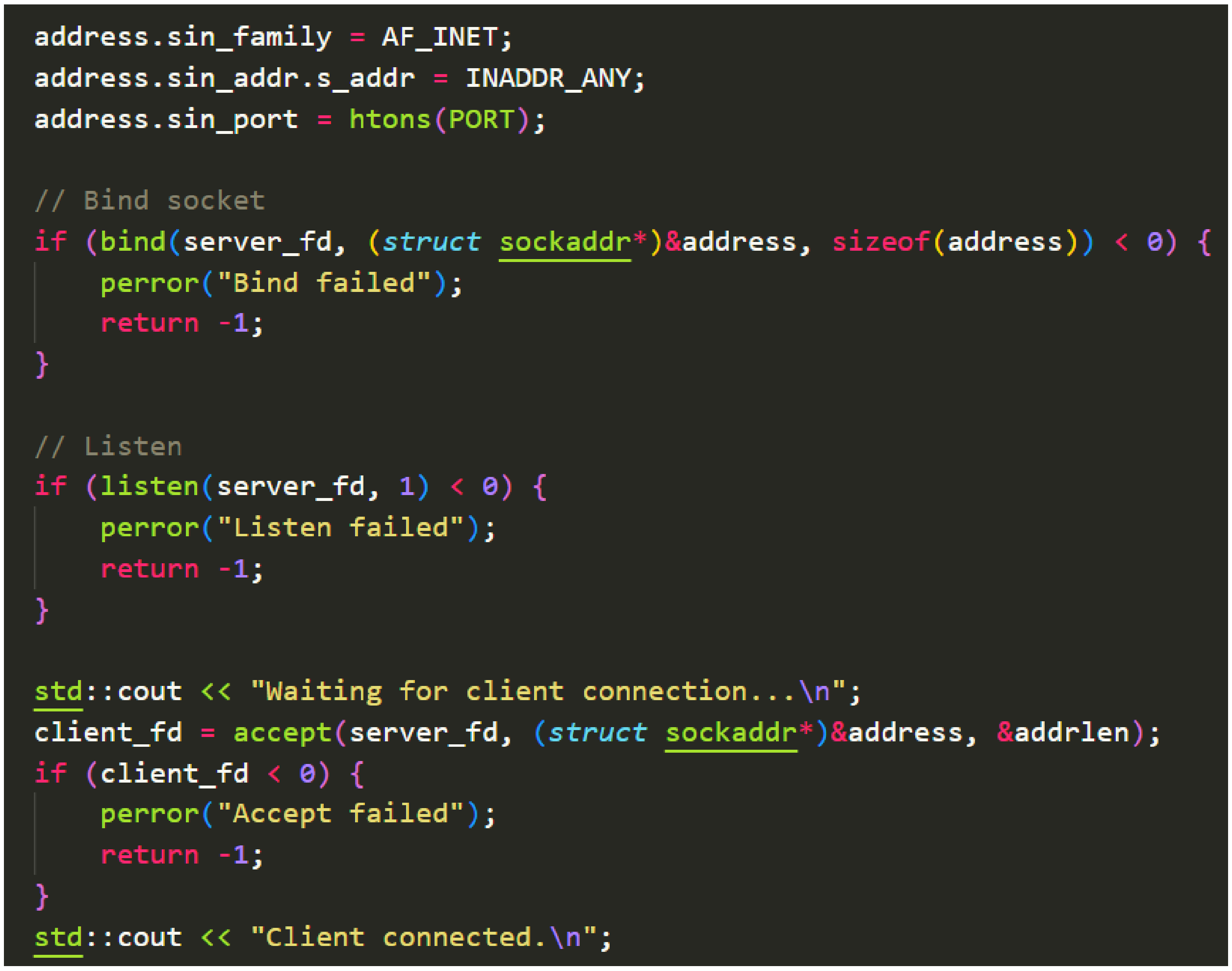
```
atharva@ATHARVA:/mnt/c/TY CS(AIML)/CNT/Lab/Hello UDP$ g++ udp_hello_server.cpp -o udp_hello_server
g++ UDP_Hello_Client.cpp -o UDP_Hello_Client
atharva@ATHARVA:/mnt/c/TY CS(AIML)/CNT/Lab/Hello UDP$ ./udp_hello_server
Client: Hello from client
atharva@ATHARVA:/mnt/c/TY CS(AIML)/CNT/Lab/Hello UDP$ ./UDP_Hello_Client 127.0.0.1 9000
Server: Hello from server
```

For files:

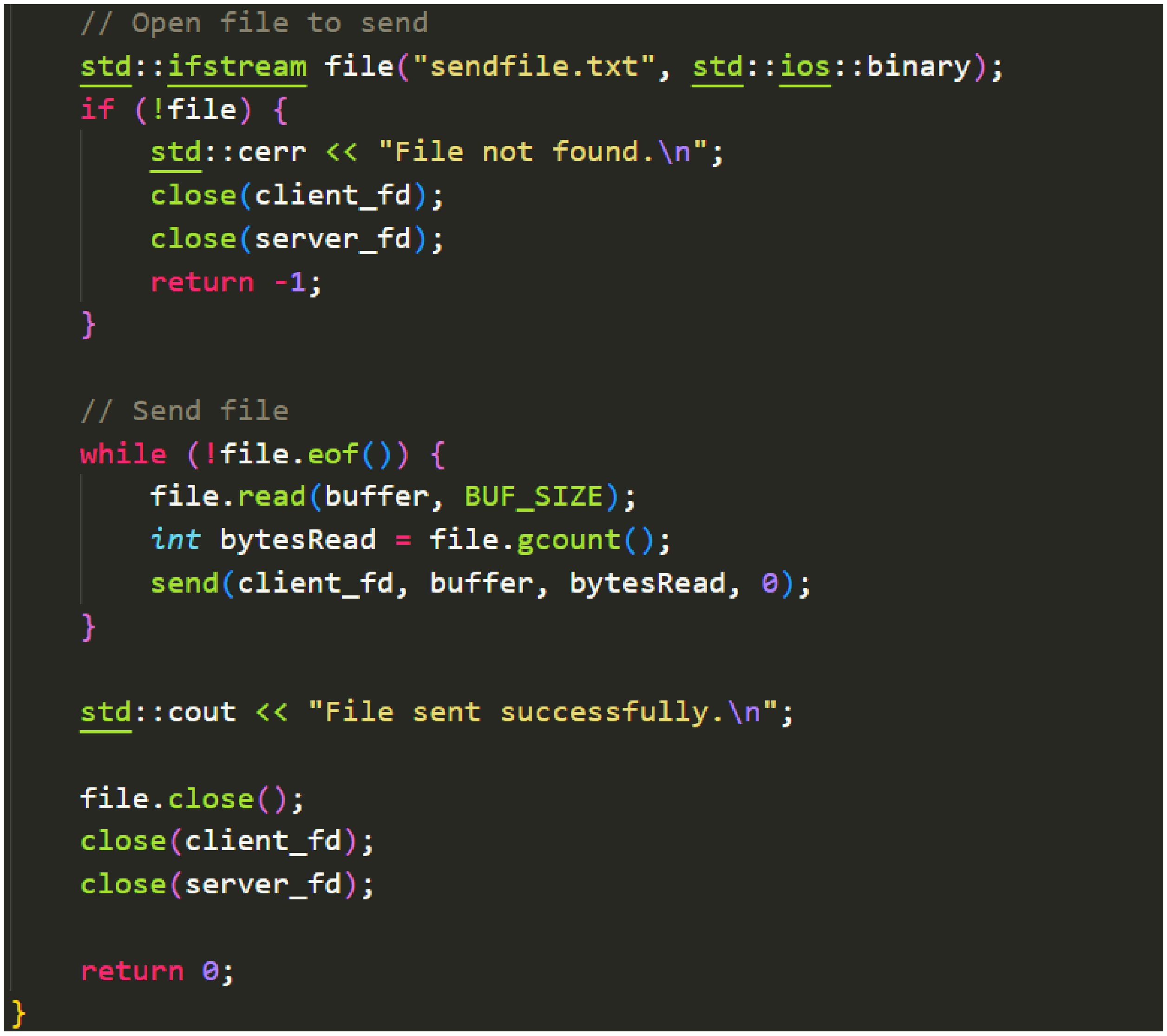
Code:-



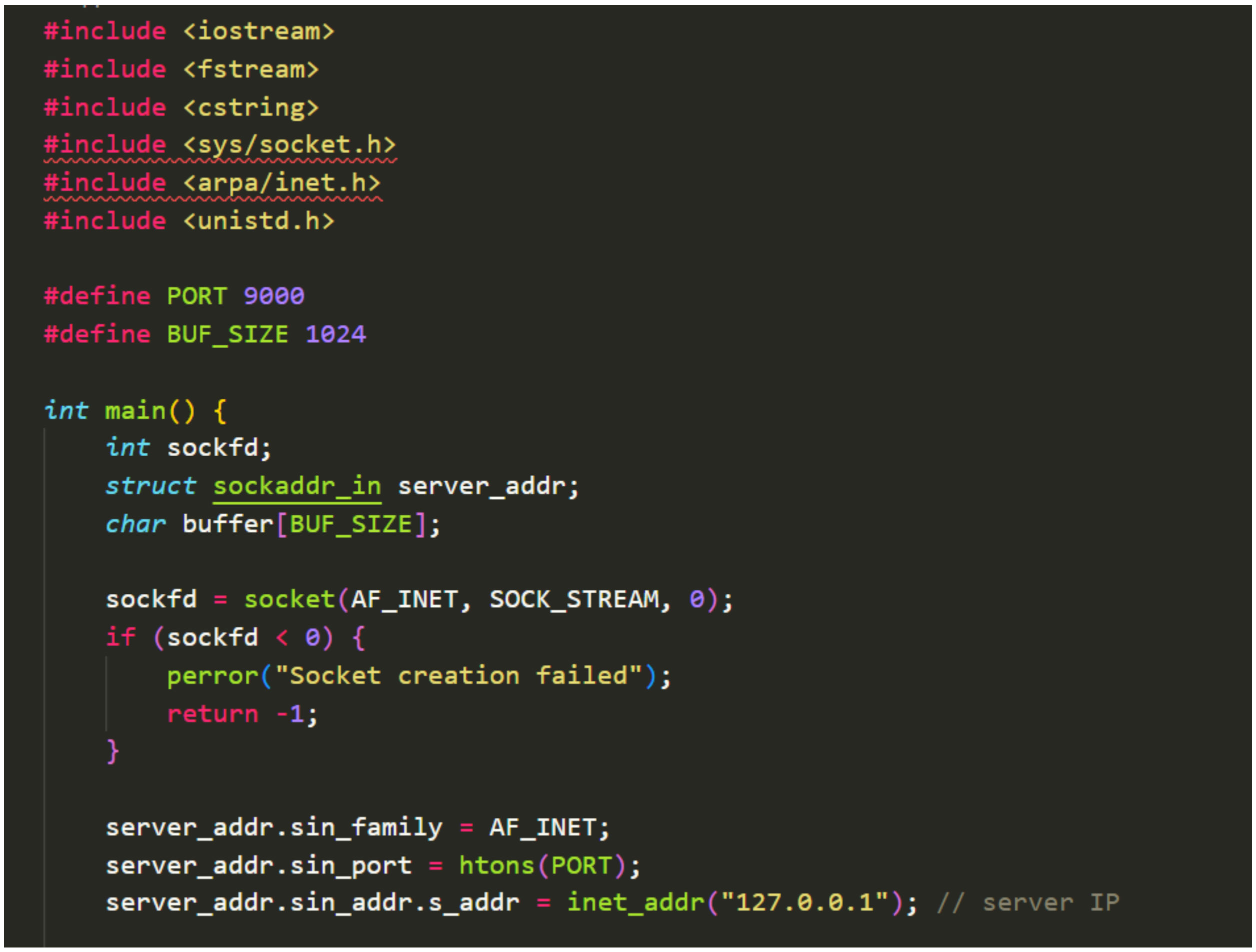
```cpp
#include <iostream>
#include <fstream>
#include <cstring>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

#define PORT 9000
#define BUF_SIZE 1024

int main() {
    int server_fd, client_fd;
    struct sockaddr_in address;
    socklen_t addrlen = sizeof(address);

    char buffer[BUF_SIZE];

    // Create socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("Socket creation failed");
        return -1;
    }
```

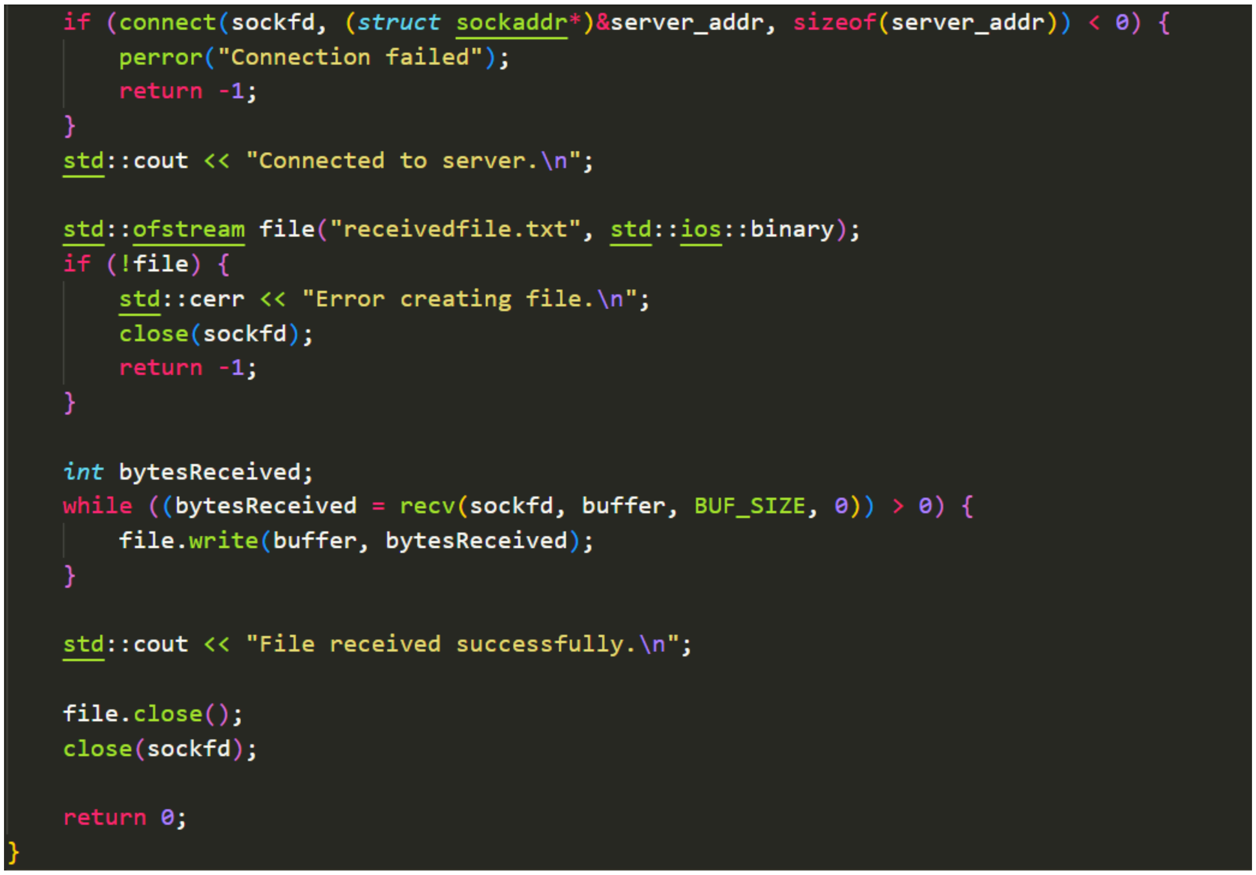


```
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(PORT);

// Bind socket
if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
    perror("Bind failed");
    return -1;
}

// Listen
if (listen(server_fd, 1) < 0) {
    perror("Listen failed");
    return -1;
}

std::cout << "Waiting for client connection...\n";
client_fd = accept(server_fd, (struct sockaddr*)&address, &addrlen);
if (client_fd < 0) {
    perror("Accept failed");
    return -1;
}
std::cout << "Client connected.\n";
```



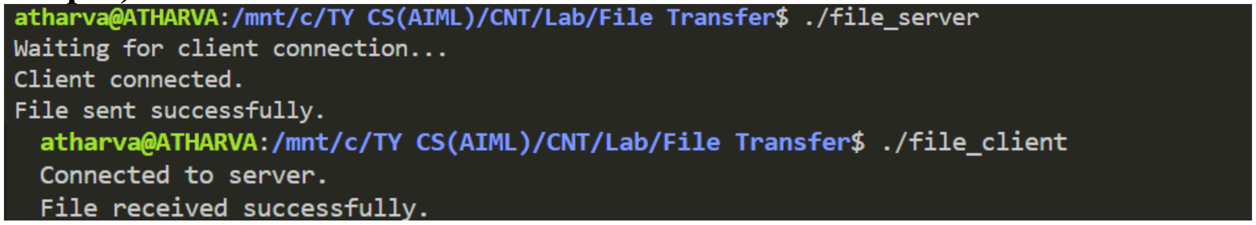
```
    // Open file to send
    std::ifstream file("sendfile.txt", std::ios::binary);
    if (!file) {
        std::cerr << "File not found.\n";
        close(client_fd);
        close(server_fd);
        return -1;
    }

    // Send file
    while (!file.eof()) {
        file.read(buffer, BUF_SIZE);
        int bytesRead = file.gcount();
        send(client_fd, buffer, bytesRead, 0);
    }

    std::cout << "File sent successfully.\n";

    file.close();
    close(client_fd);
    close(server_fd);

    return 0;
}
```


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

#define PORT 9000
#define BUF_SIZE 1024

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

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Socket creation failed");
        return -1;
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1"); // server IP
```



```cpp
    if (connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("Connection failed");
        return -1;
    }
    std::cout << "Connected to server.\n";

    std::ofstream file("receivedfile.txt", std::ios::binary);
    if (!file) {
        std::cerr << "Error creating file.\n";
        close(sockfd);
        return -1;
    }

    int bytesReceived;
    while ((bytesReceived = recv(sockfd, buffer, BUF_SIZE, 0)) > 0) {
        file.write(buffer, bytesReceived);
    }

    std::cout << "File received successfully.\n";

    file.close();
    close(sockfd);

    return 0;
}
```

Output)



```
atharva@ATHARVA:/mnt/c/TY CS(AIML)/CNT/Lab/File Transfer$ ./file_server
Waiting for client connection...
Client connected.
File sent successfully.
  atharva@ATHARVA:/mnt/c/TY CS(AIML)/CNT/Lab/File Transfer$ ./file_client
  Connected to server.
  File received successfully.
```