

P2P COMMUNICATION

COMPUTER NETWORKS LAB PROJECT



**Submitted to : Ma’am Rabia Answer**

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | Eman Fatima | 22F-3343 | BSCS 5C |
| 2 | Ishrat | 22F-3616 | BSCS 5C |
| 3 | Aneela Bashir | 22F-3414 | BSCS 5C |
| 4 | Sahla | 22F-3435 | BSCS 5C |

**Date : 9-12-2024**

**Submitted by :**

**List of implemented peer operation(s)**

* Send message
* Send file
* Upload file
* Delete file
* Download file
* Preview file
* Preview peer file
* Query
* Published
* Start server
* Connect to server
* Quite

**Server.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <sys/stat.h>

#include <pthread.h>

#include <dirent.h> // Include this header for directory operations

#include <arpa/inet.h>

#include <openssl/evp.h>

#include <openssl/aes.h>

#include <errno.h>

#include <fcntl.h>

#include <signal.h>

#include <openssl/evp.h>

#include <openssl/aes.h>

#include <openssl/rand.h>

#define BUFFER\_SIZE 1024 // Increased buffer size

#define USERNAME "user"

#define PASSWORD "password"

#define FILE\_FOLDER "server\_folder"

#define KEY\_SIZE 32 // AES-256

#define IV\_SIZE 16 // AES block size

int client\_socket = -1;

int server\_socket = -1;

#define ENCRYPTION\_KEY "1234567890abcdef" // 16-byte (128-bit) key

void query\_files(int client\_socket)

{

DIR \*dir;

struct dirent \*entry;

// Open the directory where files are stored

if ((dir = opendir(FILE\_FOLDER)) == NULL) {

perror("Failed to open directory");

send(client\_socket, "Error opening directory.\n", 25, 0);

return;

}

// Send the list of files in the directory to the client

char file\_list[BUFFER\_SIZE];

int first\_file = 1;

while ((entry = readdir(dir)) != NULL) {

// Skip "." and ".." entries (current and parent directories)

if (strcmp(entry->d\_name, ".") != 0 && strcmp(entry->d\_name, "..") != 0) {

// Format the file name

snprintf(file\_list, sizeof(file\_list), "%s\n", entry->d\_name);

// Send file name to client

send(client\_socket, file\_list, strlen(file\_list), 0);

first\_file = 0;

}

}

// If no files were found, notify the client

if (first\_file) {

send(client\_socket, "No files available.\n", 20, 0);

}

// Close the directory

closedir(dir);

}

int decrypt\_data(const unsigned char \*encrypted\_data, unsigned char \*decrypted\_data, int encrypted\_data\_len, const unsigned char \*key, const unsigned char \*iv) {

EVP\_CIPHER\_CTX \*ctx = EVP\_CIPHER\_CTX\_new();

int len;

int decrypted\_len = 0;

// Initialize the decryption operation (AES-256-CBC in this case)

if (1 != EVP\_DecryptInit\_ex(ctx, EVP\_aes\_256\_cbc(), NULL, key, iv)) {

EVP\_CIPHER\_CTX\_free(ctx);

return -1; // Initialization failed

}

// Decrypt the message

if (1 != EVP\_DecryptUpdate(ctx, decrypted\_data, &len, encrypted\_data, encrypted\_data\_len)) {

EVP\_CIPHER\_CTX\_free(ctx);

return -2; // Decryption update failed

}

decrypted\_len = len;

// Finalize the decryption

if (1 != EVP\_DecryptFinal\_ex(ctx, decrypted\_data + len, &len)) {

EVP\_CIPHER\_CTX\_free(ctx);

return -3; // Finalization failed (padding issue or data corruption)

}

decrypted\_len += len;

// Clean up

EVP\_CIPHER\_CTX\_free(ctx);

return decrypted\_len;

}

// Function to create the shared folder if it doesn't exist

void create\_shared\_folder() {

if (access(FILE\_FOLDER, F\_OK) == -1) {

if (mkdir(FILE\_FOLDER, 0700) == -1) {

perror("Failed to create shared folder");

exit(EXIT\_FAILURE);

}

printf("Shared folder '%s' created.\n", FILE\_FOLDER);

}

}

void handle\_file\_request\_peer(int client\_socket, const char \*filename)

{

char filepath[BUFFER\_SIZE];

snprintf(filepath, sizeof(filepath), "%s/%s", FILE\_FOLDER, filename);

FILE \*file = fopen(filepath, "r");

if (file == NULL)

{

send(client\_socket, "File not found.\n", 16, 0);

send(client\_socket, "END\_OF\_PREVIEW", 15, 0); // Send preview end marker

return;

}

char buffer[BUFFER\_SIZE];

while (fgets(buffer, sizeof(buffer), file))

{

send(client\_socket, buffer, strlen(buffer), 0);

}

fclose(file);

// Send end-of-preview marker

send(client\_socket, "END\_OF\_PREVIEW", 15, 0);

}

// Function to create a test file in the server folder

void create\_test\_file() {

char filepath[BUFFER\_SIZE];

snprintf(filepath, sizeof(filepath), "%s/server.txt", FILE\_FOLDER);

FILE \*file = fopen(filepath, "w");

if (!file) {

perror("Failed to create test file");

return;

}

fprintf(file, "This is a server file created automatically by the server program.\n");

fclose(file);

printf("Test file created at %s\n", filepath);

}

// Function to handle server-side client communication

void \*handle\_client(void \*arg)

{

int sock = \*(int \*)arg;

char buffer[BUFFER\_SIZE];

ssize\_t bytes\_received;

// Step 1: Authentication

char username[BUFFER\_SIZE], password[BUFFER\_SIZE];

send(sock, "Enter username: ", 16, 0);

bytes\_received = recv(sock, username, sizeof(username), 0);

if (bytes\_received <= 0)

{

printf("Error receiving username.\n");

close(sock);

return NULL;

}

username[bytes\_received] = '\0';

send(sock, "Enter password: ", 16, 0);

bytes\_received = recv(sock, password, sizeof(password), 0);

if (bytes\_received <= 0)

{

printf("Error receiving password.\n");

close(sock);

return NULL;

}

password[bytes\_received] = '\0';

// Validate credentials

if (strcmp(username, USERNAME) != 0 || strcmp(password, PASSWORD) != 0)

{

send(sock, "Authentication failed. Closing connection.\n", 44, 0);

close(sock);

return NULL;

}

send(sock, "Authentication successful.\n", 26, 0);

printf("Client authenticated successfully.\n");

// Step 2: Communication with client

while (1) {

bytes\_received = recv(sock, buffer, sizeof(buffer), 0);

if (bytes\_received <= 0) {

perror("Failed to receive data");

close(sock);

return NULL;

}

buffer[bytes\_received] = '\0'; // Null terminate the received string

// Tokenize the command by splitting at the first space

char \*command = strtok(buffer, " "); // Get the first word (before space)

printf("Received command: '%s'\n", command);

if (command != NULL) {

if (strcmp(command, "SEND\_MESSAGE:") == 0)

{

printf("Decrypted message: %s\n", buffer + strlen(command) + 1);// Decrypt and print the message part after the command

return NULL;

}

else if (strcmp(command, "SEND\_FILE:") == 0)

{

// Extract the filename from the buffer after the "SEND\_FILE:" command

char \*filename = buffer + strlen(command) + 1; // Move past the command part to get the filename

printf("Received file command: %s\n", filename);

// Step 2: Construct the file path where to save the file

char file\_path[BUFFER\_SIZE];

strcpy(file\_path, FILE\_FOLDER);

if (file\_path[strlen(file\_path) - 1] != '/')

{

strcat(file\_path, "/");

}

strcat(file\_path, filename);

// Security check for invalid filenames

if (strchr(filename, '/') != NULL || strchr(filename, '\\') != NULL) {

fprintf(stderr, "Invalid filename.\n");

close(sock);

return NULL;

}

// Ensure file path doesn't exceed buffer size

if (strlen(file\_path) >= sizeof(file\_path))

{

fprintf(stderr, "Error: Path too long to create file.\n");

close(sock);

return NULL;

}

// Step 3: Open file for writing (create file if not exists)

FILE \*file = fopen(file\_path, "wb");

if (!file) {

perror("Failed to create file");

close(sock);

return NULL;

}

// Step 4: Receive file data in chunks and write it to the file

while ((bytes\_received = recv(sock, buffer, BUFFER\_SIZE, 0)) > 0) {

if (fwrite(buffer, 1, bytes\_received, file) != bytes\_received) {

perror("Error writing to file");

fclose(file);

close(sock);

return NULL;

}

}

// Step 5: If file is received successfully

if (bytes\_received == 0)

{

printf("File '%s' received and stored successfully.\n", filename);

send(sock, "FILE\_RECEIVED", strlen("FILE\_RECEIVED"), 0);

}

else

{

perror("Error receiving file data");

}

fclose(file);

close(sock);

return NULL;

}

else if (strcmp(command, "UPLOAD\_FILE:") == 0)

{

char \*filename = buffer + strlen(command) + 1; // Move past the command part to get the filename

printf("File name: %s\n", filename);

// Print the file content that follows

printf("File content:\n");

// Now expect the file data in chunks and print it instead of writing to a file

while ((bytes\_received = recv(sock, buffer, sizeof(buffer), 0)) > 0)

{

// Print the received data to the console

fwrite(buffer, 1, bytes\_received, stdout); // Display the file content

}

if (bytes\_received == 0)

{

printf("\nFile '%s' received successfully.\n", filename);

}

else

{

perror("Error receiving file data");

}

}

else if (strcmp(command, "DOWNLOAD\_FILE:") == 0)

{

// Extract the filename from the buffer after the "SEND\_FILE:" command

char \*filename = buffer + strlen(command) + 1; // Move past the command part to get the filename

printf("Received file command: %s\n", filename);

// Step 2: Construct the file path where to save the file

char file\_path[BUFFER\_SIZE];

strcpy(file\_path, FILE\_FOLDER);

if (file\_path[strlen(file\_path) - 1] != '/')

{

strcat(file\_path, "/");

}

strcat(file\_path, filename);

// Ensure file path doesn't exceed buffer size

if (strlen(file\_path) >= sizeof(file\_path))

{

fprintf(stderr, "Error: Path too long to create file.\n");

close(sock);

return NULL;

}

// Step 3: Open file for writing (create file if not exists)

FILE \*file = fopen(file\_path, "wb");

if (!file) {

perror("Failed to create file");

close(sock);

return NULL;

}

// Step 4: Receive file data in chunks and write it to the file

while ((bytes\_received = recv(sock, buffer, BUFFER\_SIZE, 0)) > 0) {

if (fwrite(buffer, 1, bytes\_received, file) != bytes\_received) {

perror("Error writing to file");

fclose(file);

close(sock);

return NULL;

}

}

// Step 5: If file is received successfully

if (bytes\_received == 0)

{

printf("File '%s' received and stored successfully.\n", filename);

} else

{

perror("Error receiving file data");

}

fclose(file);

return NULL;

}

if (strncmp(buffer, "PREVIEW:", 8) == 0)

{

char \*filename = buffer + strlen(command) + 1; // Move past the command part to get the filename

printf("Received file command: %s\n", filename);

printf("File preview requested for: %s\n", filename); // Display a preview message on the server

// Construct the full path to the file

char file\_path[BUFFER\_SIZE];

strcpy(file\_path, FILE\_FOLDER);

strcat(file\_path, "/");

strcat(file\_path, filename);

// Open the file for reading

FILE \*file = fopen(file\_path, "rb");

if (file == NULL) {

perror("Failed to open file");

send(sock, "Failed to open file.\n", 20, 0);

return NULL;

}

// Read and send file content in chunks to the client

char buffer\_preview[BUFFER\_SIZE];

size\_t bytes\_read;

while ((bytes\_read = fread(buffer\_preview, 1, sizeof(buffer\_preview), file)) > 0) {

// Send file content back to the peer for preview

send(sock, buffer\_preview, bytes\_read, 0);

}

fclose(file);

// Send the end-of-preview message

send(sock, "\nFile preview ended.\n", 21, 0);

}

else if (strncmp(command, "PUBLISH:", 8) == 0)

{

// Extract the filename from the command (skip past the "PUBLISH:" part)

char \*filename = buffer + strlen(command) + 1; // Move past the command part to get the filename

printf("Received publish request for file: %s\n", filename);

// Construct the full file path

char file\_path[BUFFER\_SIZE];

strcpy(file\_path, FILE\_FOLDER); // Assuming FILE\_FOLDER is defined correctly

strcat(file\_path, "/");

strcat(file\_path, filename); // Append the filename to the folder path

// Open the file in read mode

FILE \*file = fopen(file\_path, "rb");

if (!file) {

perror("Error opening file");

char \*error\_message = "Error opening file.\n";

send(sock, error\_message, strlen(error\_message), 0); // Send error message to the client

return NULL;

}

// Print the file content that follows

printf("Displaying content of file '%s':\n", filename);

// Receive and print the file content in chunks

char buffer[BUFFER\_SIZE];

size\_t bytes\_received;

while ((bytes\_received = fread(buffer, 1, sizeof(buffer), file)) > 0) {

// Print the received data to the console

fwrite(buffer, 1, bytes\_received, stdout); // Display the file content

// Optionally, send the content back to the client if needed

send(sock, buffer, bytes\_received, 0);

}

fclose(file); // Close the file after reading its content

printf("\nFile '%s' displayed successfully.\n", filename);

}

else if (strcmp(command, "QUERY") == 0) {

// Open the server folder

DIR \*dir = opendir(FILE\_FOLDER);

if (!dir) {

perror("Error opening directory");

return NULL;

}

// Prepare the list of files to send

char file\_list[BUFFER\_SIZE];

file\_list[0] = '\0'; // Initialize the file list

struct dirent \*entry;

while ((entry = readdir(dir)) != NULL)

{

if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0)

{

continue;

}

// Add the filename to the list

if (strlen(file\_list) + strlen(entry->d\_name) + 2 < sizeof(file\_list))

{

strcat(file\_list, entry->d\_name);

strcat(file\_list, "\n"); // Newline separates filenames

}

}

closedir(dir);

// If no files were found, send a message indicating so

if (strlen(file\_list) == 0)

{

strcpy(file\_list, "No files available in the server folder.\n");

}

// Send the list of files to the client

send(sock, file\_list, strlen(file\_list), 0);

}

else

{

//printf("Unknown command received: %s\n", buffer);

//send(sock, "Unknown command.\n", 18, 0);

}

}

}

close(sock);

}

// Function to start the server

void start\_server(int port)

{

struct sockaddr\_in server\_addr, client\_addr;

socklen\_t client\_len = sizeof(client\_addr);

server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_socket == -1) {

perror("Failed to create server socket");

exit(EXIT\_FAILURE);

}

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(port);

if (bind(server\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) == -1) {

perror("Failed to bind server socket");

exit(EXIT\_FAILURE);

}

if (listen(server\_socket, 5) == -1) {

perror("Failed to listen on server socket");

exit(EXIT\_FAILURE);

}

printf("Server started on port %d. Waiting for connections...\n", port);

while (1) {

int client\_socket = accept(server\_socket, (struct sockaddr \*)&client\_addr, &client\_len);

if (client\_socket == -1) {

perror("Failed to accept client connection");

continue;

}

printf("Client connected.\n");

pthread\_t server\_thread;

if (pthread\_create(&server\_thread, NULL, handle\_client, &client\_socket) != 0) {

perror("Failed to create thread");

close(client\_socket);

continue;

}

pthread\_detach(server\_thread);

}

}

void signal\_handler(int sig) {

printf("\nServer shutting down...\n");

// Notify the client that the server is shutting down

if (client\_socket != -1) {

const char \*shutdown\_message = "Server is shutting down...\n";

send(client\_socket, shutdown\_message, strlen(shutdown\_message), 0);

close(client\_socket); // Close the client socket

}

if (server\_socket != -1) {

close(server\_socket); // Close the server socket

}

exit(0);

}

int main() {

int port;

printf("Enter port to start server on: ");

scanf("%d", &port);

getchar(); // Consume newline

create\_shared\_folder();

create\_test\_file();

signal(SIGINT, signal\_handler); // Handle Ctrl+C to clean up resources

start\_server(port);

return 0;

}

**Client.c :**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <sys/stat.h>

#include <pthread.h>

#include <dirent.h> // Include this header for directory operations

#include <arpa/inet.h>

#include <openssl/evp.h>

#include <openssl/aes.h>

#include <errno.h>

#include <fcntl.h>

#include <signal.h>

#define BUFFER\_SIZE 1024

#define FILE\_FOLDER "p2p-files"

#define USERNAME "user"

#define PASSWORD "password"

#define ENCRYPTION\_KEY "0123456789abcdef" // AES encryption key (16 bytes)

// Global variables

extern int peer\_socket; // Connection to the peer

extern int server\_socket; // Connection to the server

int server\_socket = -1;

int client\_socket = -1;

int peer\_socket = -1;

int encrypt\_data(const unsigned char \*input, unsigned char \*output, int length) {

EVP\_CIPHER\_CTX \*ctx;

int len;

int ciphertext\_len;

// Initialize the context

if (!(ctx = EVP\_CIPHER\_CTX\_new())) {

perror("EVP\_CIPHER\_CTX\_new failed");

return -1;

}

// Initialize the encryption operation with AES-128-ECB

if (1 != EVP\_EncryptInit\_ex(ctx, EVP\_aes\_128\_ecb(), NULL, (unsigned char \*)ENCRYPTION\_KEY, NULL)) {

perror("EVP\_EncryptInit\_ex failed");

EVP\_CIPHER\_CTX\_free(ctx);

return -1;

}

// Encrypt the input data

if (1 != EVP\_EncryptUpdate(ctx, output, &len, input, length)) {

perror("EVP\_EncryptUpdate failed");

EVP\_CIPHER\_CTX\_free(ctx);

return -1;

}

ciphertext\_len = len;

// Finalize encryption

if (1 != EVP\_EncryptFinal\_ex(ctx, output + len, &len)) {

perror("EVP\_EncryptFinal\_ex failed");

EVP\_CIPHER\_CTX\_free(ctx);

return -1;

}

ciphertext\_len += len;

// Clean up

EVP\_CIPHER\_CTX\_free(ctx);

// Debugging: Print encrypted data in hex

printf("Encrypted data (hex): ");

for (int i = 0; i < ciphertext\_len; i++) {

printf("%02x ", output[i]);

}

printf("\n");

return ciphertext\_len; // Return the length of the encrypted data

}

int authenticate\_client(int client\_socket) {

char buffer[BUFFER\_SIZE];

// Send authentication prompt to client

send(client\_socket, "Enter username: ", strlen("Enter username: "), 0);

recv(client\_socket, buffer, sizeof(buffer), 0); // Receive username from client

buffer[strcspn(buffer, "\n")] = 0; // Remove newline character

if (strcmp(buffer, USERNAME) != 0) {

send(client\_socket, "Invalid username.\n", strlen("Invalid username.\n"), 0);

return 0; // Authentication failed

}

// Send password prompt to client

send(client\_socket, "Enter password: ", strlen("Enter password: "), 0);

recv(client\_socket, buffer, sizeof(buffer), 0); // Receive password from client

buffer[strcspn(buffer, "\n")] = 0; // Remove newline character

if (strcmp(buffer,PASSWORD ) != 0) {

send(client\_socket, "Invalid password.\n", strlen("Invalid password.\n"), 0);

return 0; // Authentication failed

}

send(client\_socket, "Authentication successful.\n", strlen("Authentication successful.\n"), 0);

return 1; // Authentication successful

}

// Function to handle server-side client communication

void \*handle\_client(void \*arg) {

int sock = \*(int \*)arg;

char buffer[BUFFER\_SIZE];

ssize\_t bytes\_received;

// Authentication

if (!authenticate\_client(sock)) {

printf("Authentication failed. Closing connection.\n");

close(sock);

return NULL;

}

printf("Authentication successful. Proceeding with communication.\n");

// Send a welcome message to the client

send(sock, "Welcome to the server!\n", strlen("Welcome to the server!\n"), 0);

while (1) {

// Step 1: Receive client message (e.g., "query" or filename for file upload)

bytes\_received = recv(sock, buffer, sizeof(buffer), 0);

if (bytes\_received <= 0) {

// Connection closed or error

printf("Connection closed by peer.\n");

break;

}

buffer[bytes\_received] = '\0'; // Null-terminate the received string

// Handle "query" command to list files in the shared folder

if (strcmp(buffer, "query") == 0)

{

char file\_list[BUFFER\_SIZE] = "";

struct dirent \*entry;

DIR \*dir = opendir(FILE\_FOLDER);

if (dir == NULL) {

perror("Failed to open directory");

break;

}

// Iterate through the files in the folder

while ((entry = readdir(dir)) != NULL)

{

// Skip current directory and parent directory entries ('.' and '..')

if (strcmp(entry->d\_name, ".") != 0 && strcmp(entry->d\_name, "..") != 0) {

strcat(file\_list, entry->d\_name);

strcat(file\_list, "\n");

}

}

closedir(dir);

// Send the list of files to the client

if (strlen(file\_list) == 0)

{

strcpy(file\_list, "No files found.\n");

}

send(sock, file\_list, strlen(file\_list), 0);

}

else

{

// Handle other client messages

printf("Received from client: %s\n", buffer);

// Handle file upload logic here

char file\_path[BUFFER\_SIZE];

strcpy(file\_path, FILE\_FOLDER);

// Ensure there is a '/' separating the directory and the filename

if (file\_path[strlen(file\_path) - 1] != '/') {

strcat(file\_path, "/");

}

// Concatenate the received filename to the path

strcat(file\_path, buffer);

// Ensure the file path doesn't exceed buffer size

if (strlen(file\_path) >= sizeof(file\_path)) {

fprintf(stderr, "Error: Path too long to create file.\n");

break;

}

// Open the file for writing (create it if it doesn't exist)

FILE \*file = fopen(file\_path, "wb");

if (!file) {

perror("Failed to create file");

break;

}

// Step 2: Receive and write file data

while ((bytes\_received = recv(sock, buffer, BUFFER\_SIZE, 0)) > 0)

{

if (fwrite(buffer, 1, bytes\_received, file) != bytes\_received)

{

perror("Error writing to file");

fclose(file);

break;

}

}

// If the entire file is received successfully

if (bytes\_received == 0) {

printf("File '%s' received and stored successfully.\n", buffer);

} else {

perror("Error receiving file data");

}

fclose(file);

}

}

close(sock);

return NULL;

}

// Server code that listens for connections and starts a thread for each client

void start\_server(int port) {

struct sockaddr\_in server\_addr, client\_addr;

socklen\_t client\_len = sizeof(client\_addr);

// Create server socket

server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_socket == -1) {

perror("Failed to create server socket");

exit(EXIT\_FAILURE);

}

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(port);

// Bind socket to the address and port

if (bind(server\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) == -1) {

perror("Failed to bind server socket");

exit(EXIT\_FAILURE);

}

// Start listening for incoming connections

if (listen(server\_socket, 5) == -1) {

perror("Failed to listen on server socket");

exit(EXIT\_FAILURE);

}

printf("Server started on port %d. Waiting for connections...\n", port);

while (1) {

// Accept client connection

client\_socket = accept(server\_socket, (struct sockaddr \*)&client\_addr, &client\_len);

if (client\_socket == -1) {

perror("Failed to accept client connection");

continue;

}

printf("Client connected.\n");

// Create a thread to handle the client

pthread\_t server\_thread;

pthread\_create(&server\_thread, NULL, handle\_client, &client\_socket);

pthread\_detach(server\_thread); // Detach the thread so it can be cleaned up automatically

}

}

// Wrapper function to start the server in a separate thread

void \*server\_thread\_function(void \*arg) {

int port = \*(int \*)arg;

start\_server(port);

return NULL;

}

void authenticate\_with\_server() {

char buffer[BUFFER\_SIZE];

char username[BUFFER\_SIZE]; // Declare a local variable for username

char password[BUFFER\_SIZE]; // Declare a local variable for password

// Receive username prompt from server

recv(peer\_socket, buffer, sizeof(buffer), 0);

printf("%s", buffer);

// Send username to the server

fgets(username, sizeof(username), stdin);

username[strcspn(username, "\n")] = '\0'; // Remove newline character

send(peer\_socket, username, strlen(username), 0);

// Receive password prompt from server

recv(peer\_socket, buffer, sizeof(buffer), 0);

printf("%s", buffer);

// Send password to the server

fgets(password, sizeof(password), stdin);

password[strcspn(password, "\n")] = '\0'; // Remove newline character

send(peer\_socket, password, strlen(password), 0);

// Receive authentication result from server

recv(peer\_socket, buffer, sizeof(buffer), 0);

printf("%s", buffer);

// Handle authentication success or failure

if (strstr(buffer, "Authentication successful") == NULL) {

printf("Authentication failed. Exiting...\n");

close(peer\_socket);

exit(EXIT\_FAILURE); // Exit the program if authentication fails

}

printf("Authentication successful. You can now communicate with the server.\n");

}

void connect\_to\_peer(const char \*ip, int port) {

struct sockaddr\_in peer\_addr;

// Create socket

peer\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (peer\_socket == -1) {

perror("Failed to create peer socket");

exit(EXIT\_FAILURE);

}

peer\_addr.sin\_family = AF\_INET;

peer\_addr.sin\_port = htons(port);

// Convert IP address from text to binary

if (inet\_pton(AF\_INET, ip, &peer\_addr.sin\_addr) <= 0) {

perror("Invalid IP address format");

exit(EXIT\_FAILURE);

}

// Attempt to connect to the peer

if (connect(peer\_socket, (struct sockaddr \*)&peer\_addr, sizeof(peer\_addr)) == -1) {

perror("Failed to connect to peer");

exit(EXIT\_FAILURE); // Exit if connection fails

}

printf("Connected to peer %s:%d\n", ip, port);

// Authenticate after connecting

authenticate\_with\_server();

}

// Function to create a folder if it doesn't exist

void create\_shared\_folder() {

if (access(FILE\_FOLDER, F\_OK) == -1) {

// Directory doesn't exist, create it

if (mkdir(FILE\_FOLDER, 0700) == -1) {

perror("Failed to create shared folder");

exit(EXIT\_FAILURE);

}

printf("Shared folder '%s' created.\n", FILE\_FOLDER);

}

}

// Function to delete a file in the shared folder

void delete\_file()

{

char filename[BUFFER\_SIZE];

printf("Enter filename to delete: ");

fgets(filename, sizeof(filename), stdin);

filename[strcspn(filename, "\n")] = '\0';

if (strlen(filename) + strlen(FILE\_FOLDER) + 1 >= BUFFER\_SIZE) {

printf("Filename too long.\n");

return;

}

char filepath[BUFFER\_SIZE];

int written = snprintf(filepath, sizeof(filepath), "%s/%s", FILE\_FOLDER, filename);

if (written < 0 || written >= sizeof(filepath)) {

printf("Failed to construct filepath.\n");

return;

}

if (remove(filepath) == 0) {

printf("File deleted: %s\n", filepath);

} else {

perror("Error deleting file");

}

}

// Function to create a test file in the shared folder

void create\_test\_file() {

char filepath[BUFFER\_SIZE];

snprintf(filepath, sizeof(filepath), "%s/file.txt", FILE\_FOLDER);

// Create or overwrite the file

FILE \*file = fopen(filepath, "w");

if (!file) {

perror("Failed to create test file");

return;

}

// Write some content to the file

fprintf(file, "This is a test file created automatically by the client program.\n");

fclose(file);

printf("Test file created at %s\n", filepath);

}

// Function to send message to server

// Function to send message to peer

void send\_message\_to\_peer() {

if (peer\_socket == -1) {

printf("No peer connected. Use option 2 to connect to a peer first.\n");

return;

}

char message[BUFFER\_SIZE];

printf("Enter message: ");

fgets(message, sizeof(message), stdin);

message[strcspn(message, "\n")] = '\0'; // Remove newline character

unsigned char encrypted\_message[BUFFER\_SIZE];

int message\_length = strlen(message);

// Encrypt the message

int encrypted\_length = encrypt\_data((unsigned char \*)message, encrypted\_message, message\_length);

// Send the command and the encrypted message

char formatted\_message[BUFFER\_SIZE];

// Instead of snprintf, directly concatenate the command and message

strcpy(formatted\_message, "SEND\_MESSAGE: ");

strcat(formatted\_message, message); // Append message to the command

send(peer\_socket, formatted\_message, strlen(formatted\_message), 0); // Send the command first

printf("Encrypted message sent.\n");

}

// Function to send file to peer

void send\_file\_to\_peer()

{

if (peer\_socket == -1)

{

printf("No peer connected. Use option 2 to connect to a peer first.\n");

return;

}

char filename[BUFFER\_SIZE];

printf("Enter filename to send: ");

fgets(filename, sizeof(filename), stdin);

filename[strcspn(filename, "\n")] = '\0'; // Remove newline character

// Construct the filepath

char filepath[BUFFER\_SIZE];

strcpy(filepath, FILE\_FOLDER); // Copy the folder path

strcat(filepath, "/"); // Add the directory separator

strcat(filepath, filename); // Append the filename

FILE \*file = fopen(filepath, "rb");

if (!file)

{

printf("File not found: %s\n", filepath);

return;

}

// Send the command and the filename first

char formatted\_message[BUFFER\_SIZE];

strcpy(formatted\_message, "SEND\_FILE: "); // Copy command into the message

strcat(formatted\_message, filename); // Append the filename

// Ensure null-termination of the message

formatted\_message[BUFFER\_SIZE - 1] = '\0';

if (send(peer\_socket, formatted\_message, strlen(formatted\_message), 0) == -1) {

perror("Error sending command to peer");

fclose(file);

return;

}

// Now send the file content in chunks

char buffer[BUFFER\_SIZE];

size\_t bytes\_read;

while ((bytes\_read = fread(buffer, 1, BUFFER\_SIZE, file)) > 0) {

if (send(peer\_socket, buffer, bytes\_read, 0) == -1) {

perror("Error sending file data");

fclose(file);

return;

}

}

fclose(file);

printf("File '%s' sent successfully.\n", filename);

}

void upload\_file(const char \*file\_path)

{

if (peer\_socket == -1) {

printf("No peer connected. Please connect to a peer first.\n");

return;

}

FILE \*file = fopen(file\_path, "rb");

if (file == NULL) {

perror("Failed to open file");

return;

}

// Extract the filename from the full file path

char file\_name[BUFFER\_SIZE];

strcpy(file\_name, file\_path);

char \*file\_name\_only = strrchr(file\_name, '/'); // Get only the file name

if (file\_name\_only) {

file\_name\_only++; // Move past the '/' to get the file name

} else {

file\_name\_only = file\_name; // In case there is no directory in the path

}

// Send the "UPLOAD\_FILE" command with the file name

char formatted\_message[BUFFER\_SIZE];

strcpy(formatted\_message, "UPLOAD\_FILE: "); // Copy command into the message

strcat(formatted\_message, file\_name\_only); // Append the filename

if (send(peer\_socket, formatted\_message, strlen(formatted\_message), 0) == -1) {

perror("Error sending command to peer");

fclose(file);

return;

}

// Send the file content in chunks

char buffer[BUFFER\_SIZE];

size\_t bytes\_read;

while ((bytes\_read = fread(buffer, 1, sizeof(buffer), file)) > 0) {

if (send(peer\_socket, buffer, bytes\_read, 0) == -1) {

perror("Error sending file data");

fclose(file);

return;

}

}

printf("File uploaded successfully.\n");

fclose(file);

}

// Function to publish files

void publish\_files() {

if (peer\_socket == -1) {

printf("No peer connected. Please connect to a peer first.\n");

return;

}

// Prepare the list of files to publish

int num\_files;

printf("Enter the number of files to publish: ");

scanf("%d", &num\_files);

getchar(); // Consume newline

for (int i = 0; i < num\_files; i++) {

char filename[BUFFER\_SIZE];

printf("Enter filename to publish: ");

fgets(filename, sizeof(filename), stdin);

filename[strcspn(filename, "\n")] = '\0'; // Remove newline character

// Construct the full path to the file manually

char file\_path[BUFFER\_SIZE];

strcpy(file\_path, FILE\_FOLDER); // Copy folder path to file\_path

strcat(file\_path, "/"); // Append directory separator

strcat(file\_path, filename); // Append filename to the path

// Check if the file exists and if it's within the correct directory

FILE \*file = fopen(file\_path, "r");

if (!file) {

printf("Error: Could not open file %s in folder %s\n", filename, FILE\_FOLDER);

continue; // Skip to the next file if the current one can't be opened

}

// Verify the file is within the correct folder

if (strncmp(file\_path, FILE\_FOLDER, strlen(FILE\_FOLDER)) != 0) {

printf("Error: The file %s is not located in the correct directory: %s\n", filename, FILE\_FOLDER);

fclose(file);

continue; // Skip to the next file

}

// Send the "PUBLISH" command with the filename to the peer

char command[BUFFER\_SIZE];

strcpy(command, "PUBLISH: "); // Command to indicate publishing

strcat(command, filename); // Append filename to the command

send(peer\_socket, command, strlen(command), 0); // Send the "PUBLISH" command

// Display file content on the client side

printf("Displaying contents of file '%s':\n", filename);

char file\_content[BUFFER\_SIZE];

size\_t bytes\_read;

while ((bytes\_read = fread(file\_content, 1, sizeof(file\_content) - 1, file)) > 0) {

file\_content[bytes\_read] = '\0'; // Null-terminate the buffer

printf("%s", file\_content); // Print file content to the client console

send(peer\_socket, file\_content, bytes\_read, 0); // Send content to the peer

}

fclose(file);

printf("\nFile '%s' published successfully.\n", filename);

}

}

// Function to query files from the server (peer)

void query\_files() {

if (peer\_socket == -1) {

printf("No peer connected. Please connect to a peer first.\n");

return;

}

// Send a "QUERY" command to the connected peer to request the list of files

const char \*query\_command = "QUERY";

if (send(peer\_socket, query\_command, strlen(query\_command), 0) == -1) {

perror("Error sending QUERY command to peer");

return;

}

// Receive the list of files from the server (peer)

char buffer[BUFFER\_SIZE];

ssize\_t bytes\_received = recv(peer\_socket, buffer, sizeof(buffer), 0);

if (bytes\_received <= 0) {

printf("Failed to receive file list or peer closed the connection.\n");

return;

}

// Null-terminate the received data and display the file list

buffer[bytes\_received] = '\0';

printf("Files available from peer:\n%s\n", buffer);

}

void download\_file(const char \*file\_path)

{

if (peer\_socket == -1) {

printf("No peer connected. Please connect to a peer first.\n");

return;

}

FILE \*file = fopen(file\_path, "rb");

if (file == NULL) {

perror("Failed to open file");

return;

}

// Extract the filename from the full file path

char file\_name[BUFFER\_SIZE];

strcpy(file\_name, file\_path);

char \*file\_name\_only = strrchr(file\_name, '/'); // Get only the file name

if (file\_name\_only) {

file\_name\_only++; // Move past the '/' to get the file name

} else {

file\_name\_only = file\_name; // In case there is no directory in the path

}

// Send the "UPLOAD\_FILE" command with the file name

char formatted\_message[BUFFER\_SIZE];

strcpy(formatted\_message, "DOWNLOAD\_FILE: "); // Copy command into the message

strcat(formatted\_message, file\_name\_only); // Append the filename

if (send(peer\_socket, formatted\_message, strlen(formatted\_message), 0) == -1) {

perror("Error sending command to peer");

fclose(file);

return;

}

// Send the file content in chunks

char buffer[BUFFER\_SIZE];

size\_t bytes\_read;

while ((bytes\_read = fread(buffer, 1, sizeof(buffer), file)) > 0) {

if (send(peer\_socket, buffer, bytes\_read, 0) == -1) {

perror("Error sending file data");

fclose(file);

return;

}

}

printf("File download successfully.\n");

fclose(file);

}

void preview\_file\_peer(const char \*filename)

{

if (peer\_socket == -1) {

printf("No peer connected. Please connect to a peer first.\n");

return;

}

// Send the "PREVIEW" command with the filename to the server

char request[BUFFER\_SIZE];

strcpy(request, "PREVIEW: ");

strcat(request, filename); // Append filename to the command

send(peer\_socket, request, strlen(request), 0); // Send the request to the server

// Set up the file descriptor set for select()

fd\_set read\_fds;

FD\_ZERO(&read\_fds);

FD\_SET(peer\_socket, &read\_fds);

// Wait for the socket to be ready for reading, with a timeout of 1 second

struct timeval timeout;

timeout.tv\_sec = 1;

timeout.tv\_usec = 0;

char buffer[BUFFER\_SIZE];

int bytes\_received;

printf("File preview from peer:\n");

while (1) {

int select\_result = select(peer\_socket + 1, &read\_fds, NULL, NULL, &timeout);

if (select\_result == -1) {

perror("Select failed");

break;

}

else if (select\_result == 0)

{

printf("\nFile preview ended.\n");

break;

} else {

// Data is available, read it

bytes\_received = recv(peer\_socket, buffer, sizeof(buffer) - 1, 0);

if (bytes\_received > 0) {

buffer[bytes\_received] = '\0'; // Null-terminate the received data

printf("%s", buffer); // Print the received data (file content) to the client terminal

} else {

printf("\nFile preview ended.\n");

break;

}

}

}

}

void signal\_handler(int sig) {

printf("\nServer shutting down...\n");

// Notify the client that the server is shutting down

if (client\_socket != -1) {

const char \*shutdown\_message = "Server is shutting down...\n";

send(client\_socket, shutdown\_message, strlen(shutdown\_message), 0);

close(client\_socket); // Close the client socket

}

if (server\_socket != -1) {

close(server\_socket); // Close the server socket

}

exit(0);

}

void preview\_file(const char \*filename)

{

char filepath[BUFFER\_SIZE];

snprintf(filepath, sizeof(filepath), "%s/%s", FILE\_FOLDER, filename);

FILE \*file = fopen(filepath, "r");

if (file == NULL)

{

printf("File not found: %s\n", filename);

return;

}

// Print file contents

char buffer[BUFFER\_SIZE];

while (fgets(buffer, sizeof(buffer), file))

{

printf("%s", buffer);

}

fclose(file); // Close the file after reading

}

// Main function

int main()

{

int choice;

char ip[BUFFER\_SIZE];

int port;

char message[BUFFER\_SIZE];

char filename[BUFFER\_SIZE];

// Create shared folder and test file

create\_shared\_folder();

create\_test\_file();

while (1) {

printf("\nMenu:\n");

printf("1. Start server\n");

printf("2. Connect to server\n");

printf("3. Send message to server\n");

printf("4. Send file to server\n");

printf("5. upload files(from whole system)\n");

printf("6. download files\n");

printf("7. preview files(of our own folder)\n");

printf("8. preview files(of our server folder)\n");

printf("9. Publish files\n");

printf("10. Query files\n");

printf("11. Delete file\n");

printf("12. Exit\n");

printf("Enter choice: ");

scanf("%d", &choice);

getchar(); // Consume newline

switch (choice) {

case 1:

pthread\_t server\_thread;

int port;

printf("Enter port: ");

scanf("%d", &port);

getchar(); // Consume newline

signal(SIGINT, signal\_handler); // Handle Ctrl+C to clean up resources

pthread\_create(&server\_thread, NULL, server\_thread\_function, &port);

pthread\_join(server\_thread, NULL); // Wait for the server thread to finish

break;

case 2:

printf("Enter peer IP: ");

fgets(ip, sizeof(ip), stdin);

ip[strcspn(ip, "\n")] = '\0';

printf("Enter peer port: ");

scanf("%d", &port);

getchar(); // Consume newline

connect\_to\_peer(ip, port);

break;

case 3:

send\_message\_to\_peer();

break;

case 4:

send\_file\_to\_peer();

break;

case 5:

printf("Enter the path of the file to upload: ");

fgets(message, sizeof(message), stdin);

message[strcspn(message, "\n")] = '\0'; // Remove newline character

upload\_file(message);

break;

case 6:

printf("Enter the path of file to download: ");

fgets(message, sizeof(message), stdin);

message[strcspn(message, "\n")] = '\0';

download\_file(message);

break;

case 7:

printf("Enter the file name to preview: ");

fgets(message, sizeof(message), stdin);

message[strcspn(message, "\n")] = '\0';

preview\_file(message);

break;

case 8:

printf("Enter the file name to preview of server: ");

fgets(message, sizeof(message), stdin);

message[strcspn(message, "\n")] = '\0';

preview\_file\_peer(message);

break;

case 9:

publish\_files(); // Publish files to the connected peer

break;

case 10:

query\_files(); // Query files from the connected peer

break;

case 11:

delete\_file();

break;

case 12:

exit(0);

default:

printf("Invalid choice.\n");

}

}

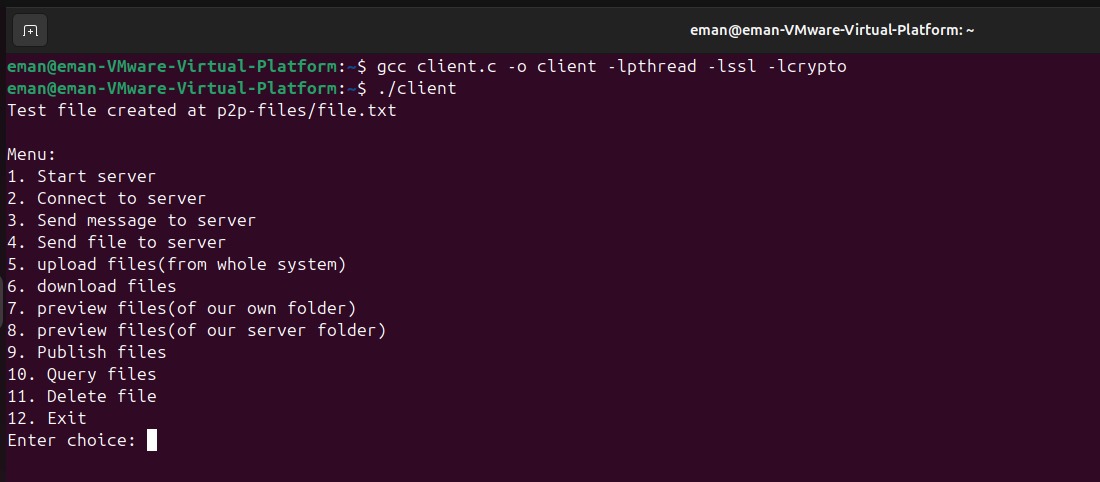
return 0;

}

**Outputs:**

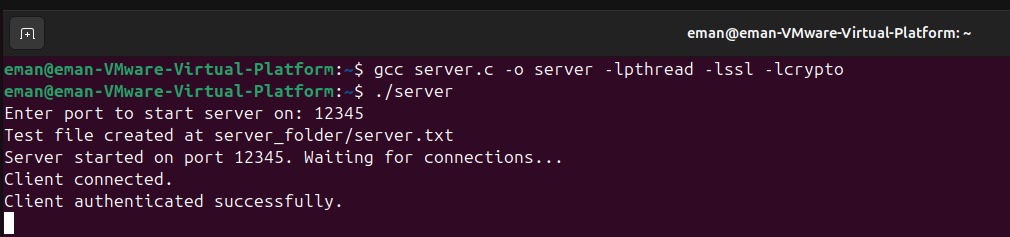
A screenshot of a computer

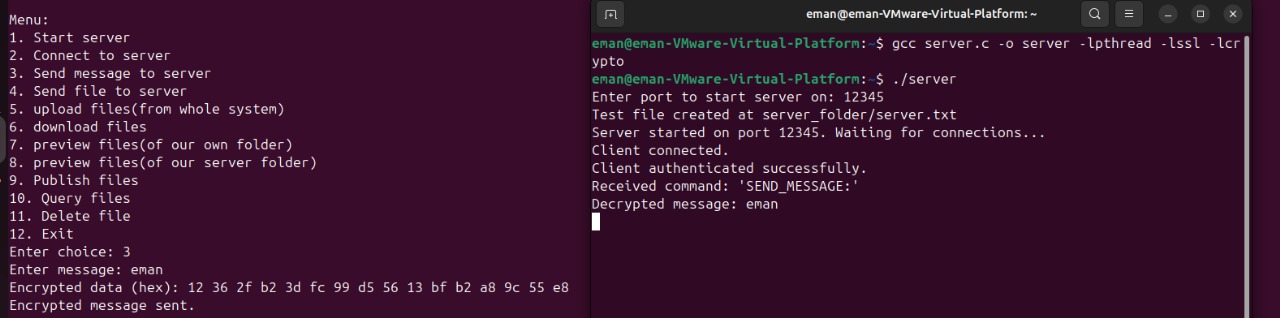
Description automatically generated

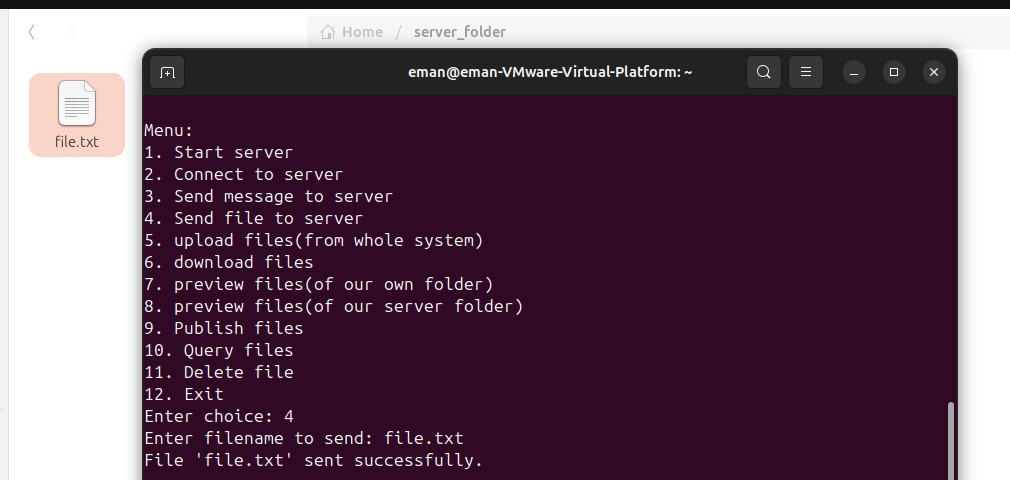


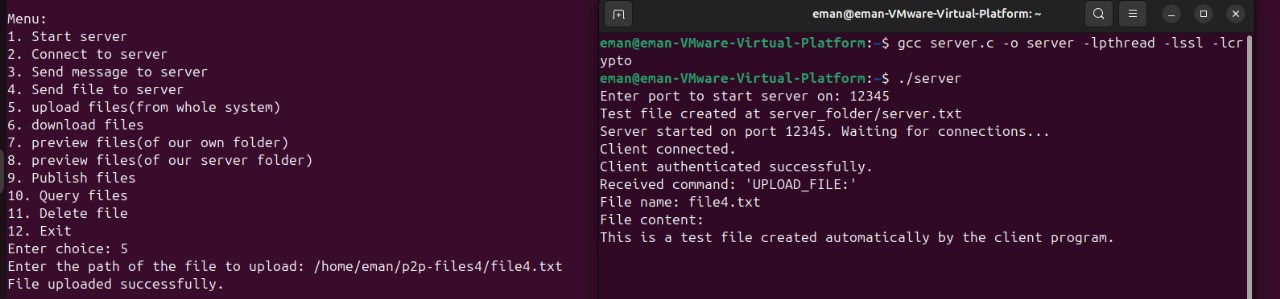
A screenshot of a computer

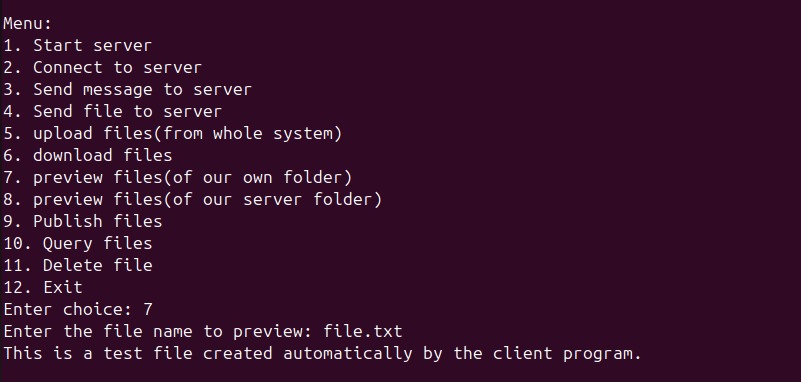
Description automatically generated

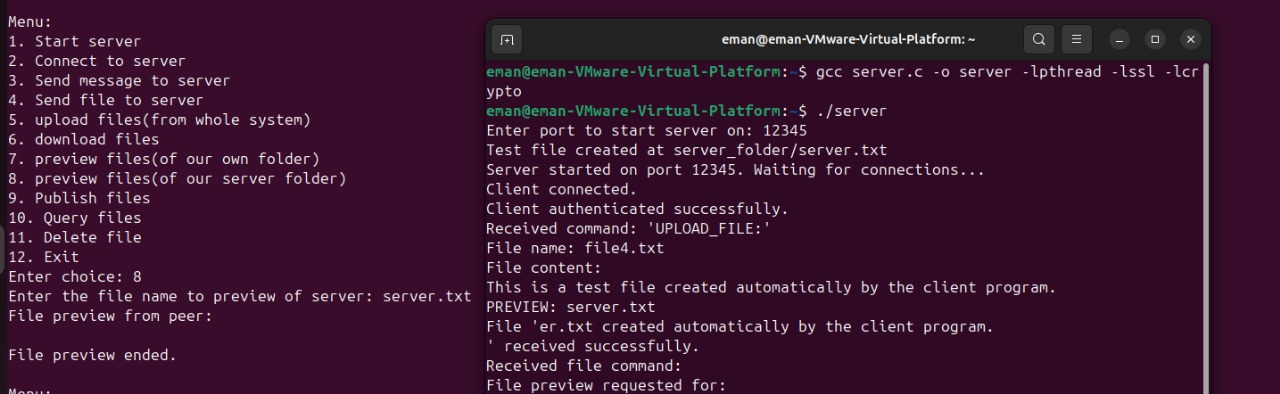


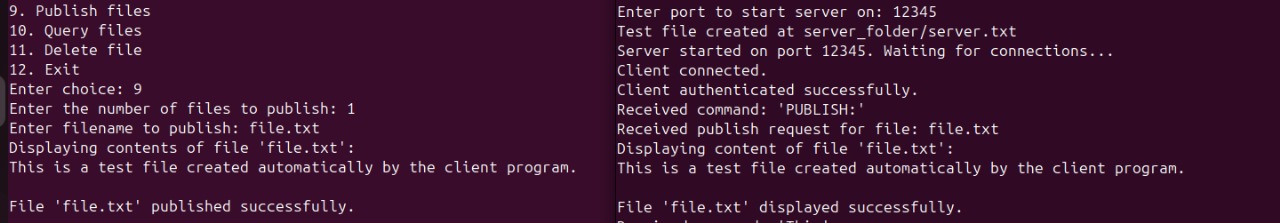


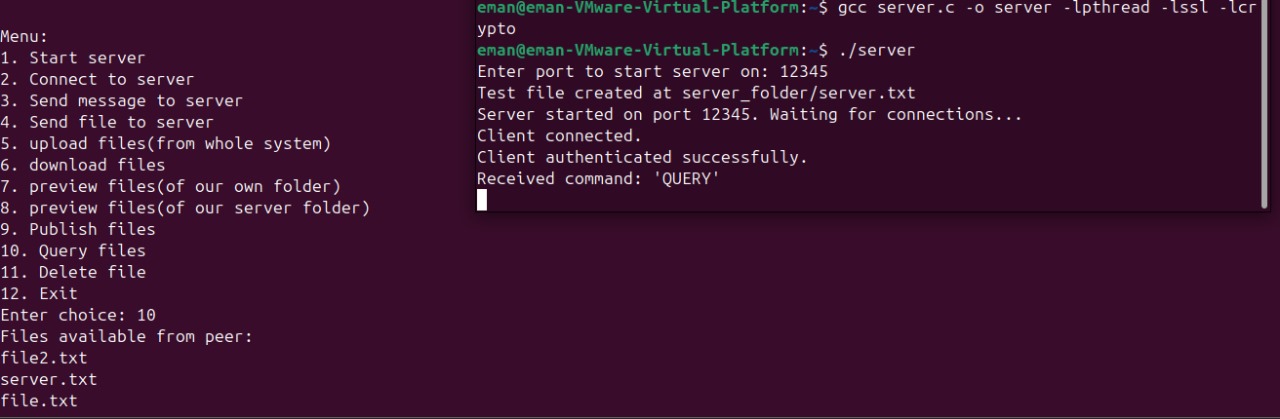


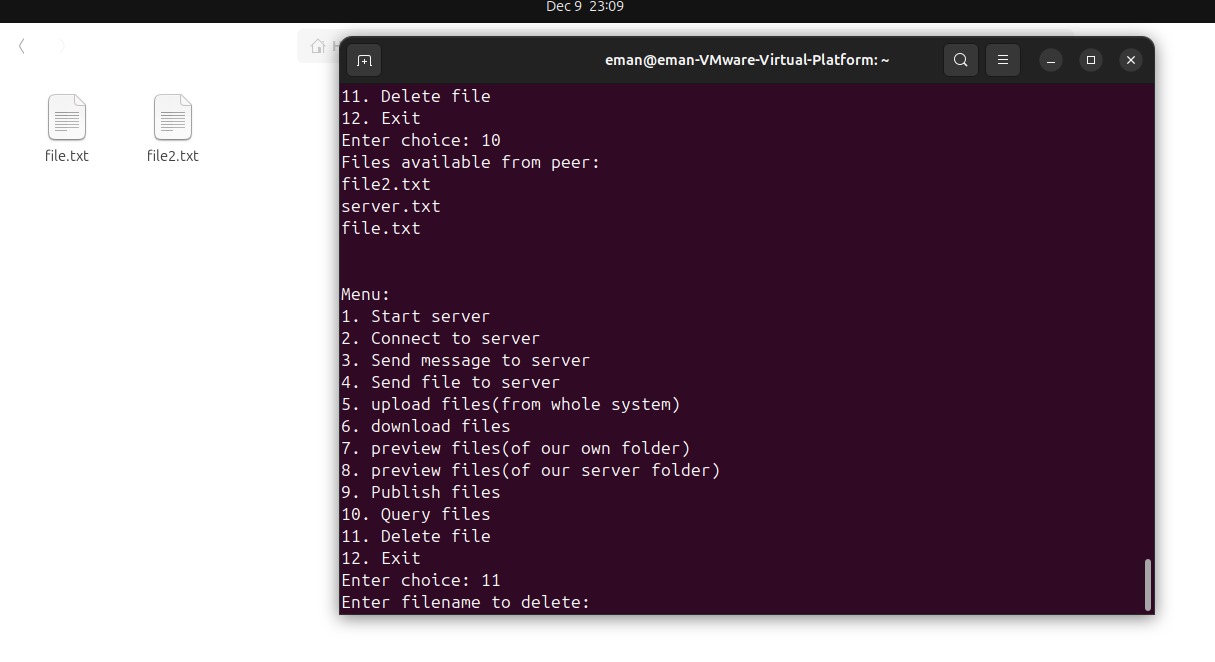


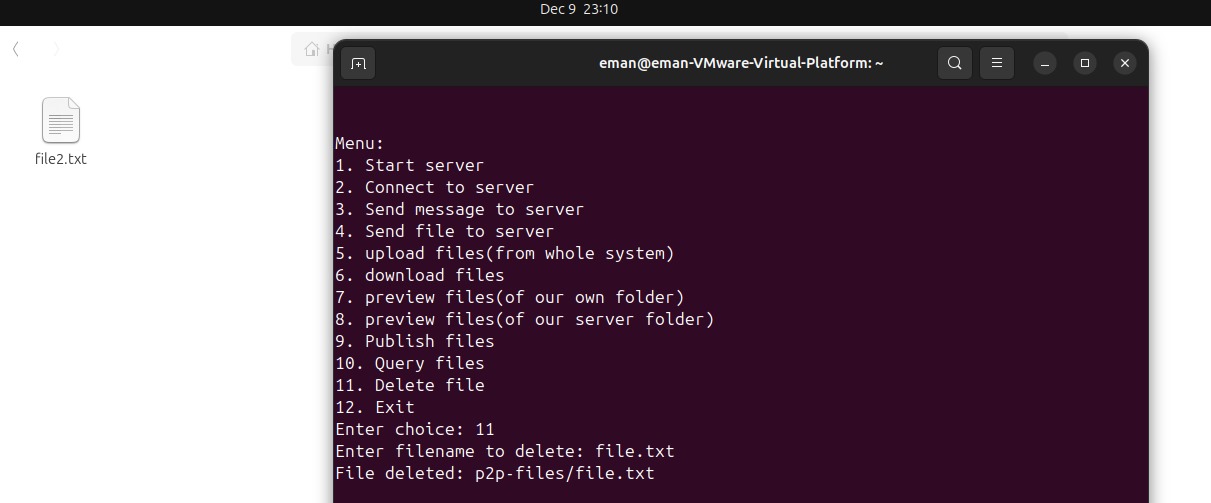


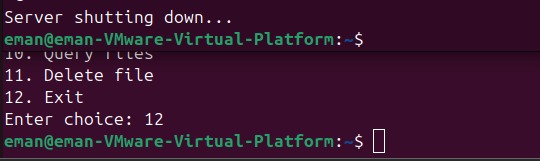












**GUI:**

