NETWORK FILE SHARING SERVER AND CLIENT

Contents

1. Project Overview

This project implements a basic network file-sharing application using C++ with TCP sockets. The server can list available files, send files to the client, and receive files from the client. The client can request a file list, download files, and upload files to the server. Basic logging is implemented on the server side to track file transfers.

2. Objectives:

Aim- Network File Sharing Over Server and Client.

3.Features:

1. **User authentication:**
   The most common method where users provide a unique username and a password. The server checks these credentials against a database to verify the user.

2. **Multi-Client Support Concurrent Connections:**
   The server can manage multiple client connections simultaneously, thanks to its multi-threaded architecture.

3. **File Transfer Cross-Platform File Sharing:**
   Clients can send files to the server from any supported platform. The server saves these files securely for further use.

4. **Real-Time Messaging Instant Messaging:**
   Clients can send real-time text messages to the server, which can be logged and reviewed later.

5. **Comprehensive Logging Timestamped Logs:**
   All actions, including file transfers and messages, are logged with precise timestamps to ensure traceability.

6. **Cross-Platform Compatibility Operating System Independence:**
   The project is compatible with both Windows and Linux environments, ensuring broad accessibility

4.Functional Requirements:

1> User Authentication:

● The system should authenticate users based on a predefined username and password
● The server verifies the credentials sent by the client and allows access only if they match the stored credentials.

2> File Transfer:
● The system should allow the client to send a file to the server. The file should be transferred in binary mode, and the server should save the received file with the correct filename.

3> Message Exchange:
● The system should allow the client to send textual messages to the server. The server should acknowledge the reception of the message.

4> Logging and Error Handling:

- The server should log connection attempts, file transfers, and any errors that occur during communication.

5. Non-Functional Requirements

1. Security: The system should use basic authentication to ensure that only authorized users can access the file transfer functionality. Sensitive data like usernames and passwords should ideally be encrypted, though this is not implemented in the given program.

2. Scalability: The server should be able to handle multiple clients simultaneously without performance degradation.

3. Reliability: The system should reliably handle file transfers, ensuring that data is not corrupted during transmission.

4. Performance: The file transfer and message exchange operations should be completed in a reasonable amount of time. The use of multi-threading ensures that the server can process multiple requests concurrently.

5. Usability: The client-side interface should be user-friendly, prompting the user for inputs like file paths, messages, and authentication credentials.

6. Maintainability: The code should be modular, with separate functions for handling file transfers, messages, and authentication. This ensures ease of maintenance and future enhancements.

6. Technologies Used

1. C++ Programming Language:

- The program is written in C++, utilizing features such as multi-threading, file handling, and socket programming.

2. POSIX Sockets API:

- **The program uses the POSIX sockets API for creating and managing network connections, allowing communication between the server and clients.**

3. Multi-threading:

- **The std::thread library is used for handling multiple client connections simultaneously on the server side.**

4. Networking Protocols:

- **The program uses TCP/IP for reliable communication between the client and server, ensuring that all data is delivered accurately and in order.**

5. Standard Libraries:

- **Libraries such as , , and are used for input/output operations, file handling, and string manipulation.**

6. Linux/Unix System Calls:

- **System calls like socket(), bind(), listen(), accept(), recv(), and send() are used to manage the network communication.**

7. Requirement Gathering and Analysis

● Objective: Identify and document the functional and non-functional requirements.

● Activities:

- Discuss and document the user needs (e.g., user authentication, file transfer, multi-client support).
- Define the system's scope, goals, and objectives.
- Identify key stakeholders and their expectations.

● Deliverables: Requirement Specification Document, Use Case Diagrams.

8. System Design

● Objective: Create a blueprint for how the system will be implemented.

● Activities:

1. Architectural Design: Define the overall architecture, including client-server communication, multi-threading, and data flow.
2. Module Design: Break down the system into modules (e.g., authentication module, file transfer module).
3. Database Design: (Optional) If the project scales, design a database schema for storing user credentials, logs, etc.

- **Deliverables:** System Architecture Diagram, Module Design Documents, Database Schema (if applicable).

3. Implementation

- **Objective:** Develop the software according to the design specifications.
- **Activities:**
  a. Set up the development environment.
  b. Code the client and server components.
  c. Implement core functionalities like authentication, file transfer, and message handling.
  d. Write unit tests for individual modules.
- **Deliverables:** Source Code, Unit Test Cases.

4. Testing

- **Objective:** Ensure that the system works as intended and meets the requirements.
- **Activities:**
  - Unit Testing: Test individual modules to ensure they work as expected.
  - Integration Testing: Test the interaction between different modules (e.g., authentication followed by file transfer).
  - System Testing: Perform end-to-end testing of the entire system, including stress testing with multiple clients.
  - User Acceptance Testing (UAT): Verify that the system meets user expectations.
- **Deliverables:** Test Plan, Test Cases, Bug Reports, Test Summary Report.

5. Deployment

- **Objective:** Deploy the system in a real-world environment.
- **Activities:**
  - Set up the server environment for hosting the application.
  - Install necessary software dependencies and configurations.
  - Deploy the client application to end users.
  - Ensure security measures are in place (e.g., firewall, encryption).
- **Deliverables:** Deployment Plan, Installed and Configured System, Deployment Documentation.

6. Maintenance and Support

- **Objective:** Ensure the system remains functional, secure, and up-to-date.

- **Activities:**
  - Monitor the system for performance issues or bugs.
  - Provide technical support and updates as needed.
  - Implement patches or updates to address security vulnerabilities.
  - Gather feedback for future improvements.
- **Deliverables:** Maintenance Logs, Update Patches, User Support Documentation.

7. Documentation and Training

- **Objective:** Provide comprehensive documentation and training for users and developers.
- **Activities:**
  - Create user manuals and technical documentation. o Provide training sessions for end users and administrators.
  - Document code and system architecture for future reference.
- **Deliverables**: User Manuals, Technical Documentation, Training Materials.

5. System Design:

Overview of the components

1. Architecture Overview

- **Client-Server Model:** The system follows a client-server architecture where multiple clients can connect to a central server to upload, download, and list files.

2. Data Flow Diagram

Client Authentication:

- The client sends a password to the server for authentication.
- The server verifies the password and responds with an acknowledgment

Low-Level Design (LLD)

1. Detailed Component Design

1. Client Application:
   - Modules:
     - Network Module: Handles socket creation, connection, and communication.

        - Authentication Module: Manages sending and receiving authentication data.
        - File Transfer Module: Handles file upload and download operations.
        - User Interface Module: Provides command-line interface for user interaction.

2. Server Application:
    - Modules:
        - Network Module: Handles socket creation, binding, listening, and accepting connections.
        - Authentication Module: Verifies client credentials.
        - File Management Module: Manages file listings, uploads, and downloads.
        - Security Module: Manages SSL/TLS setup and encryption.

2. Class Diagrams

1. Client Classes:

- Client:
    - Methods: `connect()`, `authenticate()`, `requestFileList()`, `downloadFile()`, `uploadFile()`
- NetworkManager:
    - Methods: `createSocket()`, `connectToServer()`, `sendData()`, `receiveData()`
- FileManager:
    - Methods: `listFiles()`, `downloadFile()`, `uploadFile()`
- UserInterface:
    - Methods: `getCommand()`, `displayFileList()`, `displayMessage()`

2. Server Classes:

- Server
    - Methods: `start()`, `acceptClient()`, `authenticateClient()`, `handleRequest()`

- NetworkManager
  - Methods: `createSocket()`, `bindSocket()`, `listenForConnections()`, `acceptConnection()`, `sendData()`, `receiveData()`
- FileManager
  - Methods: `listFiles()`, `sendFile()`, `receiveFile()`
- SecurityManager
  - Methods: `initializeSSL()`, `createContext()`, `configureContext()`

3. Sequence Diagrams

1. Client Authentication Sequence:
   - Client -> Server: `connect()`
   - Client -> Server: `authenticate(password)`
   - Server -> Client: `authenticationResult()`
2. File Listing Sequence:
   - Client -> Server: `requestFileList()`
   - Server -> Client: `fileList()`
3. File Download Sequence:
   - Client -> Server: `downloadFile(fileName)`
   - Server -> Client: `sendFileData(fileData)`
4. File Upload Sequence:
   - Client -> Server: `uploadFile(fileName)`
   - Client -> Server: `sendFileData(fileData)`
   - Server -> Client: `uploadConfirmation()`

9.Flow Diagram For Server Client Architecture

The server application diagram is designed to represent the different modules and their interactions within the server application. Each module has a specific role in handling client requests, managing files, and ensuring secure communication.

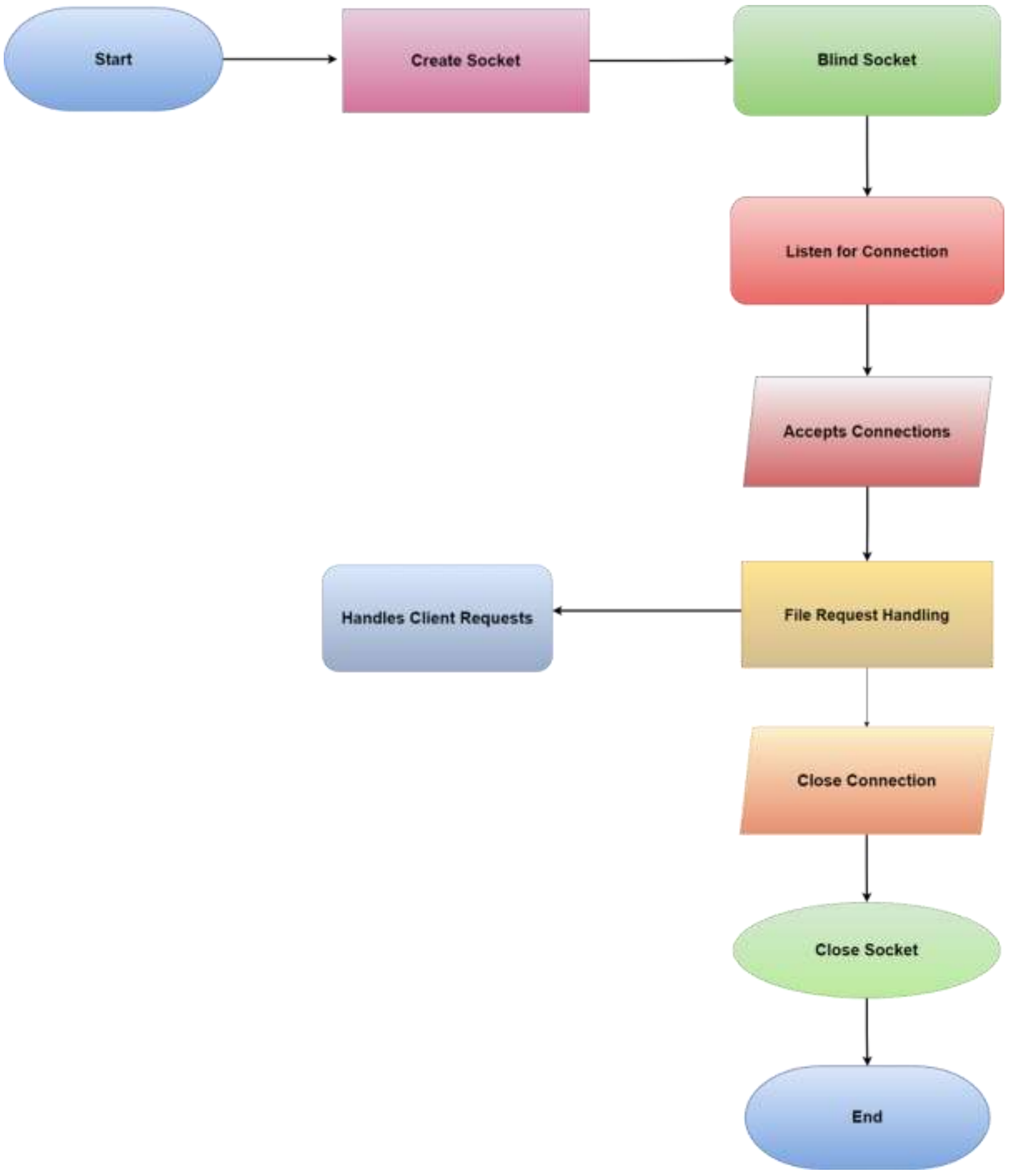
CLIENT

This is the main class that orchestrates the operations of the client application. It utilizes other modules to establish a connection to the server, authenticate, request file listings, and manage file transfers (upload/download).

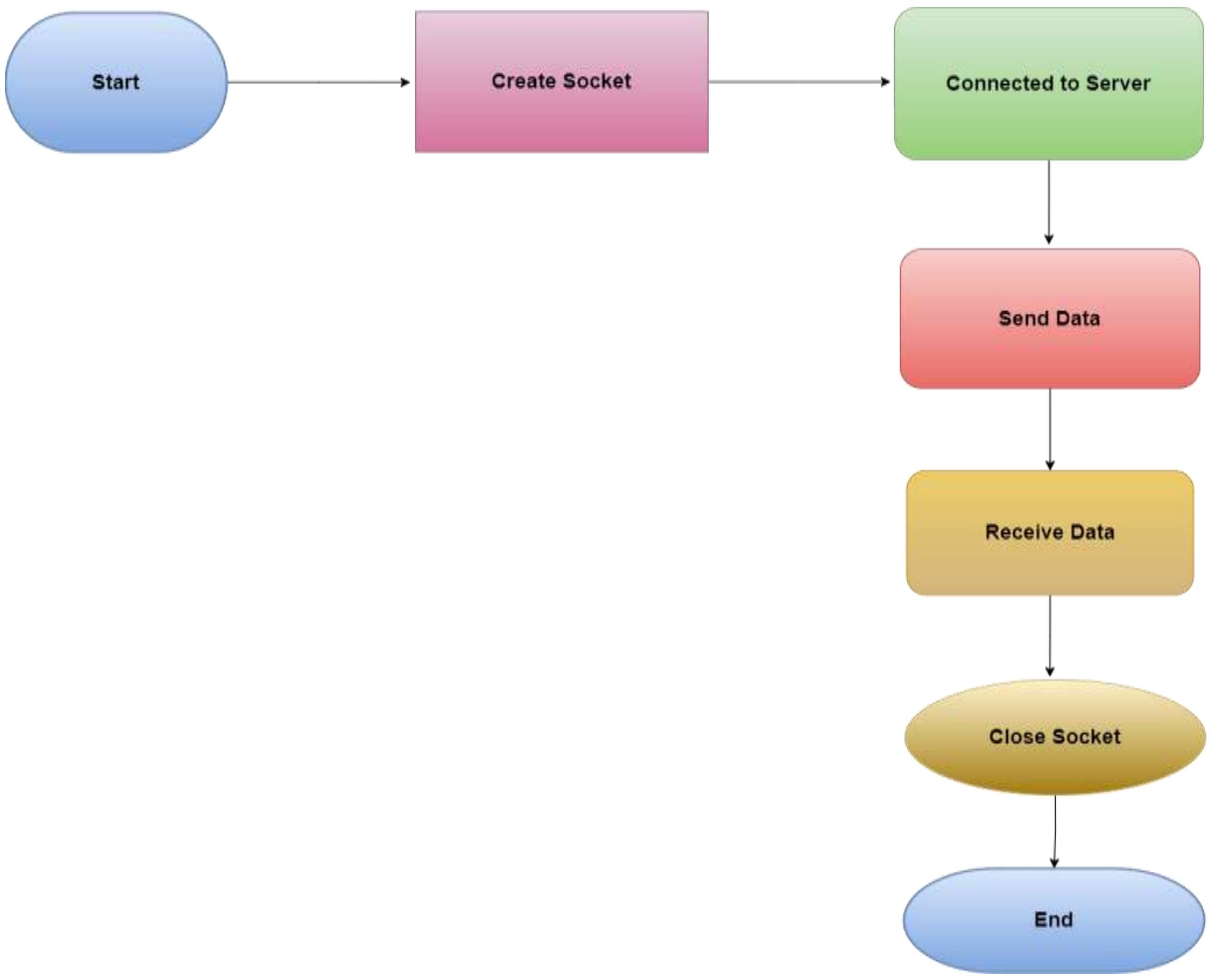
The `Client` class uses the `NetworkManager` to send authentication credentials (e.g., password) to the server.

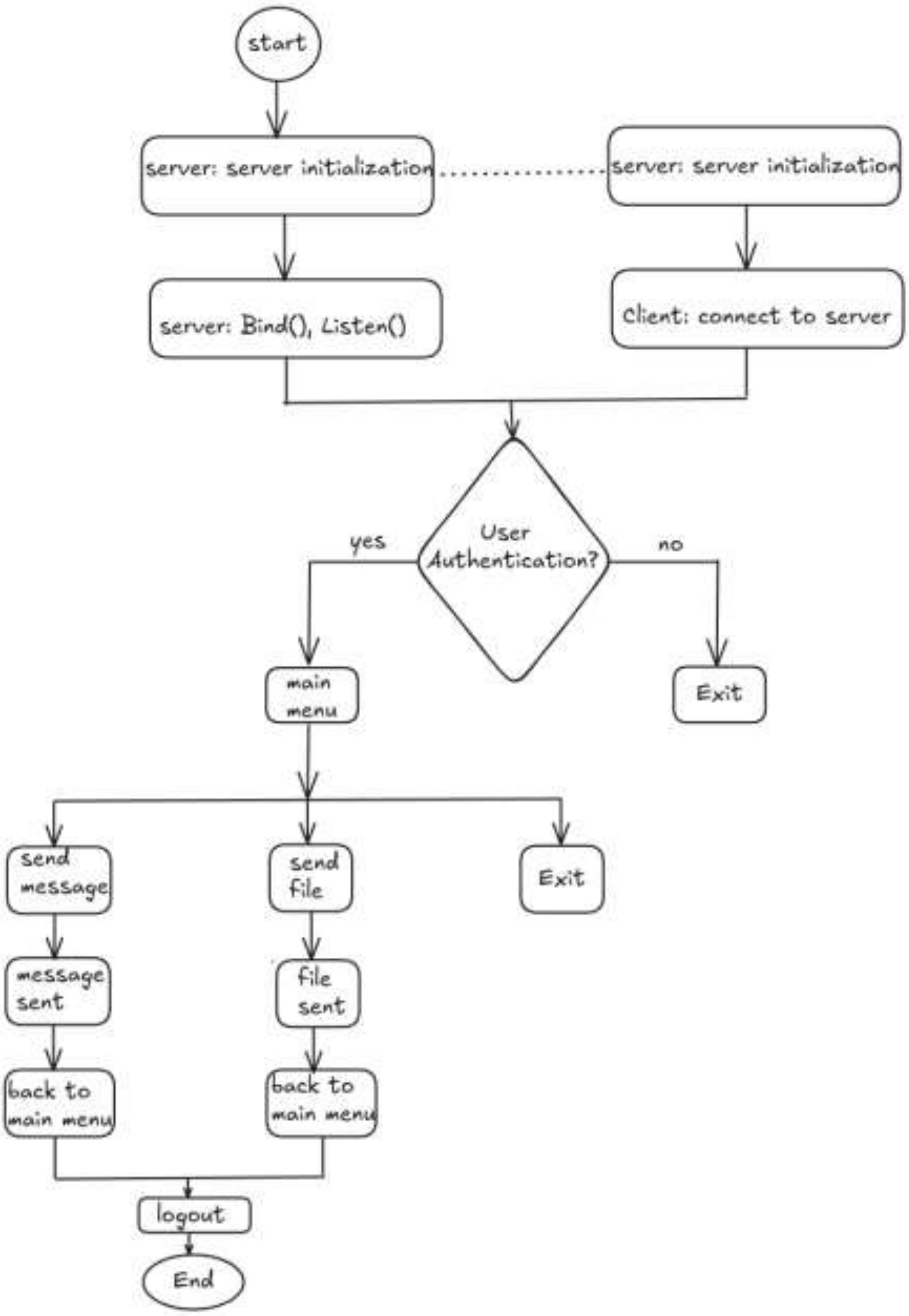
The server verifies the credentials and responds. The `NetworkManager` receives this response and passes it back to the `Client.`

Server Flow Diagram



Client Flow Diagram




start
server: server initialization
server: server initialization
server: Bind(), Listen()
Client: connect to server
User Authentication?
yes
no
main menu
Exit
send message
send file
Exit
message sent
file sent
back to main menu
back to main menu
logout
End

Make File

```
# Define compiler and flags

CXX = g++

CXXFLAGS = -std=c++11 -Wall -Wextra

# Define targets and their dependencies

all: client server

client: client.o

	$(CXX) $(CXXFLAGS) -o client client.o

server: server.o

	$(CXX) $(CXXFLAGS) -o server server.o

# Compile the client source file

client.o: client.cpp

	$(CXX) $(CXXFLAGS) -c client.cpp


# Compile the server source file

server.o: server.cpp

	$(CXX) $(CXXFLAGS) -c server.cpp


# Clean up build files

clean:

	rm -f *.o client server


.PHONY: all clean
```

SERVER CODE

```
#include <iostream>

#include <fstream>

#include <cstring>

#include <thread>

#include <vector>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>


const int PORT = 8086;

const int BUFFER_SIZE = 1024;

const std::string USERNAME = "admin"; // Example username

const std::string PASSWORD = "password"; // Example password


void handleClient(int clientSocket, std::string clientIP) {

    char buffer[BUFFER_SIZE];

    int bytesRead;


    // Authenticate client

    std::string receivedUsername;

    std::string receivedPassword;


    // Receive username

    bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);

    if (bytesRead <= 0) {
```

```
    std::cerr << "[" << clientIP << "] Error receiving username." << std::endl;
    close(clientSocket);
    return;
}
buffer[bytesRead] = '\0';
receivedUsername = buffer;

// Receive password
bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);
if (bytesRead <= 0) {
    std::cerr << "[" << clientIP << "] Error receiving password." << std::endl;
    close(clientSocket);
    return;
}
buffer[bytesRead] = '\0';
receivedPassword = buffer;

// Check credentials
if (receivedUsername != USERNAME || receivedPassword != PASSWORD) {
    std::cerr << "[" << clientIP << "] Authentication failed." << std::endl;
    close(clientSocket);
    return;
}

// Notify successful authentication
send(clientSocket, "AUTH_OK", strlen("AUTH_OK"), 0);
```

```
    // Receive filename
    bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);
    if (bytesRead <= 0) {
        std::cerr << "[" << clientIP << "] Error receiving filename." << std::endl;
        close(clientSocket);
        return;
    }
    buffer[bytesRead] = '\0';
    std::string filename(buffer); // Store the filename received from the client

    // Log the filename and start receiving the file
    std::cout << "[" << clientIP << "] Receiving file (" << filename << ") from client..." <<
std::endl;

    while ((bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0)) > 0) {
        buffer[bytesRead] = '\0';

        if (strcmp(buffer, "FILE_START") == 0) {
            std::ofstream outFile(filename, std::ios::binary);
            if (!outFile) {
                std::cerr << "Error opening file for writing." << std::endl;
                close(clientSocket);
                return;
            }
            std::cout << "[" << clientIP << "] Writing to file: " << filename << std::endl;
            while ((bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0)) > 0) {
                if (strcmp(buffer, "FILE_END") == 0) break;
                outFile.write(buffer, bytesRead);
            }
            outFile.close();
            std::cout << "[" << clientIP << "] File (" << filename << ") received successfully." <<
std::endl;
            send(clientSocket, "ACK: File received", strlen("ACK: File received"), 0);
        }
 else {
            std::cout << "[" << clientIP << "] Message from client: " << buffer << std::endl;
            send(clientSocket, "ACK: Message received", strlen("ACK: Message received"), 0);
        }

    }

    if (bytesRead < 0) {
        std::cerr << "[" << clientIP << "] Error receiving data." << std::endl;
    }

    close(clientSocket);
}

int main() {
    int serverSocket, clientSocket;
    struct sockaddr_in serverAddr, clientAddr;

    socklen_t addrSize = sizeof(clientAddr);
    std::vector<std::thread> threads;
```

```
// Create socket
serverSocket = socket(AF_INET, SOCK_STREAM, 0);
if (serverSocket < 0) {
    std::cerr << "Socket creation error." << std::endl;
    return -1;
}

// Bind socket
serverAddr.sin_family = AF_INET;

serverAddr.sin_addr.s_addr = INADDR_ANY;
serverAddr.sin_port = htons(PORT);

if (bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) < 0) {
    std::cerr << "Bind error." << std::endl;
    return -1;
}

// Listen for connections

if (listen(serverSocket, 5) < 0) {
    std::cerr << "Listen error." << std::endl;
    return -1;
}

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

```
}

    while (true) {
        clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddr, &addrSize);

     if (clientSocket < 0) {
            std::cerr << "Accept error." << std::endl;
            continue;
        }

        std::string clientIP = inet_ntoa(clientAddr.sin_addr);
        std::cout << "Client connected from IP: " << clientIP << std::endl;

        // Handle client in a new thread
        threads.emplace_back(handleClient, clientSocket, clientIP);
    }


    // Join threads (this will never actually happen in this example)
    for (auto& t : threads) {
        t.join();

    }

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

CLIENT CODE

```
#include <iostream>

#include <fstream>

#include <cstring>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>


const int PORT = 8086;

const int BUFFER_SIZE = 1024;


void sendFile(int socket, const std::string& filePath) {

  std::ifstream file(filePath, std::ios::binary);

  if (!file) {

    std::cerr << "Error opening file." << std::endl;

    return;

  }


  // Extract the filename from the file path

  std::string filename = filePath.substr(filePath.find_last_of("/\\") + 1);


  // Send the filename

  send(socket, filename.c_str(), filename.size(), 0);

  usleep(100); // Small delay to ensure filename is sent before FILE_START


  char buffer[BUFFER_SIZE];
```

```
    // Notify server about file transfer
    send(socket, "FILE_START", strlen("FILE_START"), 0);

    // Send file content
    while (file.read(buffer, BUFFER_SIZE)) {
        send(socket, buffer, file.gcount(), 0);
    }
    send(socket, buffer, file.gcount(), 0);  // Send remaining bytes

    // Notify server end of file transfer
    send(socket, "FILE_END", strlen("FILE_END"), 0);

    // Receive acknowledgment
    int bytesRead = recv(socket, buffer, BUFFER_SIZE, 0);
    if (bytesRead > 0) {
        buffer[bytesRead] = '\0';
        std::cout << "Server: " << buffer << std::endl;
    }

    std::cout << "File sent successfully." << std::endl;
}

void sendMessage(int socket, const std::string& message) {
    send(socket, message.c_str(), message.size(), 0);
    std::cout << "Message sent: " << message << std::endl;

    char buffer[BUFFER_SIZE];
    int bytesRead = recv(socket, buffer, BUFFER_SIZE, 0);
    if (bytesRead > 0) {
        buffer[bytesRead] = '\0';
        std::cout << "Server: " << buffer << std::endl;
    }
}

int main() {
    int socketFd;
    struct sockaddr_in serverAddr;
    std::string input;

    // Create socket
    socketFd = socket(AF_INET, SOCK_STREAM, 0);
    if (socketFd < 0) {
        std::cerr << "Socket creation error." << std::endl;
        return -1;
    }

    // Connect to server
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    if (inet_pton(AF_INET, "127.0.0.1", &serverAddr.sin_addr) <= 0) { // Use appropriate IP
address
        std::cerr << "Invalid address/Address not supported." << std::endl;
        return -1;
    }
```

```
if (connect(socketFd, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) < 0) {
    std::cerr << "Connection error." << std::endl;
    return -1;
}

// Authentication
std::string username, password;
std::cout << "Enter username: ";
std::getline(std::cin, username);

std::cout << "Enter password: ";
std::getline(std::cin, password);

// Send username and password
send(socketFd, username.c_str(), username.size(), 0);
usleep(100); // Small delay to ensure username is sent before password
send(socketFd, password.c_str(), password.size(), 0);

// Receive authentication response
char buffer[BUFFER_SIZE];
int bytesRead = recv(socketFd, buffer, BUFFER_SIZE, 0);

if (bytesRead > 0) {
    buffer[bytesRead] = '\0';

if (strcmp(buffer, "AUTH_OK") != 0) {
```

```
            std::cerr << "Authentication failed." << std::endl;

            close(socketFd);

            return -1;

        }

    }



else {

        std::cerr << "Error receiving authentication response." << std::endl;

        close(socketFd);

        return -1;

    }


    // Loop for sending messages or files

    while (true) {

        std::cout << "Enter 'file' to send a file, 'message' to send a message, or 'exit' to quit: ";


        std::cin >> input;

        std::cin.ignore();


  // Ignore the newline character after input


        if (input == "file") {

            std::string filePath;

            std::cout << "Enter file path: ";


           std::getline(std::cin, filePath);
```

```
            sendFile(socketFd, filePath);
        }


 else if (input == "message") {
            std::string message;
            std::cout << "Enter message: ";
            std::getline(std::cin, message);
            sendMessage(socketFd, message);
        }
 else if (input == "exit") {
            std::cout << "Exiting..." << std::endl;
            break;
        }
else {
            std::cerr << "Invalid option." << std::endl;
        }
    }

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

BUG Tracker Report

Title: Bind Error: Server Fails to Bind to Port

Description:

The server application occasionally fails to bind to the specified port, resulting in a "Bind error" message. This issue prevents the server from starting and accepting connections, leading to downtime and unavailability of the network file-sharing service.

Steps to Reproduce:

- Start the server application.
- Observe the server output for a "Bind error" message.

Expected Behavior:

The server should bind to the specified port and start listening for incoming connections without any errors.

Actual Behavior:

The server sometimes fails to bind to the port, displaying a "Bind error" message and preventing the application from starting.

Error Log:

Example error message: Bind error.

Fix Description:

Port Availability Check: Implement a check to ensure the port is available before attempting to bind.

Retry Mechanism: Implement a retry mechanism that attempts to bind to the port multiple times before failing.

Error Logging: Improve error logging to provide more detailed information about the bind error, helping to diagnose the issue more effectively.

BUG ID: 002

Title: Server Delayed Response

Description:

The server sometimes takes an unexpectedly long time to respond after a file transfer is initiated by the client. This delay may affect the user experience, especially in real-time applications where quick acknowledgment is critical.

Steps to Reproduce:

- Start the server by running ./server.
- Start the client by running ./client.
- Send a file from the client to the server.
- Observe the delay before the server acknowledges the file transfer completion.
- Observe the output in the client terminal.

Expected Result: The server should respond immediately after the file has been completely received.

Actual Behavior: The server delays its response by a few seconds, leading to potential performance issues.

Root Cause Analysis: The cause of the delay is currently under investigation.

Solution Implemented: Not yet implemented.

Testing: No testing has been conducted as the issue is still open.

BUG ID: 003

Title: Server Not Exiting Properly

Description:
The server does not exit gracefully upon receiving termination signals or when it encounters an error. This issue can lead to resource leaks and potential instability in the system.

Steps to Reproduce:

- Start the server application.
- Attempt to terminate the server using a signal (e.g., Ctrl+C) or after a fatal error.
- Observe that the server does not exit cleanly.

Expected Behavior:

The server should release all resources and exit gracefully upon receiving a termination signal or after encountering a critical error.

Actual Behavior:

The server either hangs or leaves resources (like sockets) open, causing instability.

Root Cause Analysis:

The issue appears to be related to improper handling of termination signals and resource cleanup.

Solution Implemented:

Not yet implemented.

Testing:

No testing has been conducted as the issue is still open.

BUG ID : 004

Title: File Not Received at Server End Despite Successful Transfer

Description:

The server does not correctly receive files, even though the client indicates that the file has been sent successfully. This leads to missing files on the server side, causing disruptions in file availability and integrity.

Steps to Reproduce:

- Start the server application.
- Connect a client to the server.
- Send a file from the client to the server.
- Observe the server to check if the file has been received.

Expected Behavior:

The server should receive and store the file correctly after it has been sent from the client.

Actual Behavior:

The client confirms the file has been sent, but the server does not receive or store the file as expected.

Root Cause Analysis:

Buffer size mismatch and improper handling of file transfer protocol may cause incomplete file transfer.

Solution Implemented:
The server code has been modified to ensure proper file handling and dynamic buffer allocation. Additionally, the contents of the file are now displayed on the server's terminal to verify successful reception.

References for the Network File Sharing Server & Client Project

Special thanks to Shwetank Sir. This project has been developed under his expert guidance and training. I am truly grateful for his support and mentorship.

- C++ Standard Library Documentation
- POSIX Socket Programming
- Logging Library
- Encryption and Security
- Tutorials & Examples
- C++ Socket Programming Example

THANK YOU

DIKSHA KUMARI