Project Report

Network File Sharing Server & Client (LSP)

Submitted by:

Name	MEHRAJ RUSTUM
Registration Number	2241018194
Branch	CSIT
Batch	13

Objective

To develop a networked file-sharing application based on **server-client architecture**, enabling secure file transfers using **socket programming**, **authentication**, and **encryption** in C++.

Overview:

This project demonstrates a **client-server communication system** using **POSIX sockets**.
It allows the user to:

- Establish socket-based communication between client and server.
- List, upload, and download files.
- Transfer files securely using simple XOR-based encryption.
- Implement basic authentication for security.

Technologies Used:

- **Language:** C++
- **Libraries:** POSIX Sockets (sys/socket.h, arpa/inet.h, unistd.h)
- **Platform:** Linux (Ubuntu/Kali/WSL)

- **Encryption:** Simple XOR-based symmetric key encryption

Code Section

SERVER.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <sys/stat.h>
#include <dirent.h>
#include <algorithm>

//CONFIGURATION
#define PORT 65432
#define BUFFER_SIZE 1024
const std::string SHARED_DIR = "server_files/";
const std::string ENCRYPTION_KEY = "a_very_simple_shared_key";
const std::string SERVER_PASSWORD = "Mehraj123";

void handle_client(int client_socket);
void send_file(int client_socket, const std::string& filename);
void receive_file(int client_socket, const std::string& filename);
std::string list_files();
void xor_encrypt_decrypt(char* data, size_t len, const std::string& key);

int main() {
    // 1. Create socket file descriptor
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == 0) {
        perror("socket failed");
        return 1;
    }

    // 2. Bind the socket
    struct sockaddr_in address;
    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");
```

```
        return 1;
    }

    // 3. Listen for connections
    if (listen(server_fd, 3) < 0) {
        perror("listen failed");
        return 1;
    }

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

    // 4. Accept loop
    while (true) {
        int new_socket;
        socklen_t addrlen = sizeof(address);

        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
&addrlen)) < 0) {
            perror("accept failed");
            continue;
        }

        std::cout << "New client connected. Waiting for authentication..."
<< std::endl;
        // Handle this client in a single thread
        handle_client(new_socket);
    }

    close(server_fd);
    return 0;
}


void xor_encrypt_decrypt(char* data, size_t len, const std::string& key) {
    if (key.empty()) return;
    for (size_t i = 0; i < len; ++i) {
        data[i] = data[i] ^ key[i % key.length()];
    }
}


void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE] = {0};
    int valread;

    //  AUTHENTICATION PHASE
    valread = read(client_socket, buffer, BUFFER_SIZE);
    if (valread <= 0) {
        std::cerr << "Client disconnected before auth." << std::endl;
        close(client_socket);
```

```
        return;
    }

    // Decrypt the password
    xor_encrypt_decrypt(buffer, valread, ENCRYPTION_KEY);
    std::string received_password(buffer, valread);

    char response_buffer[32];
    if (received_password == SERVER_PASSWORD) {
        std::cout << "Client authenticated successfully." << std::endl;
        strncpy(response_buffer, "AUTH_OK", sizeof(response_buffer));
    } else {
        std::cerr << "Client failed authentication." << std::endl;
        strncpy(response_buffer, "AUTH_FAIL", sizeof(response_buffer));
    }

    // Encrypt and send the response
    xor_encrypt_decrypt(response_buffer, strlen(response_buffer),
ENCRYPTION_KEY);
    send(client_socket, response_buffer, strlen(response_buffer), 0);

    // If auth failed, close connection
    if (received_password != SERVER_PASSWORD) {
        close(client_socket);
        return;
    }

    while ( (valread = read(client_socket, buffer, BUFFER_SIZE)) > 0) {

        // Decrypt the command
        xor_encrypt_decrypt(buffer, valread, ENCRYPTION_KEY);
        std::string command_line(buffer, valread);

        // Trim newline characters
        command_line.erase(command_line.find_last_not_of(" \n\r\t")+1);

        std::cout << "Received command: [" << command_line << "]" <<
std::endl;

        std::string command, filename;
        size_t space_pos = command_line.find(' ');
        if (space_pos != std::string::npos) {
            command = command_line.substr(0, space_pos);
            filename = command_line.substr(space_pos + 1);
        } else {
            command = command_line;
        }

        // --- COMMAND LOGIC ---
```

```
        if (command == "LIST") {
            std::string file_list = list_files();
            // Encrypt and send file list
            xor_encrypt_decrypt(const_cast<char*>(file_list.c_str()),
file_list.length(), ENCRYPTION_KEY);
            send(client_socket, file_list.c_str(), file_list.length(), 0);

        } else if (command == "GET" && !filename.empty()) {
            send_file(client_socket, filename);

        } else if (command == "PUT" && !filename.empty()) {
            receive_file(client_socket, filename);

        } else if (command == "QUIT") {
            std::cout << "Client sent QUIT. Closing connection." <<
std::endl;
            break; // Exit the loop
        }

        // Clear the buffer for the next read
        memset(buffer, 0, BUFFER_SIZE);
    }

    std::cout << "Client disconnected. Closing socket." << std::endl;
    close(client_socket); // Close the connection socket
}

// SEND FILE
void send_file(int client_socket, const std::string& filename) {
    std::string filepath = SHARED_DIR + filename;

    std::ifstream file(filepath, std::ios::in | std::ios::binary);
    if (!file.is_open()) {
        std::string msg = "FILE_NOT_FOUND";
        long file_size = -1;

        // Encrypt and send error size
        xor_encrypt_decrypt(reinterpret_cast<char*>(&file_size),
sizeof(long), ENCRYPTION_KEY);
        send(client_socket, &file_size, sizeof(long), 0);

        // Encrypt and send error message
        xor_encrypt_decrypt(const_cast<char*>(msg.c_str()), msg.length(),
ENCRYPTION_KEY);
        send(client_socket, msg.c_str(), msg.length(), 0);

        std::cerr << "File not found: " << filename << std::endl;
        return;
    }
```

```
    // 1. Send file size (Crucial for receiver)
    struct stat file_status;
    stat(filepath.c_str(), &file_status);
    long file_size = file_status.st_size;

    // Encrypt and send file size
    long encrypted_size = file_size;
    xor_encrypt_decrypt(reinterpret_cast<char*>(&encrypted_size),
sizeof(long), ENCRYPTION_KEY);
    send(client_socket, &encrypted_size, sizeof(long), 0);

    // 2. Send file chunks
    char buffer[BUFFER_SIZE];
    while (file.read(buffer, BUFFER_SIZE)) {
        // Encrypt chunk
        xor_encrypt_decrypt(buffer, BUFFER_SIZE, ENCRYPTION_KEY);
        send(client_socket, buffer, BUFFER_SIZE, 0);
    }
    // Send the last partial chunk
    if (file.gcount() > 0) {
        // Encrypt last chunk
        xor_encrypt_decrypt(buffer, file.gcount(), ENCRYPTION_KEY);
        send(client_socket, buffer, file.gcount(), 0);
    }

    file.close();
    std::cout << "Successfully sent file: " << filename << std::endl;
}

// RECEIVE FILE (MODIFIED)
void receive_file(int client_socket, const std::string& filename) {
    std::string filepath = SHARED_DIR + filename;

    // 1. Receive file size
    long file_size;
    if (read(client_socket, &file_size, sizeof(long)) <= 0) {
        std::cerr << "Error receiving file size." << std::endl;
        return;
    }
    // Decrypt file size
    xor_encrypt_decrypt(reinterpret_cast<char*>(&file_size), sizeof(long),
ENCRYPTION_KEY);

    // 2. Open file for writing
    std::ofstream file(filepath, std::ios::out | std::ios::binary);
    if (!file.is_open()) {
        std::cerr << "Error opening file for writing: " << filename <<
std::endl;
```

```
        return;
    }

    // 3. Receive file chunks
    char buffer[BUFFER_SIZE];
    long total_received = 0;

    while (total_received < file_size) {
        long bytes_to_read = std::min((long)BUFFER_SIZE, file_size -
total_received);

        int bytes_received = recv(client_socket, buffer, bytes_to_read,
0);

        if (bytes_received <= 0) {
            std::cerr << "Connection closed prematurely or error." <<
std::endl;
            break;
        }

        // Decrypt chunk
        xor_encrypt_decrypt(buffer, bytes_received, ENCRYPTION_KEY);

        file.write(buffer, bytes_received);
        total_received += bytes_received;
    }

    file.close();
    std::cout << "Successfully received file: " << filename << " (" <<
total_received << " bytes)" << std::endl;
}

// FILE LISTING
std::string list_files() {
    DIR *dir;
    struct dirent *ent;
    std::string list = "";

    if ((dir = opendir(SHARED_DIR.c_str())) != NULL) {
        while ((ent = readdir(dir)) != NULL) {
            std::string name(ent->d_name);
            if (name != "." && name != "..") {
                list += name + "\n";
            }
        }
        closedir(dir);
    } else {
        list = "ERROR: Could not open shared directory.";
    }
```

```
    return list;
}
```

CLIENT.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/stat.h>
#include <algorithm>

#define SERVER_IP "127.0.0.1"
#define PORT 65432
#define BUFFER_SIZE 1024

const std::string CLIENT_DIR = "client_downloads/";
const std::string ENCRYPTION_KEY = "a_very_simple_shared_key";

void send_file(int sock, const std::string& filename);
void receive_file(int sock, const std::string& filename);
void xor_encrypt_decrypt(char* data, size_t len, const std::string& key);

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

    // 2. Setup server address structure
    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, SERVER_IP, &serv_addr.sin_addr) <= 0) {
        std::cerr << "Invalid address/Address not supported." <<
std::endl;
        return -1;
    }

    // 3. Connect to the server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) <
0) {
```

```
        std::cerr << "Connection Failed. Is the server running?" <<
std::endl;
        return -1;
    }

    std::cout << "Connected to server." << std::endl;

    char auth_buffer[BUFFER_SIZE] = {0};

    // Prompt user for password
    std::string user_entered_password;
    std::cout << "Please enter the server password: ";
    std::getline(std::cin, user_entered_password);
    strncpy(auth_buffer, user_entered_password.c_str(), BUFFER_SIZE);
    auth_buffer[BUFFER_SIZE - 1] = '\0';

    xor_encrypt_decrypt(auth_buffer, user_entered_password.length(),
ENCRYPTION_KEY);
    send(sock, auth_buffer, user_entered_password.length(), 0);
    memset(auth_buffer, 0, BUFFER_SIZE);
    int valread = recv(sock, auth_buffer, BUFFER_SIZE, 0);
    if (valread <= 0) {
        std::cerr << "Server disconnected during auth." << std::endl;
        close(sock);
        return -1;
    }

    // Decrypt response
    xor_encrypt_decrypt(auth_buffer, valread, ENCRYPTION_KEY);
    std::string auth_response(auth_buffer, valread);

    if (auth_response != "AUTH_OK") {
        std::cerr << "Authentication failed. Server response: " <<
auth_response << std::endl;
        close(sock);
        return -1;
    }

    std::cout << "Authentication successful." << std::endl;
    std::cout << "Successfully connected to server " << SERVER_IP << ":"
<< PORT << std::endl;

    // 5. MAIN COMMAND LOOP
    std::string command_line;
    char buffer[BUFFER_SIZE] = {0};

    while (true) {
        std::cout << "\nEnter command (LIST, GET <file>, PUT <file>,
QUIT): ";
```

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

        if (command_line.empty()) continue;

        char command_buffer[BUFFER_SIZE] = {0};
        strncpy(command_buffer, command_line.c_str(), BUFFER_SIZE);

        command_buffer[BUFFER_SIZE - 1] = '\0';

        xor_encrypt_decrypt(command_buffer, command_line.length(),
ENCRYPTION_KEY);
        send(sock, command_buffer, command_line.length(), 0);

        if (command_line == "QUIT") {
            break;
        }

        std::string command, filename;
        size_t space_pos = command_line.find(' ');
        if (space_pos != std::string::npos) {
            command = command_line.substr(0, space_pos);
            filename = command_line.substr(space_pos + 1);
        } else {
            command = command_line;
        }

        if (command == "LIST") {
            memset(buffer, 0, BUFFER_SIZE);
            int bytes_recvd = recv(sock, buffer, BUFFER_SIZE, 0);
            if (bytes_recvd > 0) {
                xor_encrypt_decrypt(buffer, bytes_recvd, ENCRYPTION_KEY);
                std::cout << "\n--- Available Files ---\n" <<
std::string(buffer, bytes_recvd) << "-----------------------\n";
            }

        } else if (command == "GET" && !filename.empty()) {
            receive_file(sock, filename);

        } else if (command == "PUT" && !filename.empty()) {
            send_file(sock, filename);
        }
    }

    close(sock);
    std::cout << "Client connection closed." << std::endl;
    return 0;
}

// ENCRYPTION HELPER (Unchanged)
```

```
void xor_encrypt_decrypt(char* data, size_t len, const std::string& key) {
    if (key.empty()) return;
    for (size_t i = 0; i < len; ++i) {
        data[i] = data[i] ^ key[i % key.length()];
    }
}

// SEND FILE (Unchanged)
void send_file(int sock, const std::string& filename) {
    std::string filepath = CLIENT_DIR + filename;

    std::ifstream file(filepath, std::ios::in | std::ios::binary);
    if (!file.is_open()) {
        std::cerr << "Local file not found: " << filename << std::endl;
        return;
    }

    // 1. Send file size
    struct stat file_status;
    stat(filepath.c_str(), &file_status);
    long file_size = file_status.st_size;

    // Encrypt and send file size
    long encrypted_size = file_size;
    xor_encrypt_decrypt(reinterpret_cast<char*>(&encrypted_size),
sizeof(long), ENCRYPTION_KEY);
    send(sock, &encrypted_size, sizeof(long), 0);

    // 2. Send file chunks
    char buffer[BUFFER_SIZE];
    while (file.read(buffer, BUFFER_SIZE)) {
        // Encrypt chunk
        xor_encrypt_decrypt(buffer, BUFFER_SIZE, ENCRYPTION_KEY);
        send(sock, buffer, BUFFER_SIZE, 0);
    }
    if (file.gcount() > 0) {
        // Encrypt last chunk
        xor_encrypt_decrypt(buffer, file.gcount(), ENCRYPTION_KEY);
        send(sock, buffer, file.gcount(), 0);
    }

    file.close();
    std::cout << "Successfully uploaded file: " << filename << std::endl;
}

// RECEIVE FILE (Unchanged)
void receive_file(int sock, const std::string& filename) {
    // 1. Receive file size (or error message)
```

```
    long file_size;
    if (recv(sock, &file_size, sizeof(long), 0) <= 0) {
        std::cerr << "Error receiving file size." << std::endl;
        return;
    }
    // Decrypt file size
    xor_encrypt_decrypt(reinterpret_cast<char*>(&file_size), sizeof(long),
ENCRYPTION_KEY);

    // Check for server error
    if (file_size == -1) {
        char error_buffer[BUFFER_SIZE] = {0};
        int bytes_recvd = recv(sock, error_buffer, BUFFER_SIZE, 0);
        if (bytes_recvd > 0) {
            xor_encrypt_decrypt(error_buffer, bytes_recvd,
ENCRYPTION_KEY);
            std::cerr << "Server Error: " << std::string(error_buffer,
bytes_recvd) << std::endl;
        }
        return;
    }

    // 2. Open file for writing
    std::string filepath = CLIENT_DIR + filename;
    std::ofstream file(filepath, std::ios::out | std::ios::binary);
    if (!file.is_open()) {
        std::cerr << "Error opening file for writing: " << filename <<
std::endl;
        return;
    }

    // 3. Receive file chunks
    char buffer[BUFFER_SIZE];
    long total_received = 0;

    while (total_received < file_size) {
        long bytes_to_read = std::min((long)BUFFER_SIZE, file_size -
total_received);

        int bytes_received = recv(sock, buffer, bytes_to_read, 0);

        if (bytes_received <= 0) {
            std::cerr << "Connection closed prematurely or error." <<
std::endl;
            break;
        }

        // Decrypt chunk
        xor_encrypt_decrypt(buffer, bytes_received, ENCRYPTION_KEY);
```

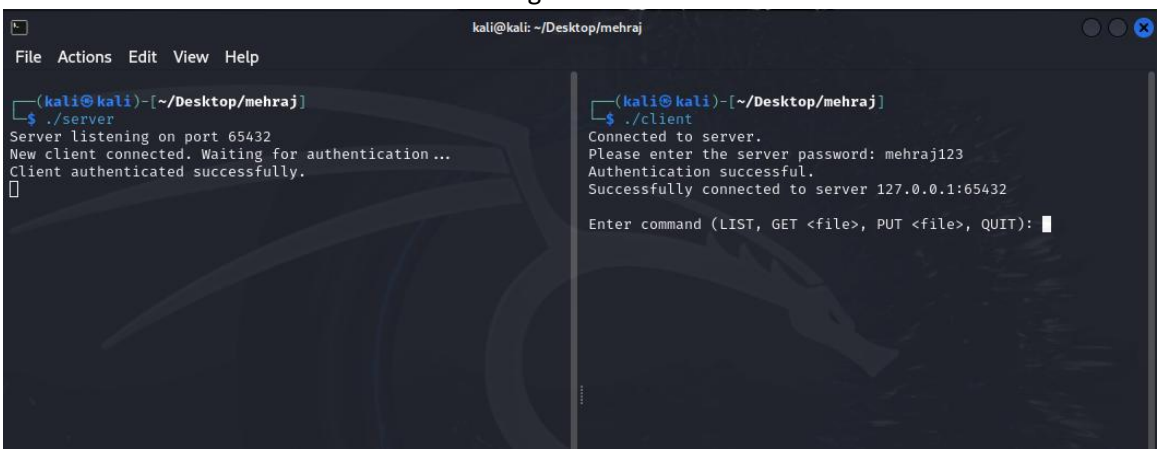
```
        file.write(buffer, bytes_received);
        total_received += bytes_received;
    }

    file.close();
    std::cout << "Successfully downloaded file: " << filename << " (" <<
total_received << " bytes)" << std::endl;
}
```

Full Screenshots Section

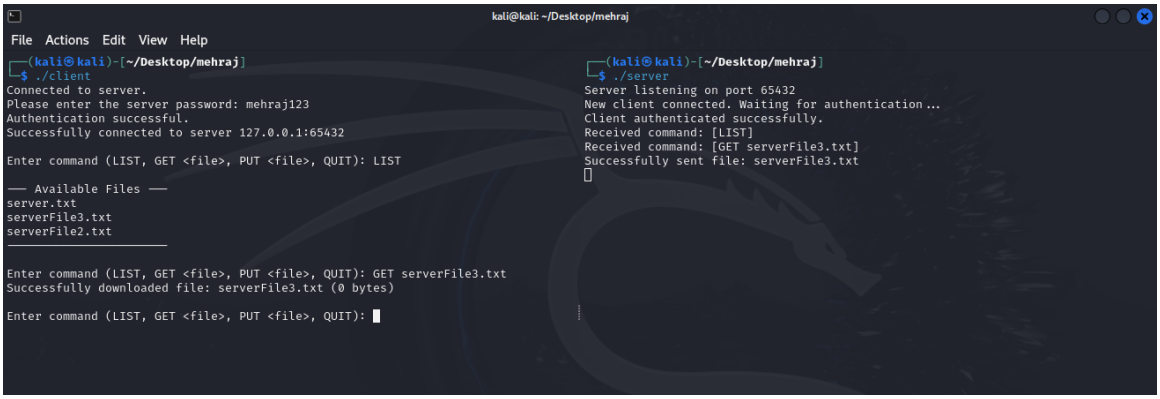
Screenshot 1: Server, connected successfully on port 65432 and authentication prompt shown

Screenshot 2 Authentication success message on both client and server sides

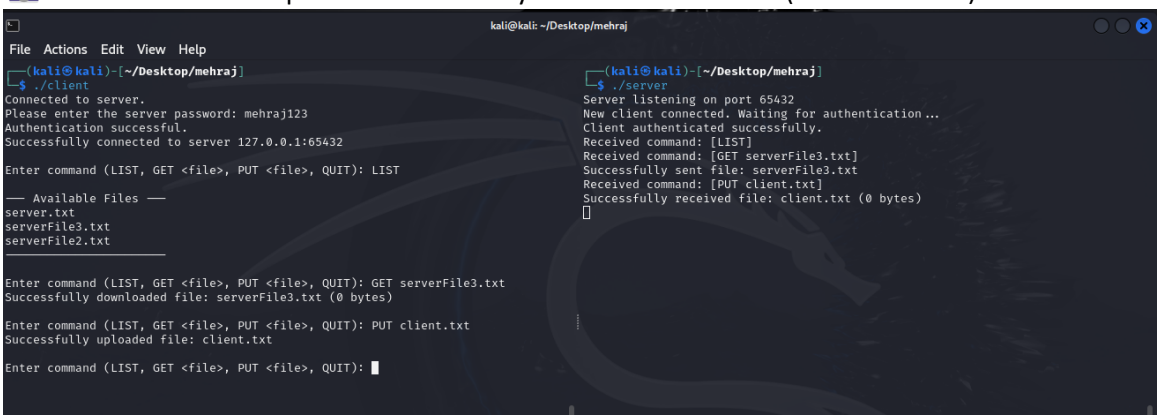


Screenshot 4: File listing shown to client (LIST command output)

Screenshot 5: File transfer in progress (GET command)



📸 **Screenshot 6:** File uploaded successfully from client to server (PUT command)



Conclusion:

This project successfully demonstrates **file sharing over TCP sockets** between a client and server with added **authentication and encryption** for basic security. It strengthens understanding of **network communication**, **socket APIs**, and **data transmission handling** in real-world systems.