Wipro COE Embedded System

PROJECT REPORT

ON

"Network File Sharing Server & Client"

Name	Vivek kumar
Regd.no	2241011211
Batch.no	06

Secure File Transfer System using Socket Programming in C++

Abstract

This project demonstrates a Secure File Transfer System developed using C++ and socket programming. The system establishes a secure connection between a client and a server, allowing encrypted file transfers, authentication, and file management operations. It is implemented for both Windows and Linux platforms, ensuring cross-platform compatibility and robust network performance.

Objective

The objective of this project is to design and develop a secure and efficient communication framework for transferring files between a client and server using socket programming. The system ensures authentication, encrypted data exchange, and platform independence.

System Architecture

The project uses a client-server model where the server listens for connections on a specified port and the client initiates communication. The communication process includes:

1. Establishing a TCP connection.
2. Authenticating the user.
3. Executing file operations (LIST, GET, PUT, QUIT).
4. Encrypting data using an XOR cipher.
5. Handling file I/O with multithreading for concurrent clients.

Windows Implementation

The following code snippet represents the Windows implementation of the server:

Windows Server Code:

```
#ifndef _WIN32_WINNT

#define _WIN32_WINNT 0x0A00

#endif

#define WIN32_LEAN_AND_MEAN

#include <windows.h>

#include <winsock2.h>

#include <ws2tcpip.h>
```

```
#pragma comment(lib, "ws2_32.lib")

#include <iostream>
#include <string>
#include <cstring>
#include <cstddef>
#include <thread>
#include <vector>
#include <fstream>
#include <sstream>
#include <map>
#include <filesystem>
#include <sys/types.h>
#include <sys/stat.h>

#define PORT 8080
#define BUFFER_SIZE 4096

const std::string XOR_KEY = "mysecretkey";

std::string xor_cipher(const std::string &data) {
   std::string output = data; for (std::size_t i = 0;
   i < output.length(); ++i)      output[i] ^=
   XOR_KEY[i % XOR_KEY.length()];    return
   output;
}

std::map<std::string, std::string> users = {
```

```
    {"admin", "pass123"},
    {"Anirban", "getbetter17"}
};

bool authenticate_user(const std::string &auth_string) {
std::stringstream ss(auth_string);     std::string cmd, user, pass;     ss >>
cmd >> user >> pass;     return (cmd == "AUTH" && users.count(user) &&
users[user] == pass);
}

long long get_file_size(const std::string &filename) {
struct _stat64 stat_buf;     int rc =
_stat64(filename.c_str(), &stat_buf);     return rc ==
0 ? stat_buf.st_size : -1;
}

void receive_file(SOCKET sock, const std::string &filename, long long file_size) {
    char buffer[BUFFER_SIZE] = {0}; std::filesystem::create_directory("server_files");

    std::ofstream output_file("server_files/" + filename, std::ios::binary);
  if (!output_file) {       std::string msg = xor_cipher("ERROR: Cannot
  create file.");       send(sock, msg.c_str(), (int)msg.length(), 0);
        return;
    }

    std::string ready = xor_cipher("READY");
send(sock, ready.c_str(), (int)ready.length(), 0);
```

```
   long long bytes_received = 0;     while (bytes_received < file_size) {        int bytes_to_read
= (int)std::min<long long>(BUFFER_SIZE, file_size - bytes_received);       int bytes_read =
recv(sock, buffer, bytes_to_read, 0);       if (bytes_read <= 0)           break;


      output_file.write(buffer, bytes_read);
bytes_received += bytes_read;
   }
   output_file.close();


   std::string response = xor_cipher(bytes_received == file_size
      ? "OK: Upload successful."     : "ERROR: Upload
    failed."); send(sock, response.c_str(),
    (int)response.length(), 0); }


void handle_client(SOCKET client_socket) {
char   buffer[BUFFER_SIZE]   =   {0};    bool
authenticated = false;


   std::cout << "[+] Client connected. Awaiting authentication..." << std::endl;


   while (true) {       memset(buffer, 0, BUFFER_SIZE);       int
bytes_read = recv(client_socket, buffer, BUFFER_SIZE - 1, 0);    if
(bytes_read <= 0) {           std::cout << "[-] Client disconnected." <<
std::endl;
         break;
      }
```

```
        std::string command = xor_cipher(std::string(buffer, bytes_read));
std::cout << "[Client] " << command << std::endl;

        if (!authenticated) {            std::string msg;            if
(authenticate_user(command)) {        authenticated = true;
msg = xor_cipher("OK: Auth successful. Welcome!");
std::cout << "[+] Authentication successful." << std::endl;
          } else {
              msg = xor_cipher("ERROR: Auth failed. Invalid user/pass.");
   std::cout    <<    "[-]    Authentication    failed."    <<    std::endl;
   send(client_socket, msg.c_str(), (int)msg.length(), 0);
              break;
          }
          send(client_socket, msg.c_str(), (int)msg.length(), 0);
continue;
        }

        if (command == "LIST") {
std::filesystem::create_directory("server_files");            std::string list = "Files
on server:\n";            for (auto &entry :
std::filesystem::directory_iterator("server_files"))                list +=
entry.path().filename().string() + "\n";            std::string encrypted_list =
xor_cipher(list);            send(client_socket, encrypted_list.c_str(),
(int)encrypted_list.length(),  0);
        }
```

```
    else if (command.rfind("GET ", 0) == 0) {
std::string filename = command.substr(4);
std::string filepath = "server_files/" + filename;

        if (filename.find("..") != std::string::npos) {       std::string
    msg = xor_cipher("ERROR: Invalid filename.");
    send(client_socket, msg.c_str(), (int)msg.length(), 0);
    continue;

        }

        long long file_size = get_file_size(filepath);          if
(file_size < 0) {              std::string msg = xor_cipher("ERROR:
File not found.");              send(client_socket, msg.c_str(),
(int)msg.length(), 0);
            continue;
        }

        std::string msg = xor_cipher("SIZE " + std::to_string(file_size));
send(client_socket, msg.c_str(), (int)msg.length(), 0);

        memset(buffer,          0,          BUFFER_SIZE);
recv(client_socket, buffer, BUFFER_SIZE - 1, 0);        if
(xor_cipher(std::string(buffer)) != "READY")
            continue;

        std::ifstream file(filepath, std::ios::binary);
while (file) {              file.read(buffer,
BUFFER_SIZE);              std::streamsize bytes =
```

```
file.gcount();                if (bytes > 0)
send(client_socket, buffer, (int)bytes, 0);
            }
            file.close();
        }
        else if (command.rfind("PUT ", 0) == 0) {
std::stringstream ss(command);
std::string cmd, filename;            long long
file_size;            ss >> cmd >> filename >>
file_size;

            if (filename.empty() || file_size <= 0) {                std::string msg
= xor_cipher("ERROR: Bad PUT command.");
send(client_socket, msg.c_str(), (int)msg.length(), 0);
                continue;
            }

            if (filename.find("..") != std::string::npos) {
std::string msg = xor_cipher("ERROR: Invalid filename.");
send(client_socket, msg.c_str(), (int)msg.length(), 0);
                continue;
            }

            receive_file(client_socket, filename, file_size);
        }

        else if (command == "QUIT") {     std::cout << "[x]
        Client exited session." << std::endl;     break;
```

```
        }
        else {                std::string msg = xor_cipher("Unknown
    command.");                    send(client_socket,  msg.c_str(),
    (int)msg.length(), 0);
        }
    }

    closesocket(client_socket);
}

int main() {
    WSADATA wsaData;     if
(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
std::cerr << "WSAStartup failed." << std::endl;
return 1;
    }

    SOCKET server_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);    if
(server_fd == INVALID_SOCKET) {        std::cerr << "Socket creation failed: "
<< WSAGetLastError() << std::endl;        WSACleanup();        return 1;
    }

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

```
  if (bind(server_fd, (sockaddr *)&address, sizeof(address)) == SOCKET_ERROR) {
std::cerr << "Bind failed: " << WSAGetLastError() << std::endl;
closesocket(server_fd);        WSACleanup();        return 1;
  }

  if (listen(server_fd, 5) == SOCKET_ERROR) {        std::cerr <<
"Listen failed: " << WSAGetLastError() << std::endl;
closesocket(server_fd);        WSACleanup();        return 1;
  }

  std::cout << "[SERVER] Secure server listening on port " << PORT << "..." << std::endl;

  while (true) {        sockaddr_in
client_addr{};        int client_addr_len =
sizeof(client_addr);
       SOCKET new_socket = accept(server_fd, (sockaddr *)&client_addr, &client_addr_len);

      if  (new_socket  == INVALID_SOCKET) {            std::cerr << "Accept
   failed: " << WSAGetLastError() << std::endl;    continue;
      }

     std::thread(handle_client, new_socket).detach();
  }

  closesocket(server_fd);
WSACleanup();
  return 0;
}
```

Refer to your provided server.cpp for complete implementation.

The following code snippet represents the Windows implementation of the client:

Windows Client Code:

```
#include <iostream>
#include <string>
#include <cstring>
#include <fstream>
#include <sstream>

#include <winsock2.h>
#include <ws2tcpip.h>
#include <sys/types.h>
#include <sys/stat.h>
#define PORT 8080
#define BUFFER_SIZE 4096

//code

const std::string XOR_KEY = "mysecretkey";

std::string xor_cipher(std::string data) {     std::string output
= data;     for (int i = 0; i < output.length(); ++i) {
output[i] = output[i] ^ XOR_KEY[i % XOR_KEY.length()];
   }
   return output;
}
```

```
long long get_file_size(const std::string &filename) {     std::ifstream
in(filename, std::ifstream::ate | std::ifstream::binary);     if (!in)
return -1;     return (long long)in.tellg();
}

void send_file(SOCKET sock, std::string filename, long long file_size) {
char buffer[BUFFER_SIZE] = {0};

   memset(buffer,  0,  BUFFER_SIZE);    int  bytes_read  =
recv(sock, buffer, BUFFER_SIZE - 1, 0);  if (bytes_read <=
0) {
       std::cerr << "Error: Server disconnected while waiting for READY." << std::endl; return;
  }
  std::string server_response = xor_cipher(std::string(buffer, bytes_read));

   if (server_response != "READY") {
      std::cerr << "Error: Server not ready: " << server_response << ". Aborting upload." <<
std::endl;
      return;
   }

   std::ifstream  file_to_send(filename,  std::ios::binary);            if  (!file_to_send)  {
std::cerr << "Error: Could not open file for reading: " << filename << std::endl;
      return;
   }

   long long bytes_sent = 0;      while (file_to_send) {
file_to_send.read(buffer, BUFFER_SIZE);        std::streamsize
```

```
bytes_actually_read = file_to_send.gcount();        if (bytes_actually_read > 0) {
int sent = send(sock, buffer, (int)bytes_actually_read, 0);            if (sent ==
SOCKET_ERROR) {                std::cerr << "Error sending data: " <<
WSAGetLastError() << std::endl;
            file_to_send.close();
             return;
         }
          bytes_sent += sent;
      }
   }
   file_to_send.close();

   memset(buffer, 0, BUFFER_SIZE);     bytes_read = recv(sock, buffer, BUFFER_SIZE - 1, 0);
if (bytes_read <= 0) {        std::cerr << "Error: Server disconnected while waiting for final
response." << std::endl;
      return;
   }
   std::cout << "Server response: " << xor_cipher(std::string(buffer, bytes_read)) << std::endl;
}

void receive_file(SOCKET sock, std::string filename, long long file_size) {
char buffer[BUFFER_SIZE] = {0};

   std::string encrypted_ready = xor_cipher("READY");     send(sock,
encrypted_ready.c_str(), (int)encrypted_ready.length(), 0);
```

```
  std::ofstream output_file(filename, std::ios::binary);      if (!output_file)
{       std::cerr << "Error: Could not create file " << filename << std::endl;
return; }


   long long bytes_received = 0;      while
(bytes_received < file_size) {        int bytes_to_read =
BUFFER_SIZE;         if (file_size - bytes_received <
BUFFER_SIZE) {            bytes_to_read = (int)(file_size -
bytes_received);
      }


      int bytes_read = recv(sock, buffer, bytes_to_read, 0);        if (bytes_read ==
SOCKET_ERROR) {            std::cerr << "Error receiving data: " <<
WSAGetLastError() << std::endl;
         break;
      }
      if (bytes_read == 0) {
         std::cerr << "Error: Server disconnected during file transfer unexpectedly." <<
std::endl;
         break;
      }


      output_file.write(buffer, bytes_read);
bytes_received += bytes_read;
  }
  output_file.close();
   if (bytes_received == file_size) {     std::cout << "Download
   complete: " << filename << std::endl;
   } else {
```

```
        std::cout << "Download failed. Received " << bytes_received << " of " << file_size << "
bytes." << std::endl;
    }
}

bool InitializeWinsock() {     WSADATA wsaData;     if
(WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
std::cerr << "WSAStartup failed.\n";         return false;
    }
    return true;
}

void CleanupWinsock() {
    WSACleanup();
}

int main() {    if
(!InitializeWinsock()) {
return 1;
   }
    SOCKET sock = INVALID_SOCKET;
    struct     sockaddr_in     serv_addr;
    char buffer[BUFFER_SIZE] = {0};

    if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == INVALID_SOCKET) {
std::cerr  <<  "\n  Socket  creation  error:  "  <<  WSAGetLastError()  <<  std::endl;
CleanupWinsock();
        return -1;
    }
```

```
   memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family            =            AF_INET;
serv_addr.sin_port = htons(PORT);

   unsigned long ip_address = inet_addr("127.0.0.1");    if
(ip_address == INADDR_NONE) {   std::cerr << "\nInvalid
address/ Address not supported \n";   closesocket(sock);
CleanupWinsock();
      return -1;
   }
   serv_addr.sin_addr.s_addr = ip_address;

   if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) == SOCKET_ERROR) {
std::cerr << "\nConnection Failed: " << WSAGetLastError() << std::endl;
closesocket(sock);

      CleanupWinsock();
      return -1;
   }

   std::string user, pass, auth_string;     std::cout << "Connected to
secure server. Please log in." << std::endl;    std::cout << "Username: ";
std::getline(std::cin, user);    std::cout << "Password: ";
std::getline(std::cin, pass);

   auth_string = "AUTH " + user + " " + pass;
```

```
    std::string encrypted_auth = xor_cipher(auth_string);     send(sock,
encrypted_auth.c_str(), (int)encrypted_auth.length(), 0);

    memset(buffer,  0,  BUFFER_SIZE);    int  bytes_read  =
recv(sock, buffer, BUFFER_SIZE - 1, 0);   if (bytes_read ==
SOCKET_ERROR || bytes_read == 0) {
        std::cerr  <<  (bytes_read  ==  SOCKET_ERROR  ?  "Auth  response  error:  "  +
std::to_string(WSAGetLastError()) : "Server disconnected during authentication.") <<
std::endl;       closesocket(sock);       CleanupWinsock();
        return -1;
    }

    std::string auth_response = xor_cipher(std::string(buffer, bytes_read));

    if (auth_response.rfind("ERROR:", 0) == 0) {
std::cerr << auth_response << std::endl;
closesocket(sock);         CleanupWinsock();
        return -1;
    }

    std::cout << auth_response << std::endl;

    std::cout << "Type 'LIST', 'GET <file>', 'PUT <file>', or 'QUIT'." << std::endl;

    while  (true)  {                  std::string
user_input;              std::cout  <<  "> ";
std::getline(std::cin, user_input);

        if (user_input.empty()) continue;
```

```
    if (user_input.rfind("PUT ", 0) == 0) {            std::string filename =
user_input.substr(4);            long long file_size = get_file_size(filename);
if (file_size < 0) {              std::cout << "Error: File not found or cannot be
read." << std::endl;
            continue;
        }

        std::string command_to_send = "PUT " + filename + " " + std::to_string(file_size);
std::string encrypted_command = xor_cipher(command_to_send);          send(sock,
encrypted_command.c_str(), (int)encrypted_command.length(), 0);

        std::cout << "Uploading " << filename << " (" << file_size << " bytes)..." << std::endl;
send_file(sock, filename, file_size);
    }
else {
        std::string encrypted_input = xor_cipher(user_input);
send(sock, encrypted_input.c_str(), (int)encrypted_input.length(), 0);

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

        if (user_input.rfind("GET ", 0) == 0) {
std::string filename = user_input.substr(4);

            memset(buffer, 0, BUFFER_SIZE);
bytes_read = recv(sock, buffer, BUFFER_SIZE - 1, 0);
```

```
            if (bytes_read == SOCKET_ERROR || bytes_read == 0) {                std::cout <<
"Server disconnected or error during GET response." << std::endl;
                break;
            }

            std::string server_response = xor_cipher(std::string(buffer, bytes_read));

            if (server_response.rfind("SIZE ", 0) == 0) {                long
long file_size = std::stoll(server_response.substr(5));
                std::cout << "Receiving " << filename << " (" << file_size << " bytes)..." <<
std::endl;                receive_file(sock, filename, file_size);

            } else {
                std::cout << "Server response: " << server_response << std::endl;
            }
}           else
{
            memset(buffer, 0, BUFFER_SIZE);
bytes_read = recv(sock, buffer, BUFFER_SIZE - 1, 0);

            if (bytes_read <= 0) {                std::cout <<
"Server disconnected." << std::endl;
                break;
            }
            std::cout << "Server response:\n" << xor_cipher(std::string(buffer, bytes_read)) <<
std::endl;
        }
    }
```

```
    }
    closesocket(sock);
    CleanupWinsock();
    return 0;
}
```

Refer to your provided client.cpp for complete implementation.

Linux Implementation

Below is the Linux-compatible version of the server program:

Linux Client Code:

```
// --- client/client.cpp (Phase 5 FIXED) ---
#include <iostream>
#include <string>
#include <cstring>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <fstream>
#include <sys/stat.h>
#include <sstream>

#define PORT 8080
#define BUFFER_SIZE 4096

// --- Encryption --- const std::string
XOR_KEY = "mysecretkey"; std::string
xor_cipher(std::string data) {     std::string
output = data;     for (int i = 0; i <
```

```
output.length(); ++i) {        output[i] =
output[i] ^ XOR_KEY[i %
XOR_KEY.length()];
   }
   return output;
}

long get_file_size(std::string filename) {
struct stat stat_buf;     int rc =
stat(filename.c_str(), &stat_buf);     return
rc == 0 ? stat_buf.st_size : -1;
}

void send_file(int sock, std::string filename, long file_size) {
char buffer[BUFFER_SIZE] = {0};

   memset(buffer, 0, BUFFER_SIZE);     int bytes_read = recv(sock, buffer,
BUFFER_SIZE - 1, 0);     std::string server_response =
xor_cipher(std::string(buffer, bytes_read));

   if (server_response != "READY") {        std::cerr << "Error: Server not
ready. Aborting upload." << std::endl;
      return;
   }
   std::ifstream file_to_send(filename, std::ios::binary);     while
(file_to_send) {        file_to_send.read(buffer, BUFFER_SIZE);
std::streamsize bytes_actually_read = file_to_send.gcount();
```

```
if (bytes_actually_read > 0) {            send(sock, buffer,
bytes_actually_read, 0);
      }
   }
   file_to_send.close();

   memset(buffer, 0, BUFFER_SIZE);     bytes_read =
recv(sock, buffer, BUFFER_SIZE - 1, 0);
   std::cout << "Server response: " << xor_cipher(std::string(buffer, bytes_read)) << std::endl;
}

void receive_file(int sock, std::string filename, long file_size) {
char buffer[BUFFER_SIZE] = {0};

   std::string encrypted_ready = xor_cipher("READY");     send(sock,
encrypted_ready.c_str(), encrypted_ready.length(), 0);

   std::ofstream output_file(filename, std::ios::binary);     if (!output_file)
{       std::cerr << "Error: Could not create file " << filename << std::endl;
return;
   }

   long bytes_received = 0;     while
(bytes_received < file_size) {         int bytes_to_read
= BUFFER_SIZE;         if (file_size - bytes_received <
BUFFER_SIZE) {            bytes_to_read = file_size -
bytes_received;
      }
```

```
      int bytes_read = recv(sock, buffer, bytes_to_read, 0);         if (bytes_read <= 0)
{           std::cerr << "Error: Server disconnected during file transfer." <<
std::endl;
          break;
      }
      output_file.write(buffer, bytes_read);
bytes_received += bytes_read;
   }
   output_file.close();

   if (bytes_received == file_size) {        std::cout << "Download
complete: " << filename << std::endl;
   } else {        std::cout << "Download failed." <<
std::endl;
   }
}

int main() {     int sock = 0;     struct
sockaddr_in serv_addr;     char
buffer[BUFFER_SIZE] = {0};

   if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
std::cerr << "\n Socket creation error \n";      return -
1;
   }
```

```
   memset(&serv_addr, '0', sizeof(serv_addr));
serv_addr.sin_family              =              AF_INET;
serv_addr.sin_port = htons(PORT);

   if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
std::cerr << "\nInvalid address/ Address not supported \n";
      return -1;
   }

   if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
std::cerr << "\nConnection Failed \n";       return -1;
   }

   // --- Login Prompt ---
   std::string user, pass, auth_string;     std::cout << "Connected to
secure server. Please log in." << std::endl;    std::cout << "Username: ";
std::getline(std::cin, user);     std::cout << "Password: ";
std::getline(std::cin,  pass);

   auth_string = "AUTH " + user + " " + pass;

   std::string encrypted_auth = xor_cipher(auth_string);
send(sock, encrypted_auth.c_str(), encrypted_auth.length(), 0);

   memset(buffer, 0, BUFFER_SIZE);     int bytes_read = recv(sock, buffer,
BUFFER_SIZE - 1, 0);     std::string auth_response =
xor_cipher(std::string(buffer, bytes_read));
```

```
  if (auth_response.rfind("ERROR:", 0) == 0) {
std::cerr << auth_response << std::endl;
close(sock);        return -1;
  }

  std::cout << auth_response << std::endl;

  std::cout << "Type 'LIST', 'GET <file>', 'PUT <file>', or 'QUIT'." << std::endl;            while (true) {
std::string user_input;        std::cout << "> ";        std::getline(std::cin, user_input);

    if (user_input.empty()) continue;

    if (user_input.rfind("PUT ", 0) == 0) {            std::string filename =
user_input.substr(4);            long file_size = get_file_size(filename);            if
(file_size < 0) {                std::cout << "Error: File not found or cannot be
read." << std::endl;
            continue;
        }

        std::string command_to_send = "PUT " + filename + " " + std::to_string(file_size);
std::string encrypted_command = xor_cipher(command_to_send);                send(sock,
encrypted_command.c_str(), encrypted_command.length(), 0);

        std::cout << "Uploading " << filename << " (" << file_size << " bytes)..." << std::endl;
send_file(sock, filename, file_size);
    }
else {
```

```
        std::string encrypted_input = xor_cipher(user_input);
send(sock, encrypted_input.c_str(), encrypted_input.length(), 0);    if
(user_input == "QUIT") {
            break;
        }

        if (user_input.rfind("GET ", 0) == 0) {
std::string filename = user_input.substr(4);

            memset(buffer, 0, BUFFER_SIZE);
bytes_read = recv(sock, buffer, BUFFER_SIZE - 1, 0);

            std::string server_response = xor_cipher(std::string(buffer, bytes_read));

            if (server_response.rfind("SIZE ", 0) == 0) {
long file_size = std::stol(server_response.substr(5));
                std::cout << "Receiving " << filename << " (" << file_size << " bytes)..." <<
std::endl;                  receive_file(sock, filename, file_size);

            } else {
                std::cout << "Server response: " << server_response << std::endl;
            }
}           else
{
            memset(buffer, 0, BUFFER_SIZE);
bytes_read = recv(sock, buffer, BUFFER_SIZE - 1, 0);
if (bytes_read <= 0) {                  std::cout << "Server
disconnected." << std::endl;
```

```
            break;
        }
        std::cout << "Server response:\n" << xor_cipher(std::string(buffer, bytes_read)) << std::endl;
      }
    }
  }

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

Refer to the generated server_linux.cpp for the complete source code.

The client code for Linux can be adapted similarly, replacing Winsock APIs with POSIX socket calls.

Linux Server Code:

```
// --- server/server.cpp (Phase 5 FIXED) ---
#include <iostream>
#include <string>
#include <cstring>
#include <unistd.h>
#include <sys/socket.h>
#include  <netinet/in.h>
#include <thread>
#include <vector>
#include <dirent.h>
#include <fstream>
#include <sys/stat.h>
#include <sstream>
#include <map>
```

```
#define PORT 8080
#define BUFFER_SIZE 4096

// --- Encryption --- const std::string
XOR_KEY = "mysecretkey";

std::string xor_cipher(std::string data) {     std::string output
= data;     for (int i = 0; i < output.length(); ++i) {
output[i] = output[i] ^ XOR_KEY[i % XOR_KEY.length()];
   }
   return output;
}

// --- User Authentication --- std::map<std::string,
std::string> users = {
   {"admin", "password"},
   {"vivek", "wipro2025"}
};

bool authenticate_user(std::string auth_string) {     std::stringstream ss(auth_string);
std::string cmd, user, pass;     ss >> cmd >> user >> pass;

   if (cmd == "AUTH" && users.count(user) && users[user] == pass) {
      return true;
   }
   return false;
}
```

```
long get_file_size(std::string filename) {
struct stat stat_buf;     int rc =
stat(filename.c_str(), &stat_buf);     return
rc == 0 ? stat_buf.st_size : -1;
}

void receive_file(int sock, std::string filename, long file_size) {
char buffer[BUFFER_SIZE] = {0};

   std::ofstream output_file("./server_files/" + filename, std::ios::binary);
if (!output_file) {        std::string msg = "ERROR: Cannot create file.";
std::string encrypted_msg = xor_cipher(msg);        send(sock,
encrypted_msg.c_str(), encrypted_msg.length(), 0);
      return;
   }

   std::string encrypted_ready = xor_cipher("READY");     send(sock,
encrypted_ready.c_str(), encrypted_ready.length(), 0);

   long bytes_received = 0;     while
(bytes_received < file_size) {        int bytes_to_read
= BUFFER_SIZE;        if (file_size - bytes_received <
BUFFER_SIZE) {           bytes_to_read = file_size -
bytes_received;
      }

      int bytes_read = recv(sock, buffer, bytes_to_read, 0);
if (bytes_read <= 0) {           break;
```

```
        }
        output_file.write(buffer, bytes_read);
    bytes_received += bytes_read;
      }
      output_file.close();

      std::string response_msg;     if
    (bytes_received == file_size) {
    response_msg = "OK: Upload successful.";
      } else {
        response_msg = "ERROR: Upload failed.";
      }
      std::string encrypted_response = xor_cipher(response_msg);
    send(sock, encrypted_response.c_str(), encrypted_response.length(), 0);
    }

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

      std::cout << "Client connected. Waiting for authentication..." << std::endl;

      while (true) {        memset(buffer, 0, BUFFER_SIZE);        int
    bytes_read = recv(client_socket, buffer, BUFFER_SIZE - 1, 0);       if
    (bytes_read <= 0) {           std::cout << "Client disconnected." <<
    std::endl;
          break;
        }
```

```
    std::string raw_data(buffer, bytes_read);       std::string command
= xor_cipher(raw_data);       std::cout << "Client says (decrypted): "
<< command << std::endl;


    if (!authenticated) {
std::string msg;
        if (authenticate_user(command)) { authenticated
            = true;

            std::cout << "Authentication successful." << std::endl;
msg = "OK: Auth successful. Welcome!";
        } else {
            std::cout << "Authentication failed." << std::endl;            msg =
"ERROR: Auth failed. Invalid user/pass.";            std::string encrypted_msg =
xor_cipher(msg);            send(client_socket, encrypted_msg.c_str(),
encrypted_msg.length(), 0);
            break;
        }
        std::string encrypted_msg = xor_cipher(msg);
send(client_socket, encrypted_msg.c_str(), encrypted_msg.length(), 0);
        continue;
    }


    if (command == "LIST") {          std::string file_list = "Files on server:\n";
DIR *dir;          struct dirent *ent;          if ((dir = opendir("./server_files")) !=
NULL) {            while ((ent = readdir(dir)) != NULL) {                if
(strcmp(ent->d_name, ".") != 0 && strcmp(ent->d_name, "..") != 0) {
file_list += ent->d_name;                    file_list += "\n";
```

```
            }
        }
        closedir(dir);
    } else {
        file_list = "Error: Cannot open server_files directory.";
    }
    std::string encrypted_list = xor_cipher(file_list);
send(client_socket, encrypted_list.c_str(), encrypted_list.length(), 0);

} else if (command.rfind("GET ", 0) == 0) {
std::string filename = command.substr(4);
std::string filepath = "./server_files/" + filename;
std::string msg;

    if (filename.find("..") != std::string::npos) {        msg = "ERROR:
Invalid filename.";        std::string encrypted_msg = xor_cipher(msg);
send(client_socket, encrypted_msg.c_str(), encrypted_msg.length(), 0);
        continue;
    }

    long file_size = get_file_size(filepath);

    if (file_size < 0) {        msg = "ERROR: File not found.";
std::string encrypted_msg = xor_cipher(msg);        send(client_socket,
encrypted_msg.c_str(), encrypted_msg.length(), 0);
    } else { msg = "SIZE " +
        std::to_string(file_size);
        std::string encrypted_msg =
```

```
            xor_cipher(msg);
            send(client_socket,
            encrypted_msg.c_str(),
            encrypted_msg.length(), 0);

            memset(buffer,              0,              BUFFER_SIZE);
recv(client_socket, buffer, BUFFER_SIZE - 1, 0);              if
(xor_cipher(std::string(buffer)) != "READY") {
                continue;
            }

            std::ifstream file_to_send(filepath, std::ios::binary);
while (file_to_send) {                  file_to_send.read(buffer,
BUFFER_SIZE);                  std::streamsize bytes_actually_read =
file_to_send.gcount();                  if (bytes_actually_read > 0) {
send(client_socket, buffer, bytes_actually_read, 0);
                }
            }
            file_to_send.close();
        }
    }
    else if (command.rfind("PUT ", 0) == 0) {
std::stringstream ss(command);
std::string cmd, filename;            long
file_size;            ss >> cmd >> filename >>
file_size;            std::string msg;
```

```
        if (filename.empty() || file_size <= 0) {               msg = "ERROR: Bad PUT
command.";              std::string encrypted_msg = xor_cipher(msg);
send(client_socket, encrypted_msg.c_str(), encrypted_msg.length(), 0);
            continue;
        }
        if (filename.find("..") != std::string::npos) {             msg = "ERROR:
Invalid filename.";             std::string encrypted_msg = xor_cipher(msg);
send(client_socket, encrypted_msg.c_str(), encrypted_msg.length(), 0);
            continue;
        }

        receive_file(client_socket, filename, file_size);
    }
    else if (command == "QUIT") {
        break;
    } else {
        std::string msg = "Unknown command.";          std::string
encrypted_msg = xor_cipher(msg);          send(client_socket,
encrypted_msg.c_str(), encrypted_msg.length(), 0);
    }
  }
  close(client_socket);
}

// --- Main Server Loop (No changes) ---
int main() {    int server_fd;    struct
sockaddr_in address;    int opt = 1;
```

```
   if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
perror("socket failed");        exit(EXIT_FAILURE);
   }

   if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))) {
perror("setsockopt");         exit(EXIT_FAILURE);
   }

   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");        exit(EXIT_FAILURE);
   }

   if (listen(server_fd, 5) < 0) {
perror("listen");         exit(EXIT_FAILURE);
   }

   std::cout << "Secure server is listening on port " << PORT << "..." << std::endl;

   while (true) {        int new_socket;        struct
sockaddr_in client_addr;        socklen_t
client_addr_len = sizeof(client_addr);

      if ((new_socket = accept(server_fd, (struct sockaddr *)&client_addr, &client_addr_len))
< 0) {             perror("accept");             std::cerr <<
"Error on accept" << std::endl;
```

```
        } else {
            std::thread(handle_client, new_socket).detach();
        }
    }

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

Compilation & Execution Steps

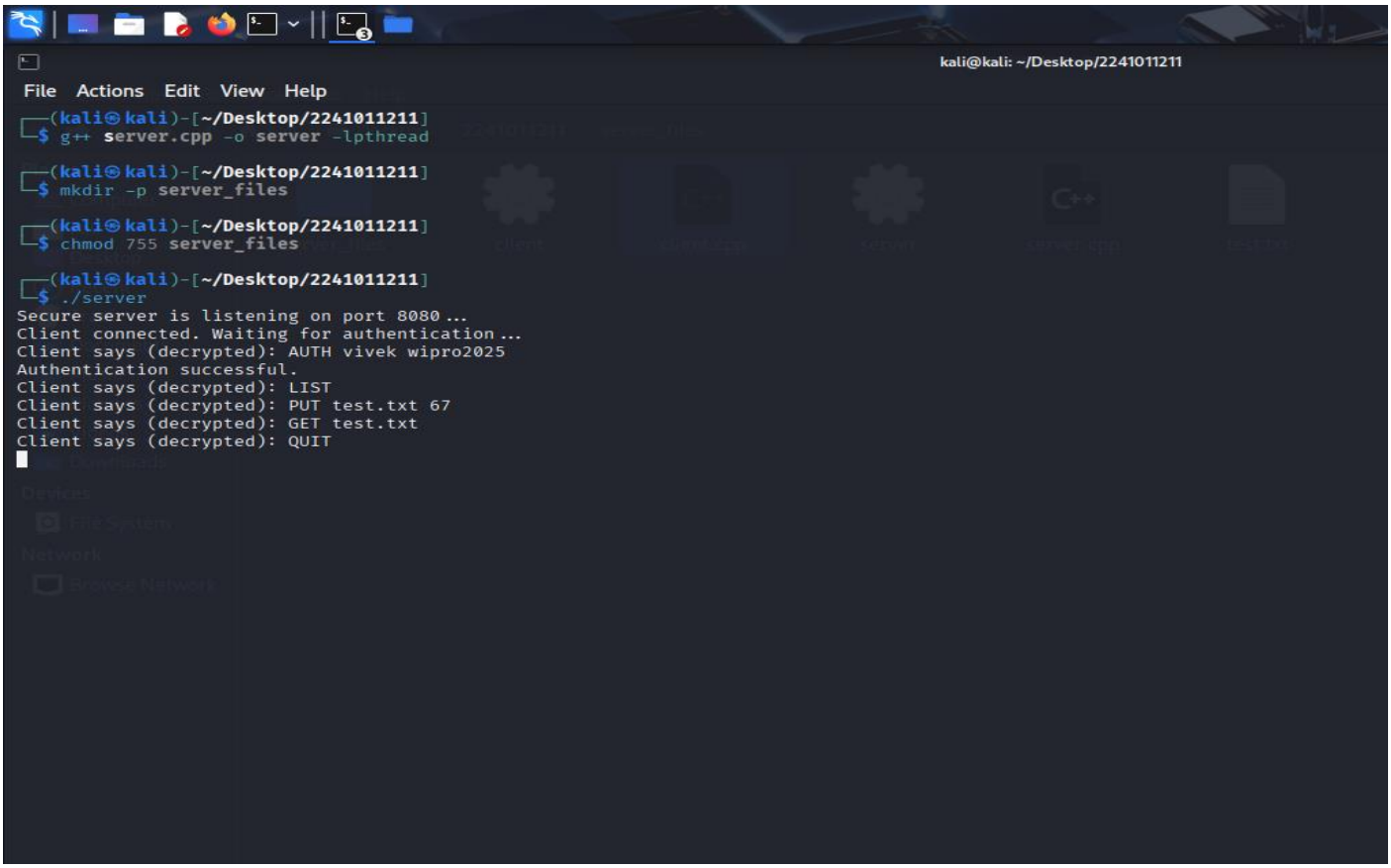
For Windows:
1. Open VS Code and navigate to your server/client directory.
2. Compile using:
   g++ server.cpp -o server.exe -std=c++17 -lws2_32 -pthread
g++ client.cpp -o client.exe -std=c++17 -lws2_32 -pthread
3. Run the server first, then the client.

For Linux:
1. Open terminal and navigate to the respective folder.
2. Compile using:
   g++ server_linux.cpp -o server -std=c++17 -pthread 3.
Run the server with ./server and client with ./client.

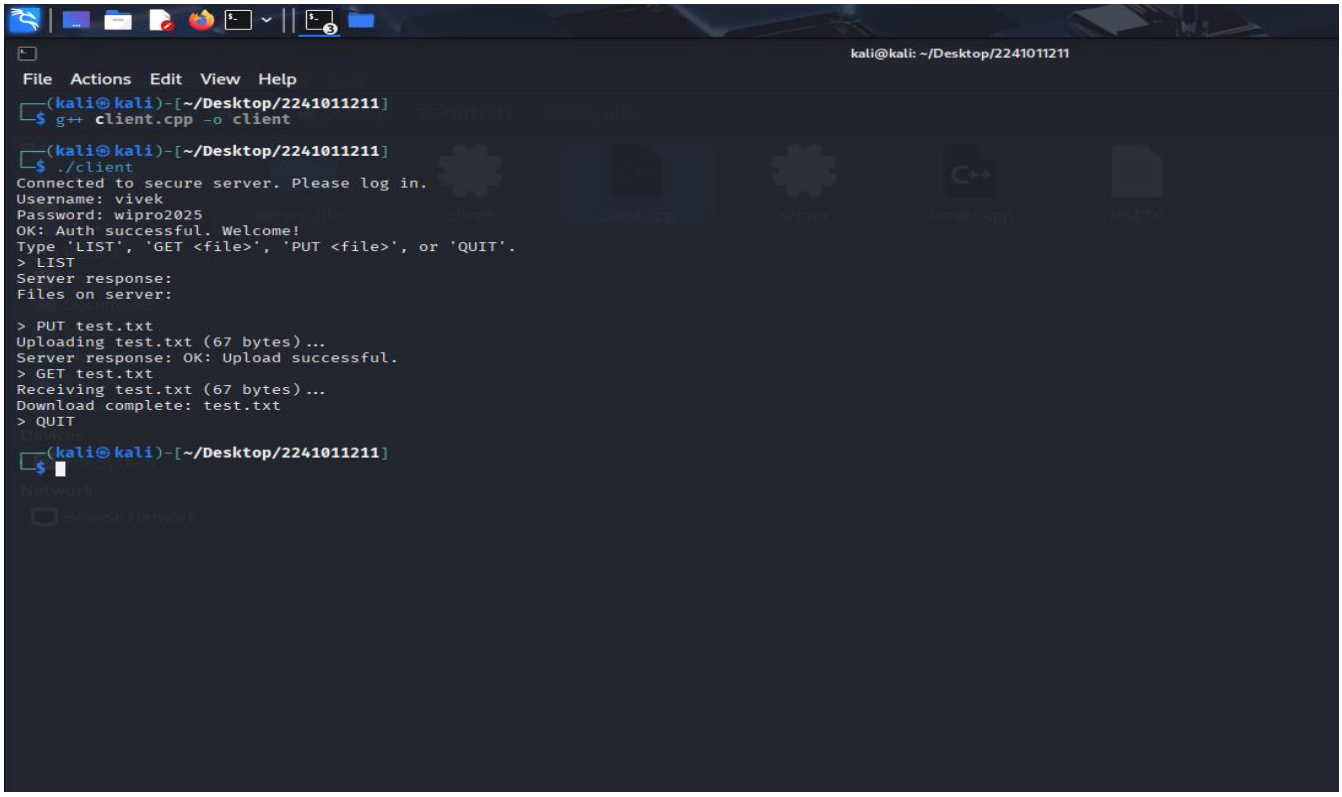
Screenshots

The following screenshots demonstrate successful compilation, server-client connection, and file transfer:

For Server :



For Client :


kali@kali: ~/Desktop/2241011211
File Actions Edit View Help
(kali㉿kali)-[~/Desktop/2241011211]
$ g++ client.cpp -o client
(kali㉿kali)-[~/Desktop/2241011211]
$ ./client
Connected to secure server. Please log in.
Username: vivek
Password: wipro2025
OK: Auth successful. Welcome!
Type 'LIST', 'GET <file>', 'PUT <file>', or 'QUIT'.
> LIST
Server response:
Files on server:
> PUT test.txt
Uploading test.txt (67 bytes)...
Server response: OK: Upload successful.
> GET test.txt
Receiving test.txt (67 bytes)...
Download complete: test.txt
> QUIT
(kali㉿kali)-[~/Desktop/2241011211]
$

Conclusion

The Secure File Transfer System effectively demonstrates socket programming principles, file handling, and cross-platform development in C++. The project ensures encrypted communication and user authentication, making it suitable for secure network data exchange. Future enhancements may include SSL integration, GUI-based file management, and improved encryption algorithms.