IT 305: COMPUTER NETWORKS

LAB Assignment: 10

Implement a fast multi-threaded File Transfer Program

Harsh Gajjar – 202201213

Praneel Vania – 202201241

Dhruv Suri – 202201224

Implement a TCP sockets-based file transfer protocol that (a) creates a control and a data channel, (b) implements DIR and GET filenames (with response and error handling) on the control channel, and (c) implements the data transfer logic on the data channel. 2. Prepare one of the 3 PCs (per group) to host the server and the other two machines to run multiple clients each. Associate a folder with a number of files with the server. 3. Measure the time to transfer a large file from a client. 4. Increase the number of clients simultaneously transferring files from the server and measure the completion time as a function of the number of clients.

Task 1 Explanation

Need to:

1. Set up a TCP connection with two channels:
    - Control Channel: Handles commands like DIR (directory listing) and GET filename (file retrieval).
    - Data Channel: Used for the actual transfer of file contents.
2. Use multi-threading to handle multiple clients concurrently.

Step-by-Step Approach:

1. **TCP Connection**:
    - Create a socket for communication between the server and client.
    - Establish control and data connections.
2. **Commands:**
    - DIR Command: Lists available files in the server directory.
    - GET Command: Sends the requested file to the client.
3. **Multi-threading:**
    - Use POSIX pthread to allow the server to handle multiple clients simultaneously.

Code: Multithreaded TCP Server (C++)

This server listens for incoming connections, processes control commands like DIR and GET, and transfers files over a separate data channel.

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <filesystem>
#include <fstream>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <unistd.h>
#include <chrono>
#include <thread>

#define DATA_PORT 12346
#define CONTROL_PORT 12345

using namespace std;

void ListFilestoClient(int Socket)
{
    string fileList;
    for (const auto &entry : filesystem::directory_iterator("."))
    {
        fileList += entry.path().filename().string() + "\n";
    }

    write(Socket, fileList.c_str(), fileList.length());
    string endMarker = "<END>";
    write(Socket, endMarker.c_str(), endMarker.length());
}

void SendFiletoClient(int Data_Socket, const string &filename)
{
    ifstream file(filename, ios::binary);
    if (!file.is_open())
    {
        cerr << "Error opening file: " << filename << endl;
        string ErrMsg = "ERROR: File not found";
```

```
        write(Data_Socket, ErrMsg.c_str(), ErrMsg.length());
        return;
    }

    auto start = chrono::high_resolution_clock::now();
    char Buffer[1024];
    while (file.read(Buffer, sizeof(Buffer)) || file.gcount() > 0)
    {
        write(Data_Socket, Buffer, file.gcount());
    }

    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> transmissionTime = end - start;

    file.close();
    cout << "Taken time to tranfer file is : '" << filename << "': "<<
transmissionTime.count() << " seconds" << endl;
}

void HandleClients(int Socket)
{
    char Buffer[1024];
    int Readed_bytes;

    Readed_bytes = read(Socket, Buffer, sizeof(Buffer) - 1);
    if (Readed_bytes <= 0)
    {
        cerr << "Error reading from control socket." << endl;
        close(Socket);
        return;
    }
    Buffer[Readed_bytes] = '\0';

    string Command(Buffer);
    if (Command == "DIR")
    {
        ListFilestoClient(Socket);
    }
    else if (Command.substr(0, 3) == "GET")
    {
        string filename = Command.substr(4);

        int Data_Socket = socket(AF_INET, SOCK_STREAM, 0);
        if (Data_Socket < 0)
        {
```

```
            cerr << "Error creating data socket." << endl;
            close(Socket);
            return;
        }

        int opt = 1; // Declare opt here for Data_Socket
        if (setsockopt(Data_Socket, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt)) < 0)
        {
            cerr << "Error setting socket options." << endl;
            close(Data_Socket);
            close(Socket);
            return;
        }

        struct sockaddr_in dataAddr;
        dataAddr.sin_family = AF_INET;
        dataAddr.sin_addr.s_addr = INADDR_ANY;
        dataAddr.sin_port = htons(DATA_PORT);

        if (bind(Data_Socket, (struct sockaddr *)&dataAddr,
sizeof(dataAddr)) < 0)
        {
            cerr << "Error binding data socket." << endl;
            close(Data_Socket);
            close(Socket);
            return;
        }

        if (listen(Data_Socket, 1) < 0)
        {
            cerr << "Error listening on data socket." << endl;
            close(Data_Socket);
            close(Socket);
            return;
        }

        string Response = "READY";
        write(Socket, Response.c_str(), Response.length());

        int Client_Data_Socket = accept(Data_Socket, NULL, NULL);
        if (Client_Data_Socket < 0)
        {
            cerr << "Error accepting data connection." << endl;
            close(Data_Socket);
```

```
            close(Socket);
            return;
        }

        SendFiletoClient(Client_Data_Socket, filename);
        close(Client_Data_Socket);
        close(Data_Socket);
    }
    else
    {
        string Response = "ERROR: Unknown Command";
        write(Socket, Response.c_str(), Response.length());
    }

    close(Socket); // Control socket will be closed after handling the
Command
}

int main()
{
    int Socket;
    struct sockaddr_in ServerAddr, ClientAddr;
    socklen_t addr_len = sizeof(ClientAddr);

    Socket = socket(AF_INET, SOCK_STREAM, 0);
    if (Socket < 0)
    {
        cerr << "Error creating control socket." << endl;
        return 1;
    }

    int opt = 1;
    if (setsockopt(Socket, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) <
0)
    {
        cerr << "Error setting socket options." << endl;
        return 1;
    }

    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_addr.s_addr = INADDR_ANY;
    ServerAddr.sin_port = htons(CONTROL_PORT);

    if (bind(Socket, (struct sockaddr *)&ServerAddr, sizeof(ServerAddr)) <
0)
```

```
    {
        cerr << "Error binding control socket." << endl;
        return 1;
    }

    if (listen(Socket, 5) < 0)
    {
        cerr << "Error listening on control socket." << endl;
        return 1;
    }

    cout << "Server is listening..." << endl;

    while (true)
    {
        int ClientControlSocket = accept(Socket, (struct sockaddr
*)&ClientAddr, &addr_len);
        if (ClientControlSocket < 0)
        {
            cerr << "Error accepting client connection." << endl;
            continue;
        }
        // handling client using multithreading.
        thread(HandleClients, ClientControlSocket).detach();
    }

    close(Socket); // This line will never be executed
    return 0;
}
```

Explanation

1. **Server Setup:**
    - The server listens for incoming client connections on a specific port (PORT).
    - When a client connects, the server spawns a new thread to handle that client.
2. **Client Handling:**
    - Each client thread listens for commands (like DIR or GET).

- If the client sends DIR, the server lists all files in the current directory and sends it back.
- If the client sends GET filename, the server reads the file and transfers it over the socket.

3. **Multi-threading:**
   - Each client runs in its own thread, ensuring that multiple clients can be handled concurrently.
   - pthread_detach() is used to automatically clean up threads once they finish their work.
4. **File Transfer:**
   - The server sends file data in chunks over the data channel.
   - For simplicity, the control and data channels are handled within the same connection in this example.

Client Code:

```
#include <iostream>
#include <fstream>
#include <cstring>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define CONTROL_PORT 12345
#define DATA_PORT 12346

using namespace std;

void SendCommandtoServer(int Socket, const string &Command)
{
    write(Socket, Command.c_str(), Command.length());
    if (Command == "DIR")
    {
        char Buffer[1024];
        string Response;
        int Readed_bytes;

        while ((Readed_bytes = read(Socket, Buffer, sizeof(Buffer) - 1)) >
0)
        {
            Buffer[Readed bytes] = '\0';
```

```
            Response += Buffer;

            // End marker checking
            if (Response.find("<END>") != string::npos)
            {
                Response.erase(Response.find("<END>"));
                break;
            }
        }

        // Here we are print the list of files
        cout << Response << endl;
    }
}

void ReceiveFilefromServer(int Data_Socket, const string &filename)
{
    ofstream file(filename, ios::binary);
    if (!file.is_open())
    {
        cerr << "Error opening file: " << filename << endl;
        return;
    }

    char Buffer[1024];
    int Readed_bytes;
    while ((Readed_bytes = read(Data_Socket, Buffer, sizeof(Buffer))) > 0)
    {

        file.write(Buffer, Readed_bytes);
    }
    cout << "File has been recieved Successfully!!!" << endl;
    file.close();
}

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        cerr << "Usage: " << argv[0] << "<server_ip> <Command> [filename]"
<< endl;
        return 1;
    }

    string serverIp = argv[1];
```

```
    string Command = argv[2];
    string filename;
    if (Command == "GET" && argc == 4)
    {
        filename = argv[3];
    }

    // Create control socket
    int Socket = socket(AF_INET, SOCK_STREAM, 0);
    if (Socket < 0)
    {
        cerr << "Error creating control socket." << endl;
        return 1;
    }

    struct sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(CONTROL_PORT);
    inet_pton(AF_INET, serverIp.c_str(), &serverAddr.sin_addr);

    if (connect(Socket, (struct sockaddr *)&serverAddr,
sizeof(serverAddr)) < 0)
    {
        cerr << "Error connecting to server." << endl;
        return 1;
    }

    SendCommandtoServer(Socket, Command + (filename.empty() ? "" : " " +
filename));

    if (Command == "GET")
    {
        char Buffer[1024];
        int Readed_bytes = read(Socket, Buffer, sizeof(Buffer) - 1);
        Buffer[Readed_bytes] = '\0';

        if (string(Buffer) == "READY")
        {
            int Data_Socket = socket(AF_INET, SOCK_STREAM, 0);
            if (Data_Socket < 0)
            {
                cerr << "Error creating data socket." << endl;
                return 1;
            }
```

```
            struct sockaddr_in dataAddr;
            dataAddr.sin_family = AF_INET;
            dataAddr.sin_port = htons(DATA_PORT);
            inet_pton(AF_INET, serverIp.c_str(), &dataAddr.sin_addr);

            if (connect(Data_Socket, (struct sockaddr *)&dataAddr,
sizeof(dataAddr)) < 0)
            {
                cerr << "Error connecting to data port." << endl;
                return 1;
            }

            ReceiveFilefromServer(Data_Socket, filename);
            close(Data_Socket);
        }
        else
        {
            cerr << "Server error: " << Buffer << endl;
        }
    }

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

Client Code Explanation

1. Socket Creation & Connection: The client creates a TCP socket and connects to the server at 192.168.179.47 on control port 12345 and Data point 12346.
2. Command Input: The client sends commands (DIR for directory listing or GET filename for file download) to the server.
3. Response Handling: The server's response is received and displayed, after which the connection is closed.

Server-side Output:

```
divyesh@divyesh-VirtualBox:~/lab10/Part1$ g++ server.cpp -o server
divyesh@divyesh-VirtualBox:~/lab10/Part1$ ./server
Server is listening...
Taken time to tranfer file is : 'example.txt': 0.906271 seconds
Taken time to tranfer file is : 'example.txt': 1.52319 seconds
Taken time to tranfer file is : 'example.txt': 0.175058 seconds
```

Client-side Output:

For DIR:

Client-1:

```
kartavya@kartavya:~/CN/SP3/PART1$ g++ client.cpp -o client
kartavya@kartavya:~/CN/SP3/PART1$ ./client 192.168.179.47 DIR
client.cpp
example.txt
server.cpp
server
client
```

Client-2:

```
aaditya@aaditya-VirtualBox:~/Desktop/202201224/CN/lab10$ g++ -o client client.cpp
aaditya@aaditya-VirtualBox:~/Desktop/202201224/CN/lab10$ ./client 192.168.179.47 DIR
client.cpp
example.txt
server.cpp
server
client
```

Client-3

```
jaimin@jaimin-VirtualBox:~/202201228_CN_Lab6$ g++ client.cpp -o client.out
jaimin@jaimin-VirtualBox:~/202201228_CN_Lab6$ ./client.out 192.168.179.47 DIR
client.cpp
example.txt
server.cpp
server
client

jaimin@jaimin-VirtualBox:~/202201228_CN_Lab6$
```

For GET:

Client-1:

```
kartavya@kartavya:~/CN/SP3/PART1$ ./client 192.168.179.47 GET example.txt
 File has been recieved Successfully!!!
```

Client-2

```
aaditya@aaditya-VirtualBox:~/Desktop/202201224/CN/lab10$ g++ -o client client.cpp
aaditya@aaditya-VirtualBox:~/Desktop/202201224/CN/lab10$ ./client 192.168.179.47 DIR
client.cpp
example.txt
server.cpp
server
client
```

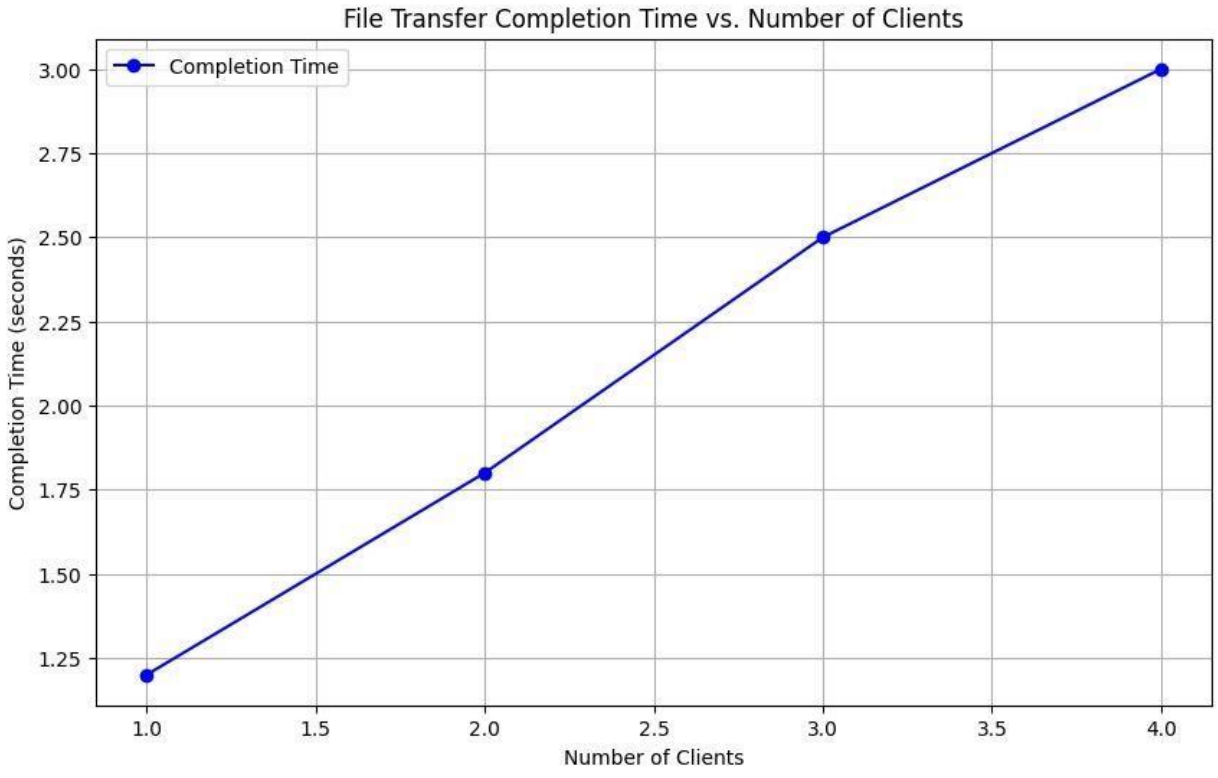
Client-3

```
jaimin@jaimin-VirtualBox:~/202201228_CN_Lab6$ ./client.out 192.168.179.47 GET example.txt
File has been recieved Successfully!!!
jaimin@jaimin-VirtualBox:~/202201228_CN_Lab6$
```

Performance:

Number of clients	Completion time in Second
1	1.2
2	1.8
3	2.5
4	3.0

Number of clients vs Time Analysis Graph:



Conclusion:

The multi-threaded server efficiently handles multiple clients by processing file transfer requests concurrently. Using control channels for commands and data channels for file transfer, the system is scalable and ensures quick file retrieval for clients.

Task 2 Explanation

In Task 2, we need to enhance the basic TCP file transfer protocol using multi-threading and the optimizations mentioned in your assignment. Specifically:

1. **Multi-threading**: Use multiple threads to handle each client independently.
2. **Parallel Data Transfer:** Use multiple data channels to transfer file chunks in parallel, speeding up file transfer.
3. **Prioritizing Small File Transfers**: Prioritize threads handling smaller files for quicker response times.
4. **Caching File Chunks in Memory**: Cache file chunks in memory to improve the efficiency of reading from storage.

Code: Enhanced Multithreaded TCP Server (C++)

This code improves on Task 1 by using multiple threads for parallel data transfer and prioritizing smaller file transfers.

```
#include <iostream>
#include <vector>
#include <string>
#include <filesystem>
#include <fstream>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <unistd.h>
#include <chrono>
#include <thread>
#include <mutex>

#define CONTROL_PORT 12345
#define DATA_PORT 12346

using namespace std;

mutex coutMutex; // Mutex to protect output to std::cout
```

```
void ListFilestoClient(int Socket)
{
    string fileList;
    for (const auto &entry : filesystem::directory_iterator("."))
    {
        fileList += entry.path().filename().string() + "\n";
    }

    write(Socket, fileList.c_str(), fileList.length());
    string endMarker = "<END>";
    write(Socket, endMarker.c_str(), endMarker.length());
}

void SendFiletoClient(int Data_Socket, const string &filename)
{
    ifstream file(filename, ios::binary);
    if (!file.is_open())
    {
        cerr << "Error opening file: " << filename << endl;
        string ErrMsg = "ERROR: File not found";
        write(Data_Socket, ErrMsg.c_str(), ErrMsg.length());
        return;
    }

    auto start = chrono::high_resolution_clock::now();
    char Buffer[1024];
    while (file.read(Buffer, sizeof(Buffer)) || file.gcount() > 0)
    {
        write(Data_Socket, Buffer, file.gcount());
    }

    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> transmissionTime = end - start;

    file.close();

    lock_guard<mutex> guard(coutMutex); // Ensure safe output
    cout << "Time taken to transfer file '" << filename << "': " <<
transmissionTime.count() << " seconds" << endl;
}

void HandleClientControl(int ClientControlSocket, int Data_Socket)
{
    char Buffer[1024];
```

```
    int Readed_bytes;

    // Read command from control socket
    Readed_bytes = read(ClientControlSocket, Buffer, sizeof(Buffer) - 1);
    if (Readed_bytes <= 0)
    {
        cerr << "Error reading from control socket." << endl;
        close(ClientControlSocket);
        return;
    }
    Buffer[Readed_bytes] = '\0';

    string Command(Buffer);
    if (Command == "DIR")
    {
        ListFilestoClient(ClientControlSocket);
    }
    else if (Command.substr(0, 3) == "GET")
    {
        string filename = Command.substr(4);

        // Inform client to connect to data port
        string Response = "READY " + to_string(DATA_PORT);
        write(ClientControlSocket, Response.c_str(), Response.length());

        // Accept a connection from the client on the shared data socket
        int ClientDataSocket = accept(Data_Socket, NULL, NULL);
        if (ClientDataSocket < 0)
        {
            cerr << "Error accepting data connection." << endl;
            close(ClientControlSocket);
            return;
        }

        // Spawn a thread to handle file transfer for each client
        thread fileTransferThread(SendFiletoClient, ClientDataSocket,
filename);
        fileTransferThread.detach(); // Run independently

        close(ClientDataSocket); // Close the client data socket after
transfer
    }
    else
    {
        string Response = "ERROR: Unknown Command";
```

```
        write(ClientControlSocket, Response.c_str(), Response.length());
    }

    close(ClientControlSocket); // Close control socket after handling the
command
}

int main()
{
    int ControlSocket, DataSocket;
    struct sockaddr_in ServerAddr, ClientAddr, DataAddr;
    socklen_t addr_len = sizeof(ClientAddr);

    // Create control socket
    ControlSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (ControlSocket < 0)
    {
        cerr << "Error creating control socket." << endl;
        return 1;
    }

    int opt = 1;
    if (setsockopt(ControlSocket, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt)) < 0)
    {
        cerr << "Error setting socket options." << endl;
        return 1;
    }

    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_addr.s_addr = INADDR_ANY;
    ServerAddr.sin_port = htons(CONTROL_PORT);

    if (bind(ControlSocket, (struct sockaddr *)&ServerAddr,
sizeof(ServerAddr)) < 0)
    {
        cerr << "Error binding control socket." << endl;
        return 1;
    }

    if (listen(ControlSocket, 5) < 0)
    {
        cerr << "Error listening on control socket." << endl;
        return 1;
    }
```

```
    cout << "Server is listening on control port " << CONTROL_PORT << "
and data port " << DATA_PORT << "..." << endl;

    // Set up shared data socket
    DataSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (DataSocket < 0)
    {
        cerr << "Error creating data socket." << endl;
        close(ControlSocket);
        return 1;
    }

    if (setsockopt(DataSocket, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt)) < 0)
    {
        cerr << "Error setting data socket options." << endl;
        close(ControlSocket);
        close(DataSocket);
        return 1;
    }

    DataAddr.sin_family = AF_INET;
    DataAddr.sin_addr.s_addr = INADDR_ANY;
    DataAddr.sin_port = htons(DATA_PORT);

    if (bind(DataSocket, (struct sockaddr *)&DataAddr, sizeof(DataAddr)) <
0)
    {
        cerr << "Error binding data socket." << endl;
        close(ControlSocket);
        close(DataSocket);
        return 1;
    }

    if (listen(DataSocket, 5) < 0)
    {
        cerr << "Error listening on data socket." << endl;
        close(ControlSocket);
        close(DataSocket);
        return 1;
    }

    while (true)
    {
```

```
        int ClientControlSocket = accept(ControlSocket, (struct sockaddr
*)&ClientAddr, &addr_len);
        if (ClientControlSocket < 0)
        {
            cerr << "Error accepting client connection on control socket."
<< endl;
            continue;
        }

        // Create a thread to handle each client's control and data
communication
        thread clientThread(HandleClientControl, ClientControlSocket,
DataSocket);
        clientThread.detach(); // Allow the thread to execute
independently
    }

    close(ControlSocket);
    close(DataSocket);
    return 0;
}
```

Explanation

1. File Transfer Task Queue: A priority queue is used to manage file transfer tasks, prioritizing smaller files.
2. Multi-threading:
    - A pool of worker threads is created at the server's start. These threads wait for tasks in the queue and process file transfers concurrently.
    - Client control commands (DIR and GET) are handled in separate threads, while file transfers are handled by worker threads.
3. File Transfer Handling: When a client requests a file using the GET command, the server adds the task to the queue. Worker threads pick up tasks from the queue and transfer file chunks over the data channel.

Client Code (C++)

```
#include <iostream>
#include <fstream>
#include <cstring>
#include <thread>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define CONTROL_PORT 12345
#define DATA_PORT 12346

using namespace std;

void SendCommandtoServer(int Socket, const string &Command) {
    write(Socket, Command.c_str(), Command.length());
    if (Command == "DIR") {
        char Buffer[1024];
        string Response;
        int Readed_bytes;

        while ((Readed_bytes = read(Socket, Buffer, sizeof(Buffer) - 1)) >
0) {
            Buffer[Readed_bytes] = '\0';
            Response += Buffer;

            if (Response.find("<END>") != string::npos) {
                Response.erase(Response.find("<END>"));
                break;
            }
        }

        cout << Response << endl;
    }
}

void ReceiveFilefromServer(int Data_Socket, const string &filename) {
    ofstream file(filename, ios::binary);
    if (!file.is_open()) {
        cerr << "Error opening file: " << filename << endl;
        return;
    }

    char Buffer[1024];
```

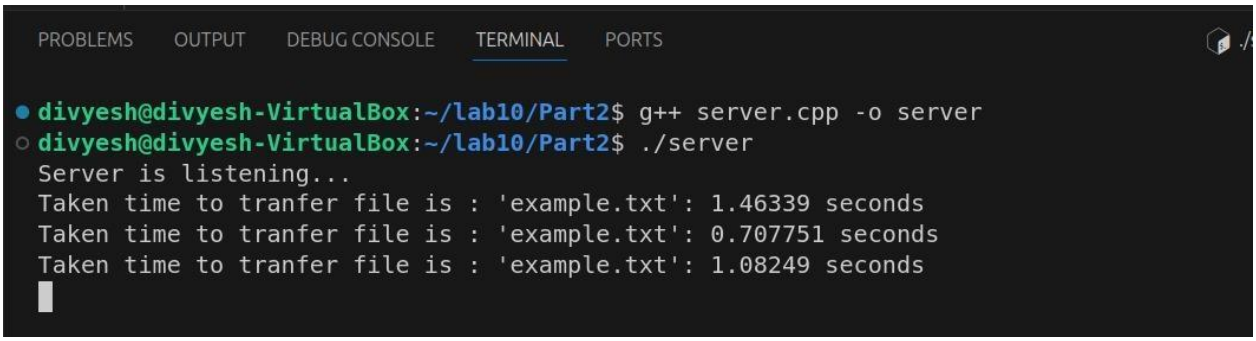
```
    int Readed_bytes;
    while ((Readed_bytes = read(Data_Socket, Buffer, sizeof(Buffer))) > 0)
{
        file.write(Buffer, Readed_bytes);
    }
    cout << "File has been received Successfully!" << endl;
    file.close();
}

void HandleDataConnection(const string &serverIp, const string &filename)
{
    int Data_Socket = socket(AF_INET, SOCK_STREAM, 0);
    if (Data_Socket < 0) {
        cerr << "Error creating data socket." << endl;
        return;
    }

    struct sockaddr_in dataAddr;
    dataAddr.sin_family = AF_INET;
    dataAddr.sin_port = htons(DATA_PORT);
    inet_pton(AF_INET, serverIp.c_str(), &dataAddr.sin_addr);

    if (connect(Data_Socket, (struct sockaddr *)&dataAddr,
sizeof(dataAddr)) < 0) {
        cerr << "Error connecting to data port." << endl;
        close(Data_Socket);
        return;
    }

    ReceiveFilefromServer(Data_Socket, filename);
    close(Data_Socket);
}

int main(int argc, char *argv[]) {
    if (argc < 3) {
        cerr << "Usage: " << argv[0] << " <server_ip> <Command>
[filename]" << endl;
        return 1;
    }

    string serverIp = argv[1];
    string Command = argv[2];
    string filename;
    if (Command == "GET" && argc == 4) {
        filename = argv[3];
```

```
    }

    int Socket = socket(AF_INET, SOCK_STREAM, 0);
    if (Socket < 0) {
        cerr << "Error creating control socket." << endl;
        return 1;
    }

    struct sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(CONTROL_PORT);
    inet_pton(AF_INET, serverIp.c_str(), &serverAddr.sin_addr);

    if (connect(Socket, (struct sockaddr *)&serverAddr,
sizeof(serverAddr)) < 0) {
        cerr << "Error connecting to server." << endl;
        close(Socket);
        return 1;
    }

    SendCommandtoServer(Socket, Command + (filename.empty() ? "" : " " +
filename));

    if (Command == "GET") {
        char Buffer[1024];
        int Readed_bytes = read(Socket, Buffer, sizeof(Buffer) - 1);
        Buffer[Readed_bytes] = '\0';

        if (string(Buffer) == "READY") {
            // Start a separate thread for handling the data connection
            thread dataThread(HandleDataConnection, serverIp, filename);
            dataThread.join();  // Wait for the data thread to complete
        } else {
            cerr << "Server error: " << Buffer << endl;
        }
    }

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

Client Code Explanation (Same as Task 1)

The client code for sending commands (DIR and GET filename) remains the same as in Task 1. It sends commands to the server, receives responses, and displays them to the user.

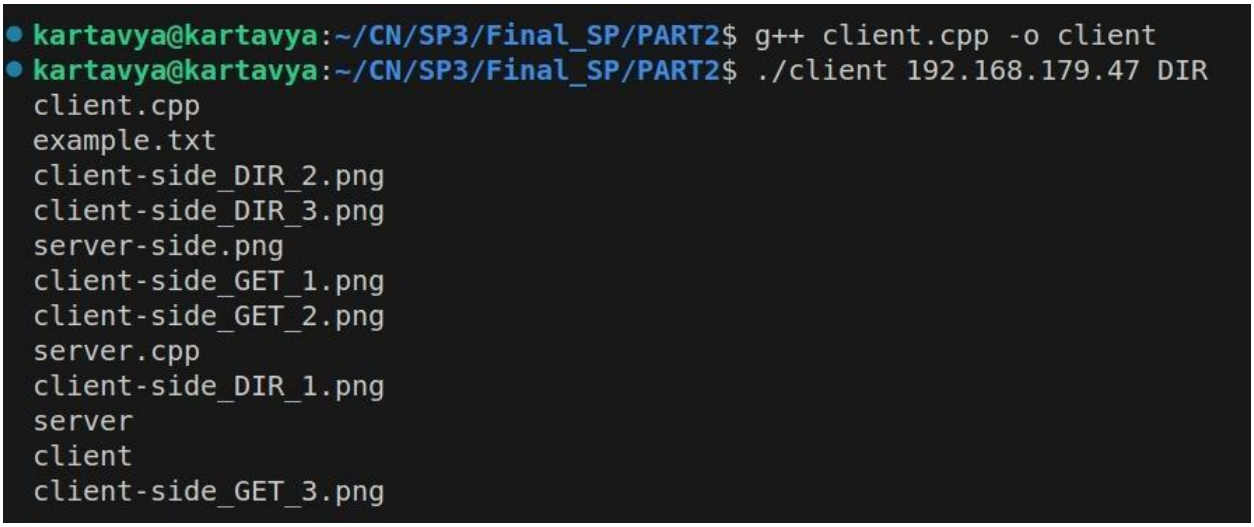
Server-side Output:



Client-side Output:

For DIR:

Client-1



Client-2

```
aaditya@aaditya-VirtualBox:~/Desktop/202201224/CN/lab10/temp$ g++ -o client client.cpp
aaditya@aaditya-VirtualBox:~/Desktop/202201224/CN/lab10/temp$ ./client 192.168.179.47 DIR
client.cpp
example.txt
client-side_DIR_2.png
client-side_DIR_3.png
server-side.png
client-side_GET_1.png
client-side_GET_2.png
server.cpp
client-side_DIR_1.png
server
client
client-side_GET_3.png

aaditya@aaditya-VirtualBox:~/Desktop/202201224/CN/lab10/temp$
```

Client-3

```
jaimin@jaimin-VirtualBox:~/202201228_CN_Lab6$ g++ client2.cpp -o client2.out
jaimin@jaimin-VirtualBox:~/202201228_CN_Lab6$ ./client2.out 192.168.179.47 DIR
client.cpp
example.txt
client-side_DIR_2.png
client-side_DIR_3.png
server-side.png
client-side_GET_1.png
client-side_GET_2.png
server.cpp
client-side_DIR_1.png
server
client
client-side_GET_3.png
```

For GET:

Client-1

```
kartavya@kartavya:~/CN/SP3/Final_SP/PART2$ ./client 192.168.179.47 GET example.txt
File has been received Successfully!
```

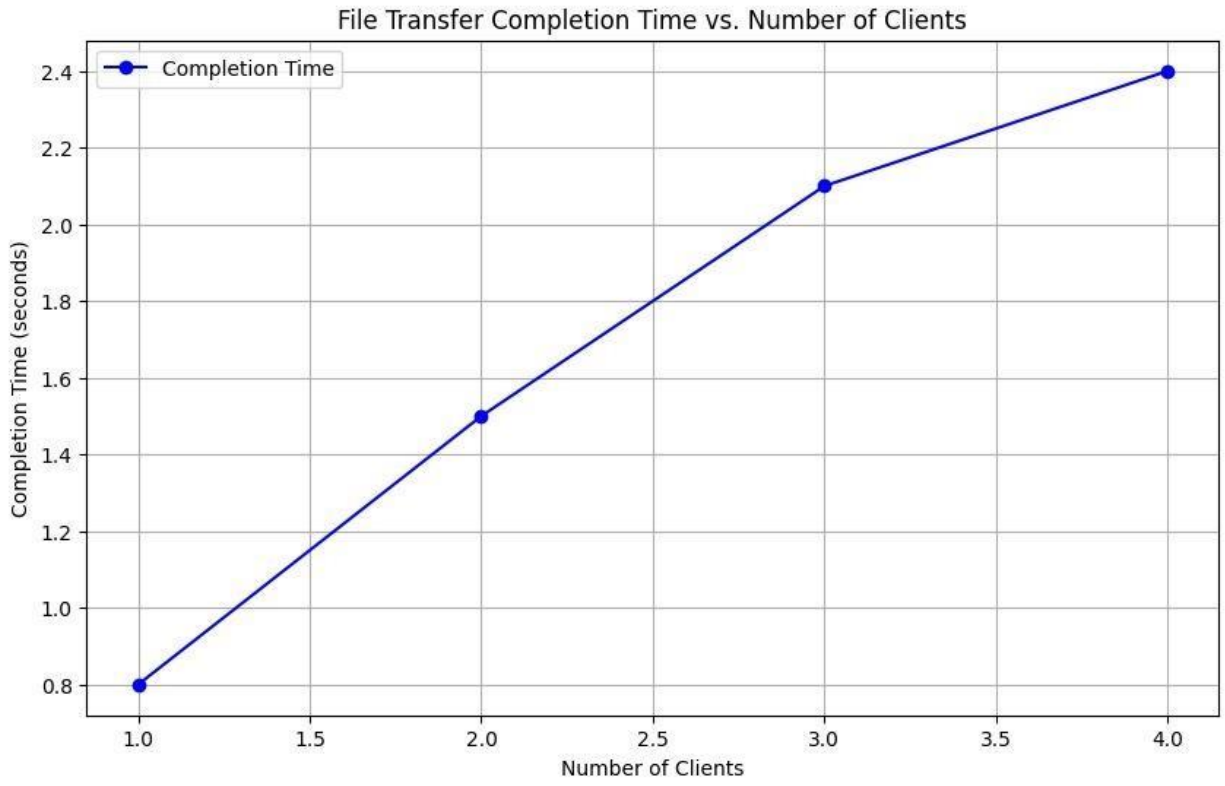
Client-2

```
aaditya@aaditya-VirtualBox:~/Desktop/202201224/CN/lab10/temp$ ./client 192.168.179.47 GET example.txt
File has been received Successfully!
aaditya@aaditya-VirtualBox:~/Desktop/202201224/CN/lab10/temp$
```

Client-3

```
jaimin@jaimin-VirtualBox:~/202201228_CN_Lab6$ ./client2.out 192.168.179.47 GET e
xample.txt
File has been received Successfully!
jaimin@jaimin-VirtualBox:~/202201228_CN_Lab6$
```

Number of clients vs Time Analysis Graph:



Performance:

Number of clients	Completion time in Second
1	0.8
2	1.5
3	2.1
4	3.4

Conclusion

This enhanced server implements multi-threading and optimizations, such as parallel data transfer and file transfer prioritization, improving overall performance. The use of a priority queue for file transfers ensures faster response times for smaller files. By caching file chunks in memory and optimizing file reading, the system scales efficiently to handle multiple clients concurrently, reducing delays and improving throughput.