UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

A REPORT ON

DISTRIBUTED SYSTEM

BY

BI12-379 Đào Xuân Quý

DEPARTMENT OF INFORMATION TECHNOLOGY AND COMMUNICATION

Contents

1 Designing the protocol

In order to establish a messaging protocol, it was necessary to implement a handshake mechanism. This ensures the reliability of communication between the server and the client. Once the connection is established, the server initiates communication by sending a message. Upon successful verification by the client, tasks commence, and information exchange between the client and server ensues. This iterative process continues until the file transfer is completed.

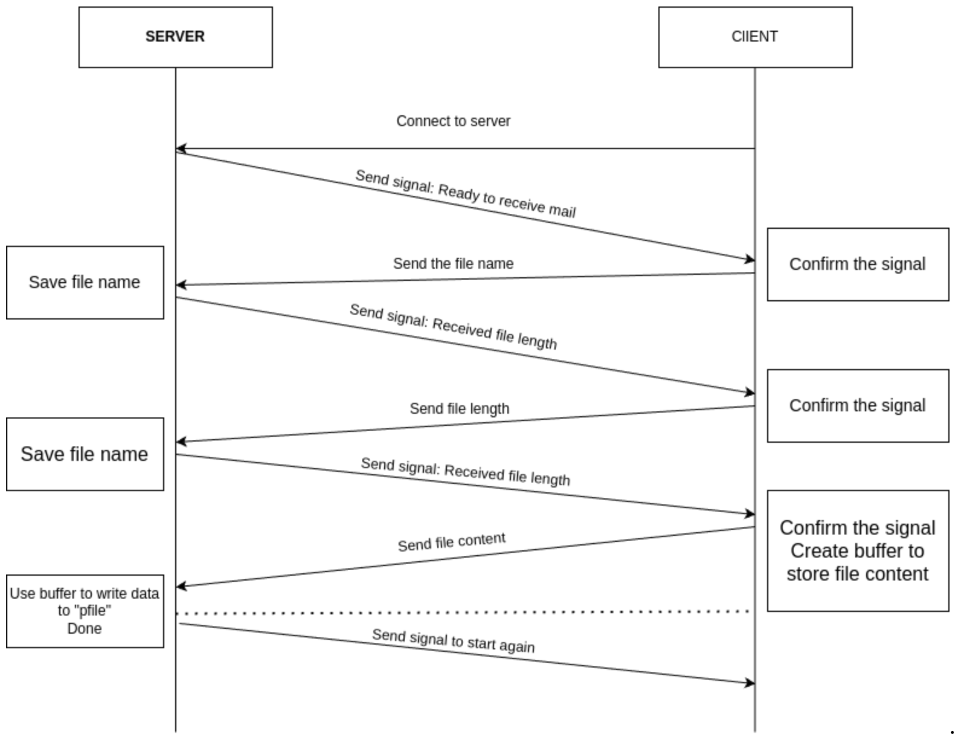
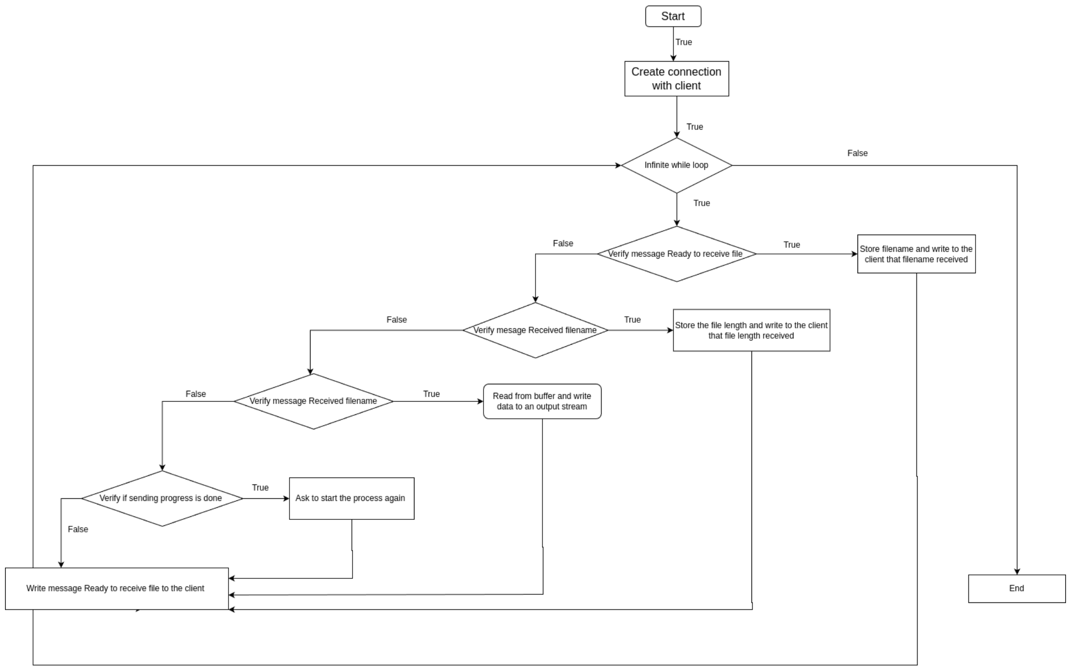

SERVER
ClIENT
Connect to server
Send signal: Ready to receive mail
Confirm the signal
Save file name
Send the file name
Send signal: Received file length
Confirm the signal
Send file length
Save file name
Send signal: Received file length
Confirm the signal Create buffer to store file content
Send file content
Use buffer to write data to "pfile" Done
Send signal to start again

Figure 1: The protocol

2 Organizing the system

The setup involves a client and a server communicating via sockets. Both initialize a socket before initiating file transfer. The server binds to a port and awaits client activity, while the client configures its address and connects to the server.

To initiate file transfer, the server first sends a readiness message to signal its preparedness. The client acknowledges this message, prompts the user to input the file's name, and forwards it to the server. Upon receiving the file name, the server sends a confirmation message. The client then determines the file's length and transmits this information to the server, which confirms receipt.



Start
True
Create connection with client
True
Infinite while loop
False
True
False
Verify message Ready to receive file
True
Store filename and write to the client that filename received
False
Verify mesage Received filename
True
Store the file length and write to the client that file length received
False
Verify message Received filename
True
Read from buffer and write data to an output stream
Verify if sending progress is done
True
Ask to start the process again
False
Write message Ready to receive file to the client
End

Figure 2: Server

Upon confirmation, the client proceeds to send the file data to the server and waits for the process to complete. The server receives the data and prompts the user to press a button to initiate another cycle of the process.

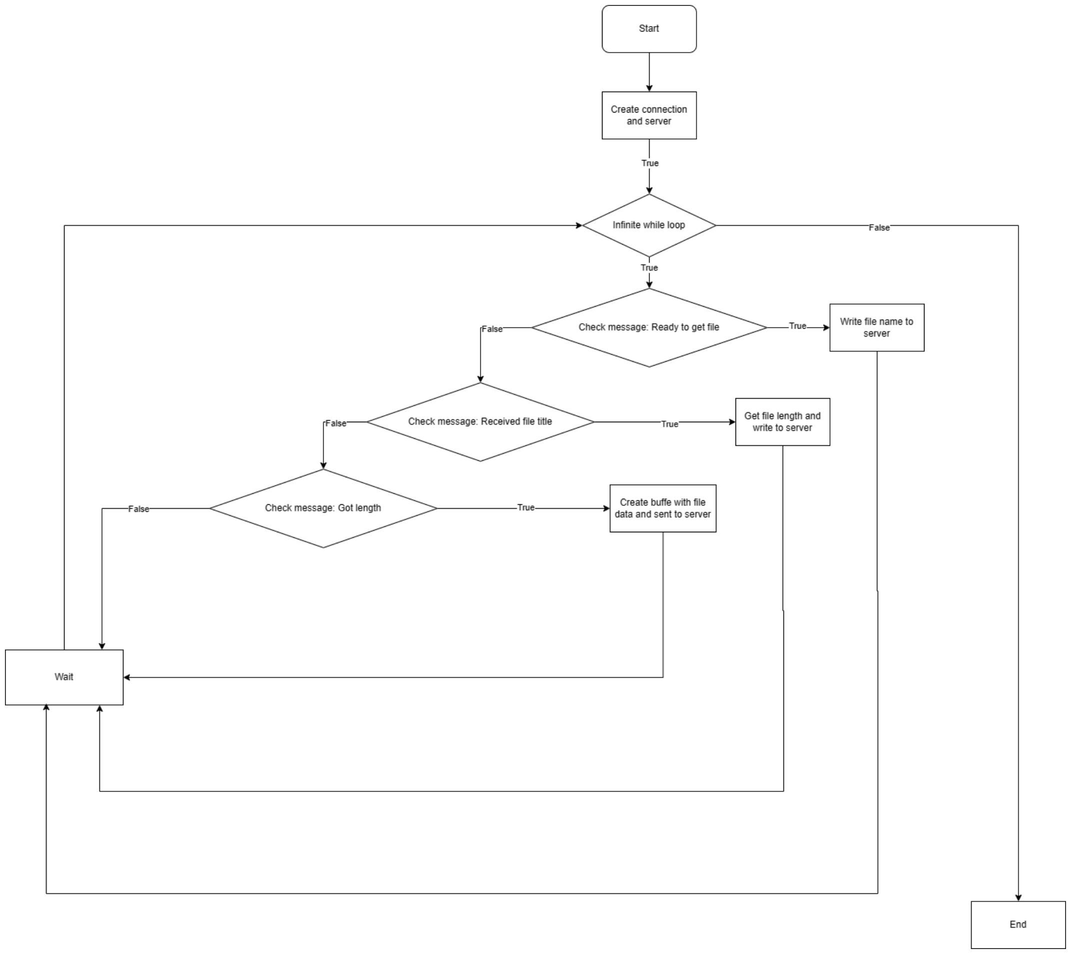
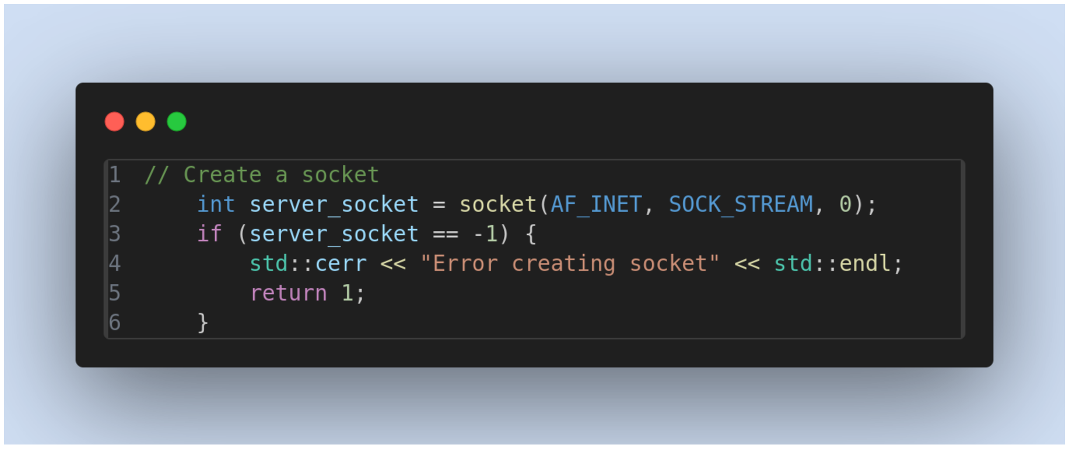

Start
Create connection and server
True
Infinite while loop
False
True
Check message: Ready to get file
False
True
Write file name to server
Check message: Received file title
False
True
Get file length and write to server
Check message: Got length
False
True
Create buffe with file data and sent to server
Wait
End

Figure 3: Client

3 Implementation

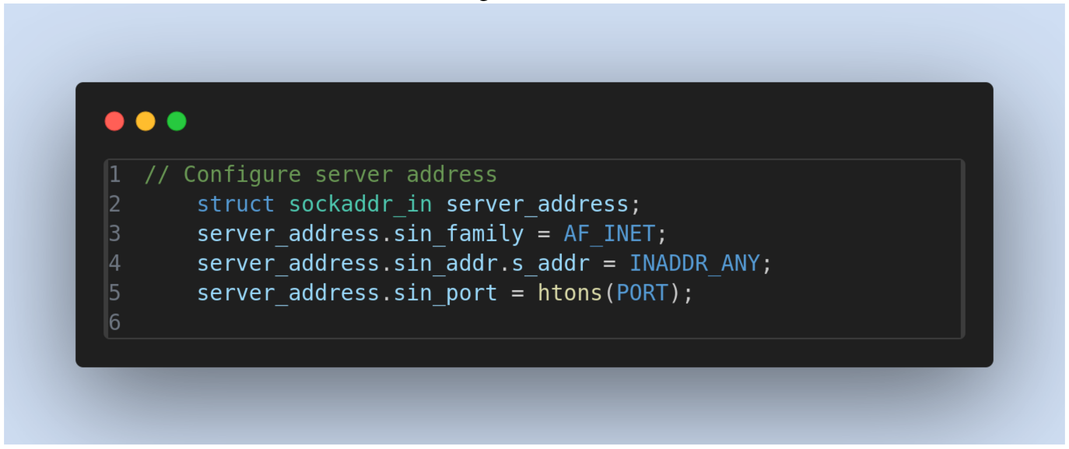
3.1 Server side

The first step is creating a socket and configuring the server address:



```cpp
// Create a socket
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        std::cerr << "Error creating socket" << std::endl;
        return 1;
    }
```

Figure 4: Socket Creation

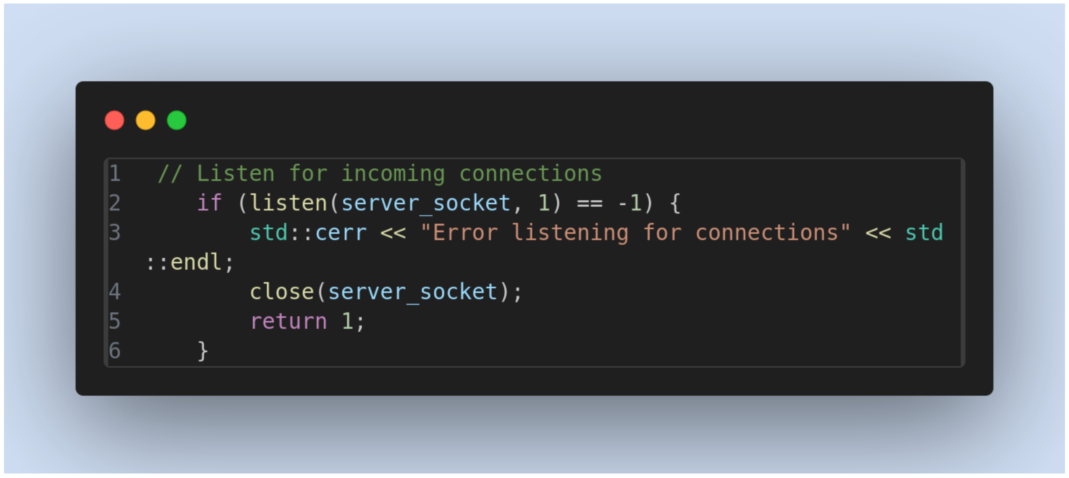


```cpp
// Configure server address
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;
    server_address.sin_port = htons(PORT);

```

Figure 5: Configuring server address

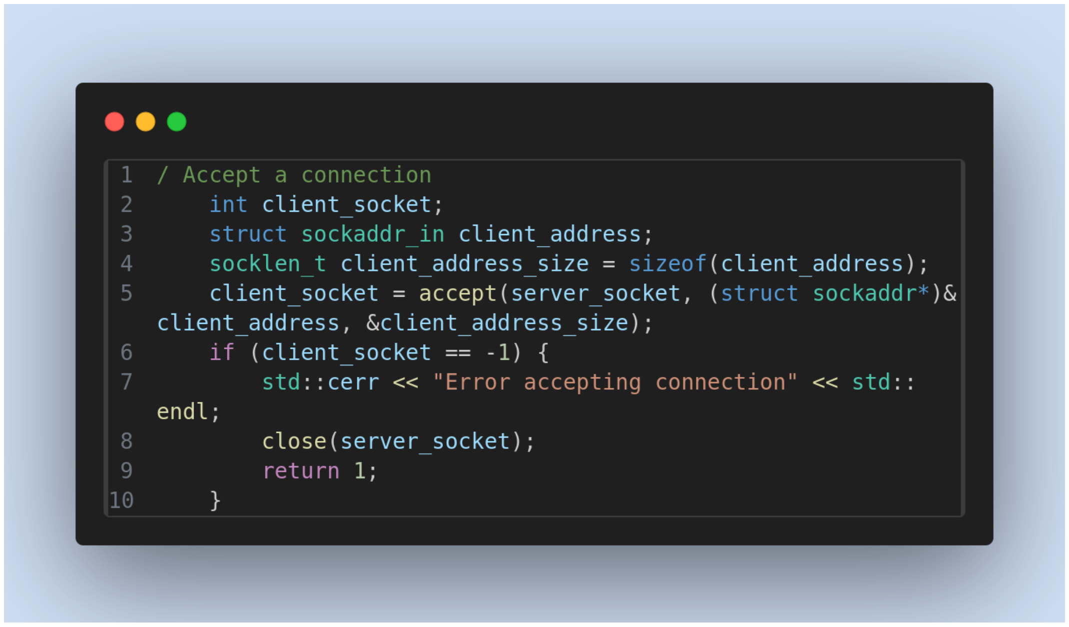
It utilizes the system's command bind() to attach the server socket to the designated server address. This action links the socket to a specific IP address and port number on the server, establishing a communication channel. If the binding process encounters any issues, an error message is displayed to alert the user, and the program is terminated to prevent potential errors.



```cpp
 // Listen for incoming connections
    if (listen(server_socket, 1) == -1) {
        std::cerr << "Error listening for connections" << std
 ::endl;
        close(server_socket);
        return 1;
    }
```

Figure 6: Check for incoming connections

The listen() function acts like a waiter at a restaurant, ready to seat incoming clients (connections). The number you provide (in this case, 1) determines how many clients can wait in line before new ones are rejected. If there's a problem starting this waiting process, an error message is shown, and the program shuts down to avoid issues.



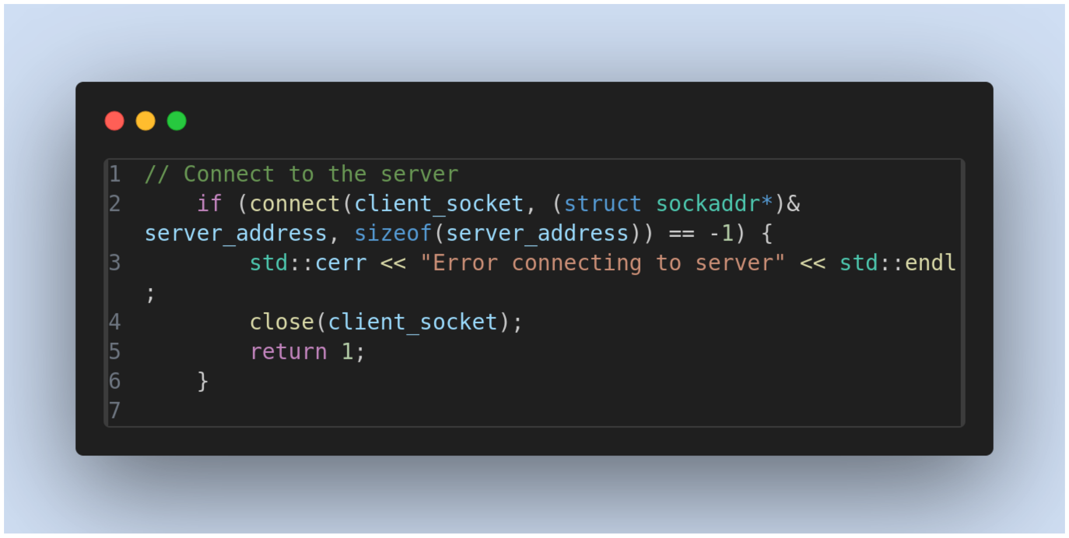
```
/ Accept a connection
    int client_socket;
    struct sockaddr_in client_address;
    socklen_t client_address_size = sizeof(client_address);
    client_socket = accept(server_socket, (struct sockaddr*)&
client_address, &client_address_size);
    if (client_socket == -1) {
        std::cerr << "Error accepting connection" << std::
endl;
        close(server_socket);
        return 1;
    }
```

Figure 7: Accept connection

The accept() function acts like a receptionist for the server. It waits patiently until a client (connection) tries to connect. Once a client connects, accept() hands the server a special code (called a new socket descriptor) to chat with that specific client. Information about the client's address is also kept track of. If the accept() function can't find anyone at the door, it shows an error message and the program stops to avoid problems.

3.2 Client side

The first step of creating socket and configuring address stays the same.

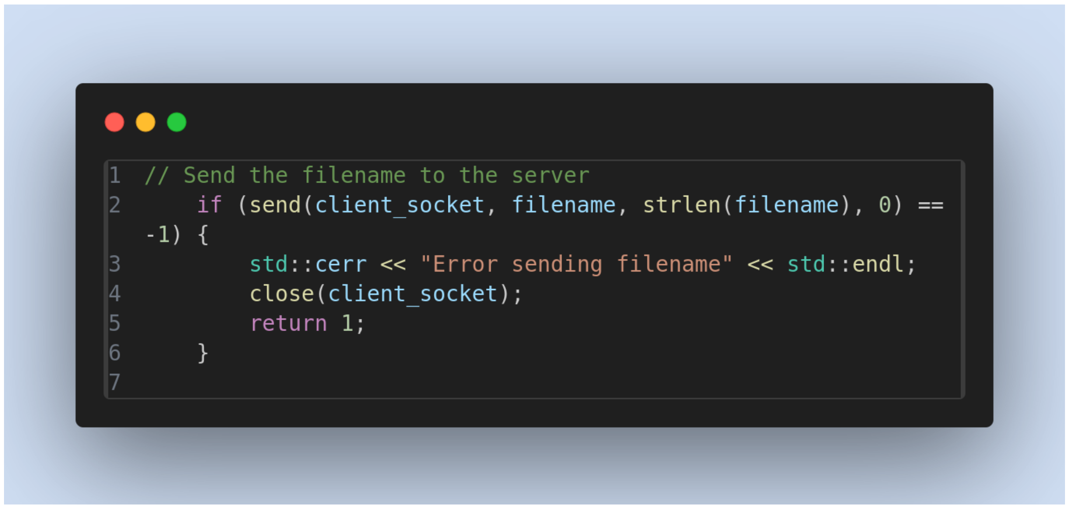


```
// Connect to the server
    if (connect(client_socket, (struct sockaddr*)&
server_address, sizeof(server_address)) == -1) {
        std::cerr << "Error connecting to server" << std::endl
;
        close(client_socket);
        return 1;
    }

```

Figure 8: Connect to the server

This function acts like a dialer on the client's end. It utilizes the client's socket (think of it as a phone line) to attempt establishing a connection with the server at the specified address (like dialing a phone number). If the connection attempt fails, the program displays an error message and exits to prevent unexpected behavior.

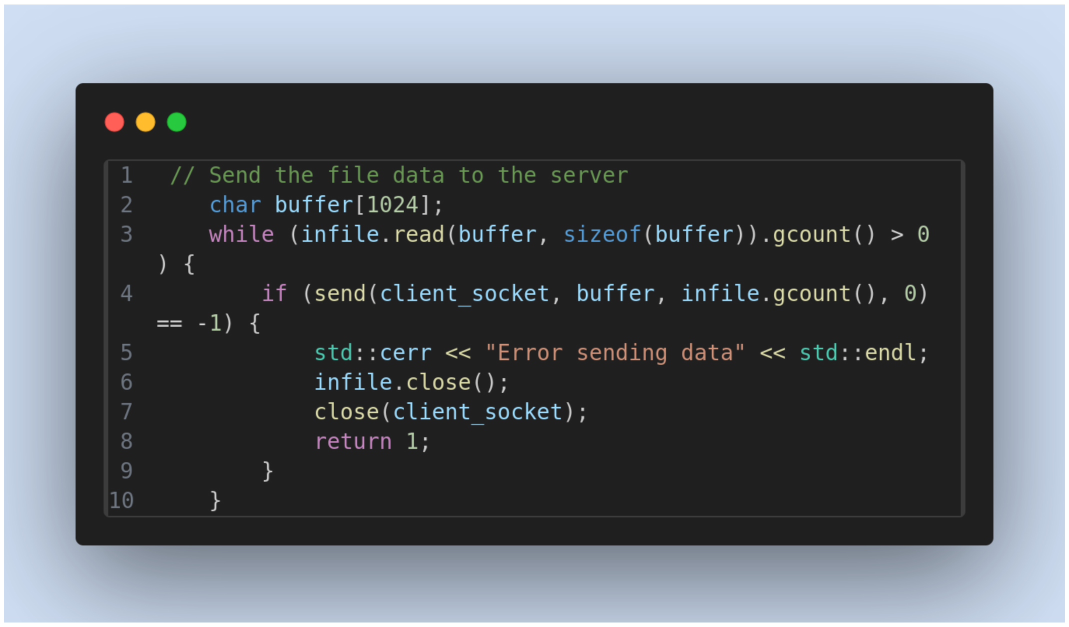


```
// Send the filename to the server
    if (send(client_socket, filename, strlen(filename), 0) ==
-1) {
        std::cerr << "Error sending filename" << std::endl;
        close(client_socket);
        return 1;
    }

```

Figure 9: Send filename to server

Once connected, the client tells the server what file it wants using send(). Send() acts like a messenger for the client. It takes the filename (including a special end marker) and delivers it through the client's socket (like a message sent through a tube). If send() can't deliver the message, an error message is shown, and the program shuts down to avoid issues.



```
 // Send the file data to the server
    char buffer[1024];
    while (infile.read(buffer, sizeof(buffer)).gcount() > 0
) {
        if (send(client_socket, buffer, infile.gcount(), 0)
== -1) {
            std::cerr << "Error sending data" << std::endl;
            infile.close();
            close(client_socket);
            return 1;
        }
    }
```

Figure 10: Send file data to the server

The client chops the file into smaller pieces and sends them to the server. It uses std::ifstream::read() to break the file into manageable chunks of 1024 bytes each (like cutting up a large pizza for delivery). These chunks are then sent to the server using send(), which acts like a delivery truck carrying the file pieces. This process continues until the entire file has been sent (like checking if there's any pizza left). If there's a problem sending a chunk (send() fails), the program displays an error message, cleans up by closing the file and socket, and then exits.