FILE TRANSFER PROTOCOL

Submitted by

AARYAN PUROHIT (RA2311003020812)

Under the guidance of

Ms. S. KAMALESWARI

(Assistant Professor, Department of Computer Science and Engineering)

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

RAMAPURAM , CHENNAI-600089

OCTOBER 2025

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Deemed to be University Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that the Project titled "Project title____" is the bonafide certificate of Name and Reg no **AARYAN PUROHIT(RA2311003020812)** of III Year CSE submitted for the course 21CSC302J COMPUTER NETWORKS for the Academic Year 2025 – 2026 Odd Semester.

SIGNATURE

Ms. S. KAMALESWARI Assistant Professor

Computer Science & Engineering

SRM Institute of Science and Technology

Ramapuram, Chennai.

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

RAMAPURAM, CHENNAI

DECLARATION

I here by declare that the entire work contained in this project report titled '**TITLE**' has been carried out **AARYAN PUROHIT(RA2311003020812**) at SRM Institute of Science and Technology, Ramapuram, Chennai, under the guidance of **Ms. S. KAMALESWARI.** Assistant professor, Department of Computer Science and Engineering.

STUDENT NAMES

PLACE: CHENNAI
DATE:

Aaryan Purohit

ABSTRACT

File Transfer Protocol (FTP) plays a fundamental role in network-based file sharing by enabling users to upload, download, and manage files over a reliable communication channel. This project presents a simplified FTP Client–Server application developed in Python to demonstrate the core principles of network programming, socket communication, and concurrent client handling. The system consists of two primary components: ftpserver and ftpclient. The server program listens on a specified TCP port and supports multiple concurrent connections using threading, allowing several clients to interact simultaneously. The client program connects to the server via its address and port number, authenticates the user through a username and password prompt, and provides an interactive command-line interface for file operations. Once authenticated, the client can issue three primary commands dir to retrieve and display the list of available files on the server, get <filename> to download a file from the server, and upload <filename> to transfer a file from the client to the server. Both command and data transmissions are handled using TCP sockets, ensuring reliable delivery and synchronization. Although the implementation does not strictly conform to the standard FTP protocol, it effectively demonstrates essential concepts of client–server architecture, TCP communication, and file transfer mechanisms in computer networks. This project serves as an educational tool that bridges theoretical networking principles with practical implementation, offering users hands-on experience in designing and managing basic file transfer systems over a network.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
KEYWORDS

- File Transfer Protocol (FTP)
- Client-Server Architecture
- Socket Programming
- Transmission Control Protocol (TCP)
- Multi-threading
- Network Security
- Authentication
- Data Connection
- Control Connection

CHAPTER 2
INTRODUCTION

1. INTRODUCTION

Computer networks are the foundation of modern communication, allowing data and information to be shared quickly and reliably between connected systems. One of the most common and important uses of networks is file transfer. The ability to send and receive files across devices is essential for sharing resources, performing backups, and collaborating efficiently. The File Transfer Protocol (FTP) was one of the earliest methods developed for this purpose and remains a fundamental concept in computer networking. Although the official FTP standard includes many advanced features, the core idea is simple: establishing a connection between a client and a server so that files can be exchanged securely and reliably.

This project focuses on creating a simplified version of an FTP system that demonstrates the key principles of network communication and socket programming. The system is made up of two programs, ftpserver and ftpclient, which work together to transfer files using the Transmission Control Protocol (TCP). The server acts as the central host, listening for incoming client connections, verifying user credentials, and handling file requests. The client provides a simple interface for users to connect to the server, log in, and perform file operations such as listing available files, downloading them, or uploading new ones.

The ftpserver program is started first and listens for connections on a specific TCP port. When a client connects, the server establishes a communication link that allows both sides to exchange commands and data. The ftpclient program can run on the same computer as the server or on another machine in the same network. To connect, the user provides the server's address and port number, for example, ftpclient sand.cise.ufl.edu 5106. Once connected, the client prompts the user to enter a username and password. After successful authentication, the user can issue three simple commands to interact with the server. The dir command retrieves and displays a list of files stored on the server. The get <filename> command allows the user to download a file from the server, while the upload <filename> command sends a local file to the server. These commands cover the basic functionality needed for transferring files between two computers over a network.

Communication between the client and server is handled through Python's socket library, which uses TCP to ensure reliable delivery of data. TCP guarantees that information is sent and received in order and without loss, which is critical for maintaining file integrity. The project also allows flexibility in how communication is managed: both commands and file data can share the same connection, or they can use separate ones. To make the system more efficient, the server supports multiple clients at once by using threads. Each client is handled independently, which means several users can perform file transfers simultaneously without affecting one another.

Developing this project provided valuable hands-on experience with several important networking concepts. It required an understanding of how socket connections are established, how data is transmitted and received, and how concurrency allows multiple users to interact with the server at the same time. The project also highlights how authentication and error handling can be incorporated into network applications to make them more secure and reliable. By working on this system, it becomes easier to connect the theory of computer networking with real-world programming practice.

Although this implementation is a simplified version of FTP, it captures the essential features that make the protocol work. It does not include advanced functionalities such as encryption, complex directory management, or multiple transfer modes, since the main goal is to focus on the core communication process. Despite its simplicity, the project successfully demonstrates how files can be transferred over a network using sockets and how clients and servers interact in a typical file-sharing setup.

In summary, this project offers a clear and practical understanding of how data transfer works in a networked environment. By building a basic FTP client and server, it shows how reliable communication can be achieved using TCP and how multiple users can share resources efficiently. The project not only provides a useful learning experience but also serves as a solid foundation for exploring more advanced networking concepts and real-world applications in distributed computing and internet communication.

2. PROBLEM STATEMENT:

The core engineering challenge is to develop a robust, custom File Transfer Protocol (FTP) client/server system using the fundamental primitives of network socket programming. This system must be segmented into two distinct, interoperable executables: the passive ftpserver and the active ftpclient. The project is not merely a file copying utility; it is a demonstration of reliable, concurrent network service provision.

Server Requirements: The ftpserver must operate as a perpetual service, initializing a listening TCP socket on a specific, user-defined port (e.g., port 5106). The server must be designed with a multi-threaded architecture. This is a non-negotiable functional requirement: upon accepting an incoming client connection, the server must delegate the entire client session—from authentication to command processing and data transfer—to a newly spawned worker thread. This threading model is essential to ensure that the process of handling one client's potentially time-consuming file transfer (I/O-bound operation) does not block the main listener thread from accepting new connections, guaranteeing high concurrency and service availability for all users. The server must manage a designated shared file directory, confining all file operations (dir, get, upload) strictly within this path for security and isolation.

Client Requirements: The ftpclient must establish a connection by taking the server's network address and port number as command-line arguments (e.g., ftpclient <server_ip> <port>). Upon successful connection, the client must immediately engage in a custom authentication handshake, requiring the user to provide valid credentials (username and password). Following successful logon, the client must present a command-line interface (ftp>) supporting three critical functions:

1.dir: A request to retrieve the contents of the server's shared directory. This requires the server to list files and send the textual output reliably to the client.

2.get <filename>: A request to download a specific file. The system must use reliable TCP streaming to ensure the file bytes are transferred in order and without corruption, even across large distances.

3.upload <filename>: A request to transmit a local file from the client to the server's shared directory. The client must read its local file and stream the data to the server, which must handle the safe creation and writing of the remote file.

CHAPTER 3
LITERATURE SURVEY

3.1 Foundational Network Models:

The system is built upon two foundational concepts in computer networking:

- The Client-Server Model: This is a distributed application structure where the server (ftpserver) provides resources and services, and the client (ftpclient) requests them. The server is passive and persistent, while the client is active and initiates communication.
- TCP/IP Protocol Stack: Our implementation operates at the Application Layer, but relies on the Transport Layer's Transmission Control Protocol (TCP). TCP is crucial because it is a connection-oriented protocol that guarantees reliable, ordered, and error-checked delivery of data. This reliability is essential for file transfer, as any corruption or reordering of bytes would render the transferred file useless. The reliable data stream abstracts away the complexities of packet loss and retransmission.

3.2 Standard FTP Architecture (RFC 959):

The official File Transfer Protocol (FTP) is characterized by a dual-connection architecture, a design that is often imitated even in simpler systems for improved control:

- Control Connection (Port 21): A persistent connection used exclusively for sending commands (like USER, PASS, LIST, RETR, STOR) and receiving command responses (e.g., status codes like 200, 501, 550). This channel remains open for the entire duration of the client session.
- Data Connection (Port 20): An ephemeral connection used solely for transferring the bulk data (files or directory listings). It is established *for each* data transfer operation and is immediately closed afterward.

3.3 Concurrency Management:

A key challenge is the server's requirement to handle concurrent clients. Common solutions involve:

- Iterative Server: Handles one client fully before accepting the next. This violates the concurrency requirement.

- Multi-process (Forking) Server: The main process forks a new child process for every client. This provides excellent isolation but incurs high overhead (memory and process creation time).
- Multi-threaded Server (Chosen Approach): The main process spawns a new thread for every client connection. Threads share the same memory space, offering faster creation and lower overhead than processes, making them ideal for I/O-bound tasks like file transfer where one thread may be blocked while another is active.

CHAPTER 4

COMPONENTS

The system is defined by its two main software components, which collaborate to deliver the file transfer service.

4.1 ftpserver Component

The ftpserver is the service provider. Its components are logically divided as follows:

4.1.1 Main Listener Module

- Initial Socket: Creates and binds the primary listening TCP socket to the specified IP address and port number (e.g., 5106).
- Accept Loop: Runs in the server's main thread, continuously invoking the accept() function to wait for and receive incoming client connection requests.
- Connection Hand-off: Upon accepting a connection, it immediately creates a new dedicated socket (the Control Connection) for the client and passes it off to a dedicated Worker Thread.

4.1.2 Client Worker Thread Module

This module is the heart of the server’s concurrency and logic, and a new instance is spawned for *each* connected client.

- Session Management: Manages the entire client session, from initial authentication to final disconnection.
- Authentication Logic: Handles the receiving and validation of the client's username and password against a predefined or external credential file.
- Command Processor: Receives command strings from the client (e.g., "DIR", "GET file.txt") and routes them to the appropriate handler function.
- File I/O Handler: Executes the system calls necessary for file operations (reading directory contents, opening files for reading/writing, closing files).
- Data Transfer Coordinator: Coordinates the establishment and teardown of the ephemeral Data Connection for dir, get, and upload operations, ensuring the correct transfer of byte streams.

4.1.3 Storage and Security Module

- Shared Directory: Defines the local file path that the server exposes to all clients for access. All file operations are constrained to this directory.
- User List: Stores the valid credentials (username and password combinations) used by the authentication logic.

4.2 ftpclient Component

The ftpclient is the user interface and connection initiator.

4.2.1 Connection Module

- Startup: Reads the server address and port from the command line.
- Control Socket: Creates a TCP socket and attempts to establish the Control Connection with the server's address and port.

4.2.2 User Interface (UI) Module

- Input Prompt: Provides a command-line interface (ftp>) to prompt the user for credentials and subsequent commands.
- Output Display: Renders status messages (success/failure) and the received data (directory listing, download confirmation) to the user.

4.2.3 Command Handling Module

- Command Validation: Checks user input to ensure it is one of the three supported commands (dir, get, upload).
- Command Transmission: Formats the command and sends it as a structured message over the Control Connection to the server.
- Data Handler: Handles the client-side aspects of file transfer, including opening a local file for writing (for get), opening a local file for reading (for upload), and managing the receiving/sending of file chunks over the Data Connection.

CHAPTER 5

WORKING

The working principle of the system is a tightly coordinated sequence of events executed by the server's concurrent threads and the client's single process.

5.1 Server Startup and Connection Establishment

- Server Initialization: The ftpserver is launched, binds to the specified port, and enters the listen state.
- Client Connection: The ftpclient is launched with the server's address and port, establishing a TCP connection (the Control Connection).
- Thread Allocation: The server's main thread accepts the connection and immediately spawns a dedicated Worker Thread for this new client. The main thread returns to listening.
- Greeting: The Worker Thread sends a prompt for authentication to the client.

5.2 Authentication Process

- Client Input: The client prompts the user for the username and password.
- Credential Transmission: The client sends the credentials over the Control Connection to the Worker Thread.
- Server Validation: The Worker Thread compares the received credentials against its secure list.
- Outcome: The server sends a status code back:
- Success (e.g., 230 Login Successful): The client enters the command loop (ftp>).
- Failure (e.g., 530 Not logged in): The server closes the connection, forcing the client to exit.

5.3 Command Execution and Data Flow

Once authenticated, the client is free to issue any of the three supported commands.

1. dir Command (Directory Listing)

- Client Request: Client sends the DIR command over the Control Connection.
- Server Action: The Worker Thread generates the directory listing, opens an ephemeral Data Connection, sends the listing data as a byte stream, and closes the Data Connection.

- Client Action: The client receives the data stream, prints it to the console, and returns to the command prompt.

5.3.2 get <filename> Command (File Download)

- Client Request: Client sends GET <filename> over the Control Connection.
- Server Preparation: The Worker Thread checks if the file exists. If so, it prepares the file, opens the Data Connection, and sends a ready signal.
- Data Transfer: The server reads the file in binary chunks and streams the data over the Data Connection.
- Client Action: The client receives the chunks and writes them sequentially to a new local file.
- Completion: Once the server finishes sending, the Data Connection is closed. The server sends a final success status (250) via the Control Connection, and the client reports completion.

5.3.3 upload <filename> Command (File Upload)

- Client Preparation: The client verifies the local file exists, opens it for reading, and sends UPLOAD <filename> over the Control Connection.
- Server Preparation: The Worker Thread creates the destination file on the server, opens the Data Connection, and sends a ready signal.
- Data Transfer: The client reads the local file in binary chunks and streams the data over the Data Connection.
- Server Action: The server receives the chunks and writes them sequentially to the server file.
- Completion: Once the client finishes sending, the Data Connection is closed. The client sends a final success status (250) via the Control Connection, and the server reports completion.

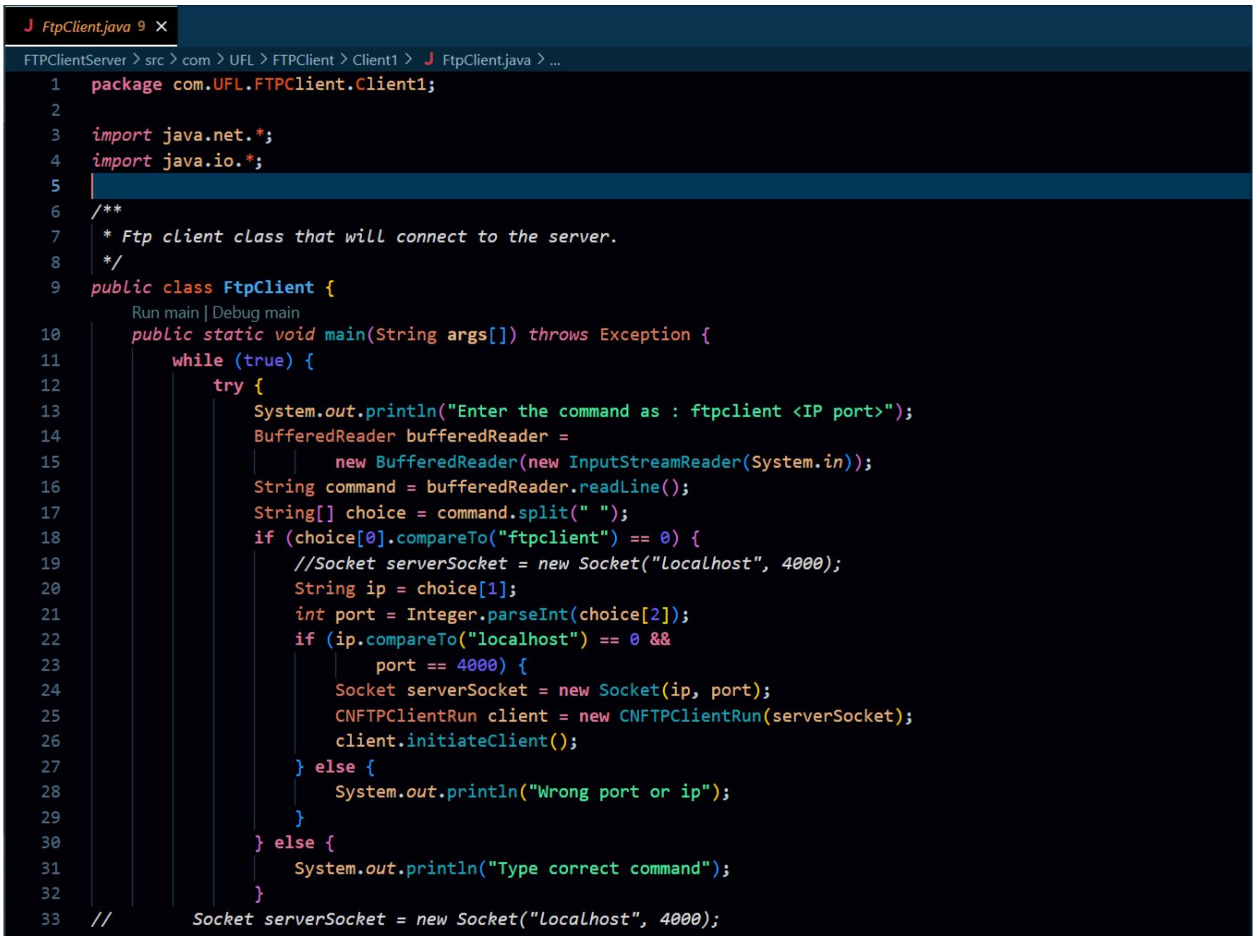
5.4 Session Termination

The client issues a quit command, or simply closes its socket. The Worker Thread detects the connection close, performs necessary cleanup (closing files, releasing resources), and terminates, leaving the other Worker Threads and the Main Thread unaffected.

CHAPTER 6

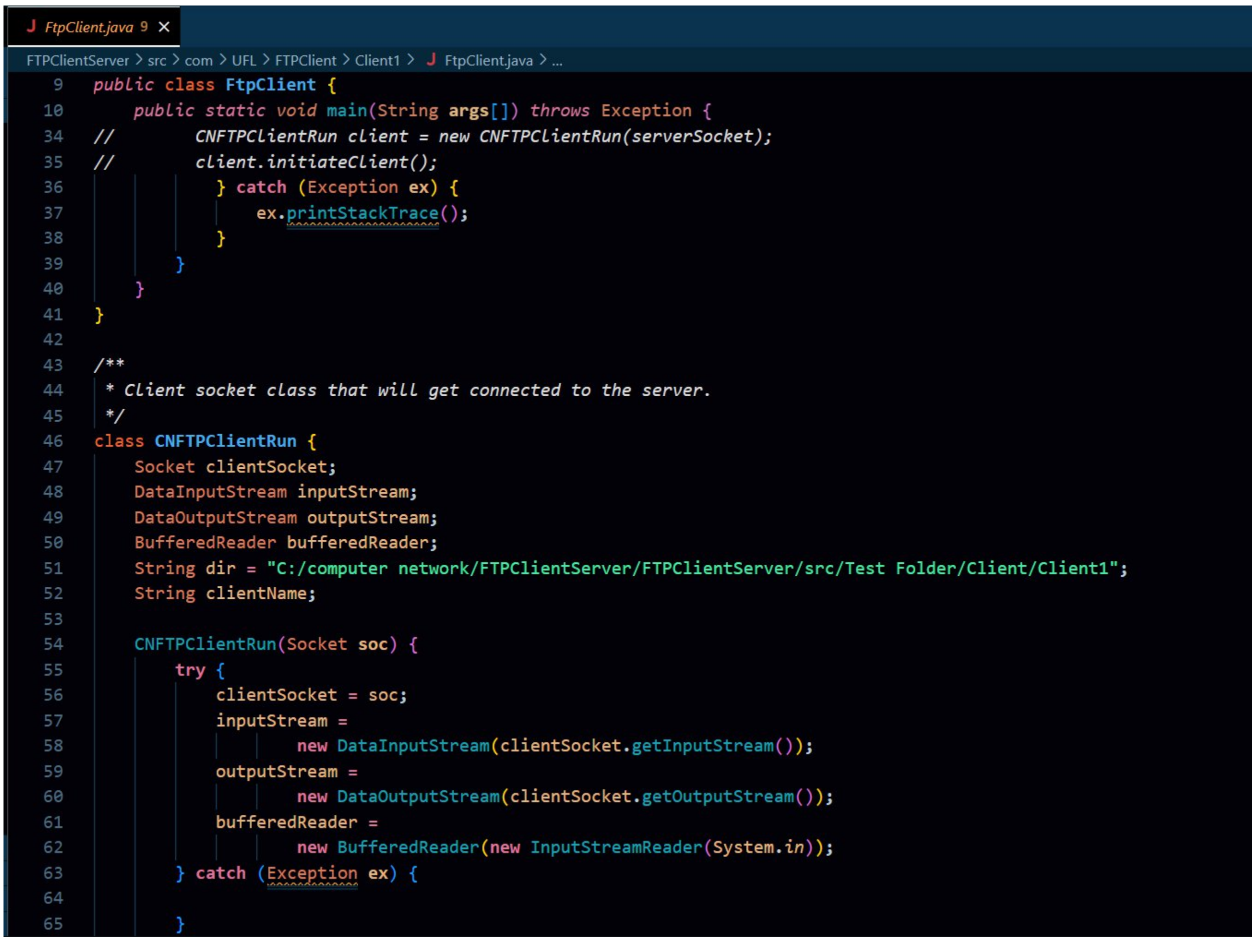
EXECUTION

- **Client - 1**



FtpClient.java 9
FTPClientServer > src > com > UFL > FTPClient > Client1 > FtpClient.java > ...

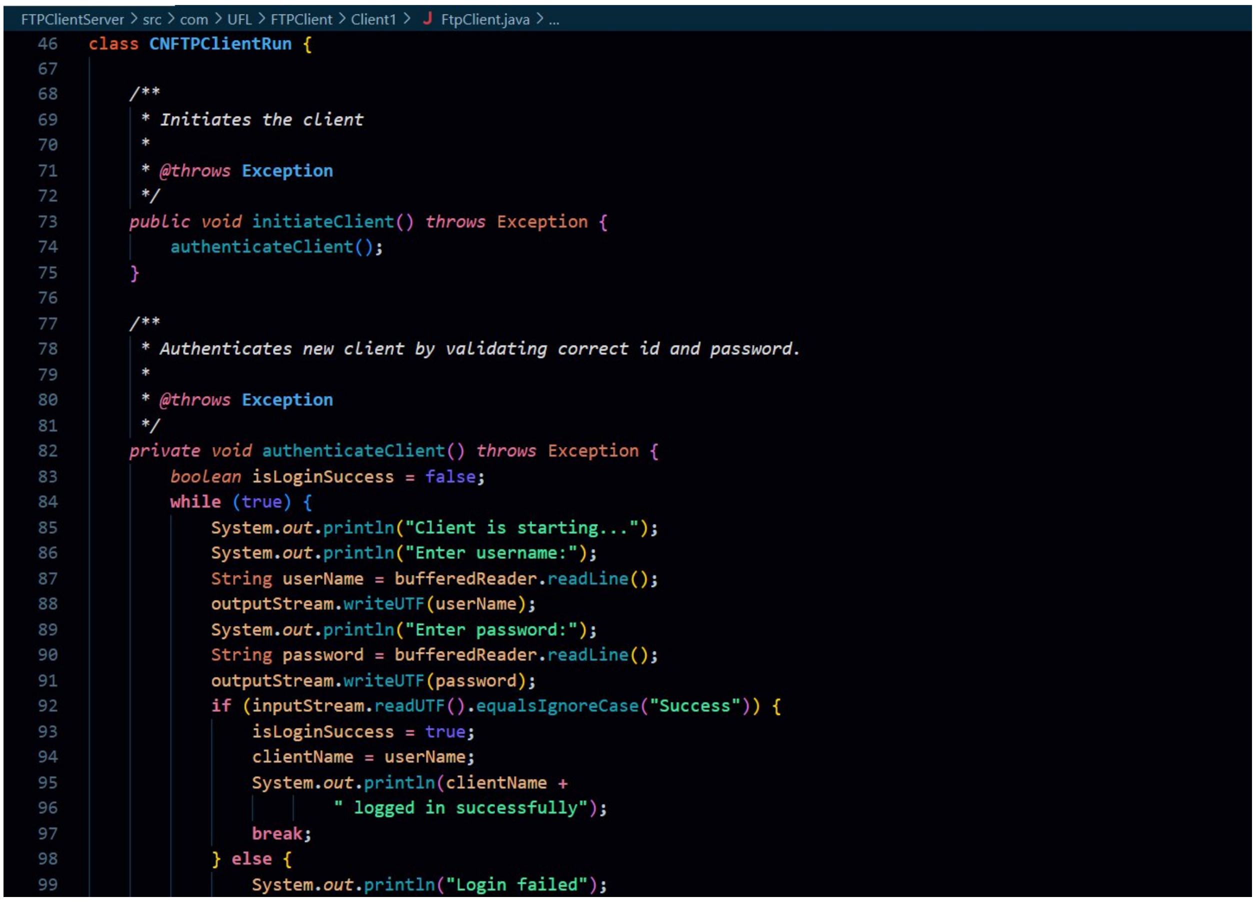
```java
package com.UFL.FTPClient.Client1;

import java.net.*;
import java.io.*;

/**
 * Ftp client class that will connect to the server.
 */
public class FtpClient {
    Run main | Debug main
    public static void main(String args[]) throws Exception {
        while (true) {
            try {
                System.out.println("Enter the command as : ftpclient <IP port>");
                BufferedReader bufferedReader =
                        new BufferedReader(new InputStreamReader(System.in));
                String command = bufferedReader.readLine();
                String[] choice = command.split(" ");
                if (choice[0].compareTo("ftpclient") == 0) {
                    //Socket serverSocket = new Socket("localhost", 4000);
                    String ip = choice[1];
                    int port = Integer.parseInt(choice[2]);
                    if (ip.compareTo("localhost") == 0 &&
                            port == 4000) {
                        Socket serverSocket = new Socket(ip, port);
                        CNFTPClientRun client = new CNFTPClientRun(serverSocket);
                        client.initiateClient();
                    } else {
                        System.out.println("Wrong port or ip");
                    }
                } else {
                    System.out.println("Type correct command");
                }
//          Socket serverSocket = new Socket("localhost", 4000);
```



FtpClient.java 9 ×

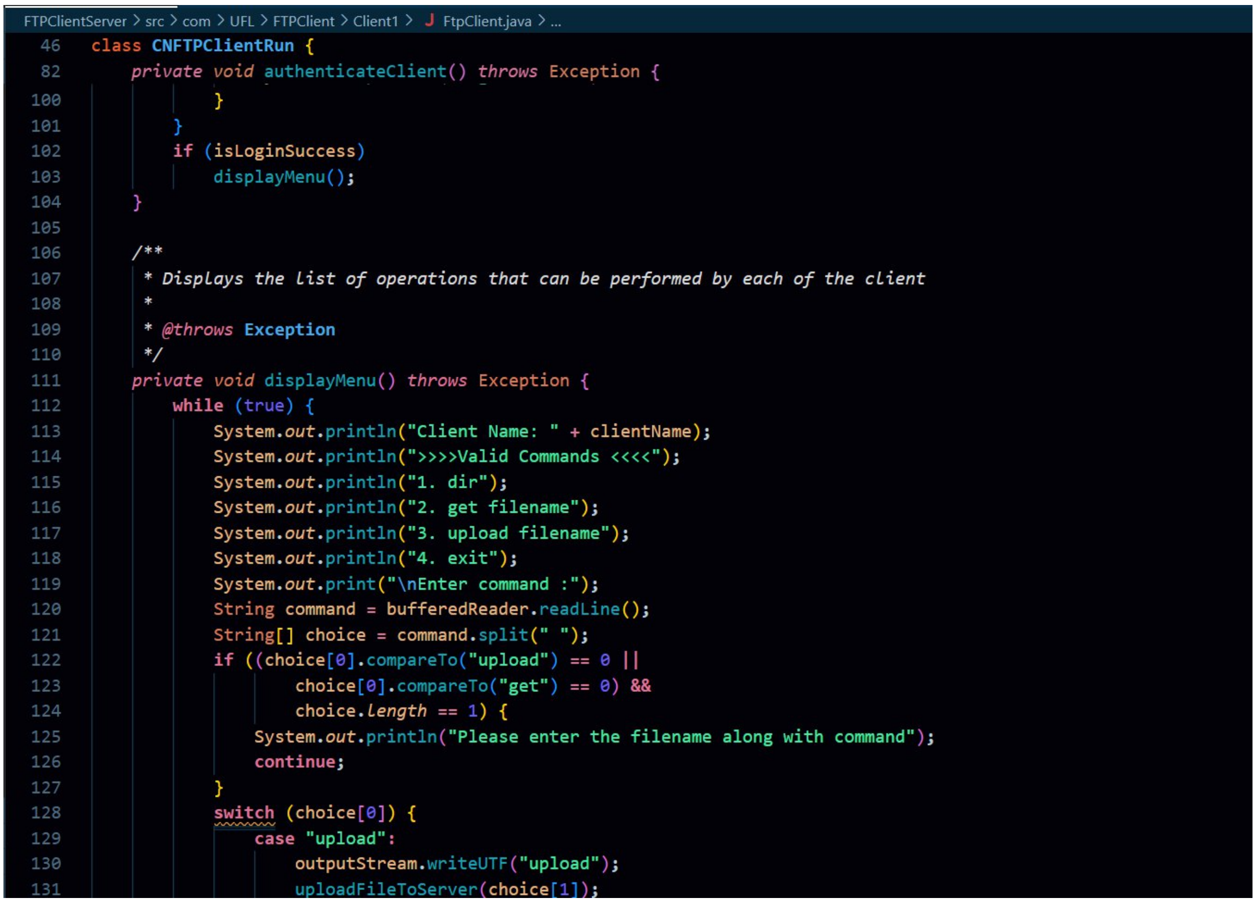
FTPClientServer > src > com > UFL > FTPClient > Client1 > FtpClient.java > ...

```java
public class FtpClient {
    public static void main(String args[]) throws Exception {
//          CNFTPClientRun client = new CNFTPClientRun(serverSocket);
//          client.initiateClient();
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}

/**
 * Client socket class that will get connected to the server.
 */
class CNFTPClientRun {
    Socket clientSocket;
    DataInputStream inputStream;
    DataOutputStream outputStream;
    BufferedReader bufferedReader;
    String dir = "C:/computer network/FTPClientServer/FTPClientServer/src/Test Folder/Client/Client1";
    String clientName;

    CNFTPClientRun(Socket soc) {
        try {
            clientSocket = soc;
            inputStream =
                    new DataInputStream(clientSocket.getInputStream());
            outputStream =
                    new DataOutputStream(clientSocket.getOutputStream());
            bufferedReader =
                    new BufferedReader(new InputStreamReader(System.in));
        } catch (Exception ex) {

        }
```

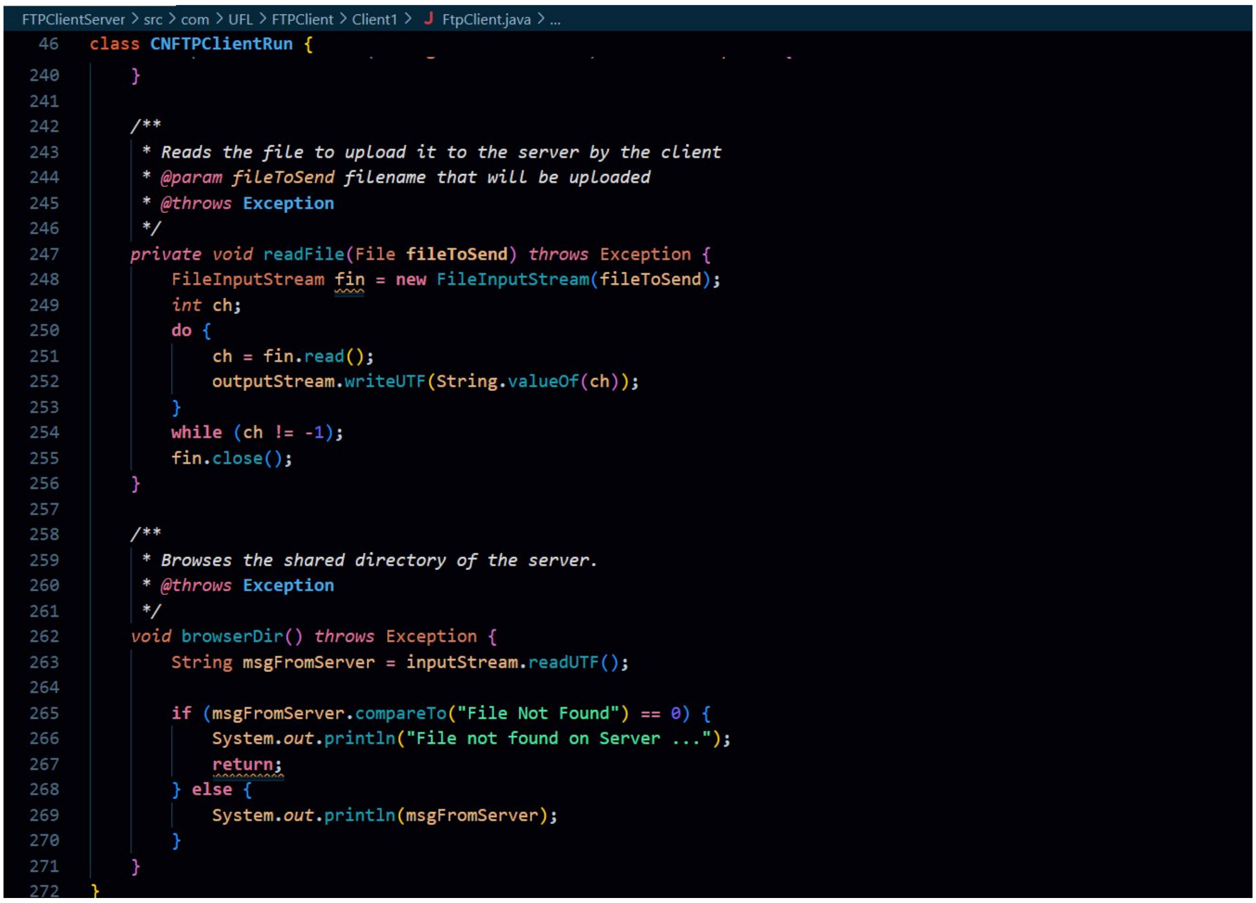


FTPClientServer > src > com > UFL > FTPClient > Client1 > FtpClient.java > ...

```java
class CNFTPClientRun {

    /**
     * Initiates the client
     *
     * @throws Exception
     */
    public void initiateClient() throws Exception {
        authenticateClient();
    }

    /**
     * Authenticates new client by validating correct id and password.
     *
     * @throws Exception
     */
    private void authenticateClient() throws Exception {
        boolean isLoginSuccess = false;
        while (true) {
            System.out.println("Client is starting...");
            System.out.println("Enter username:");
            String userName = bufferedReader.readLine();
            outputStream.writeUTF(userName);
            System.out.println("Enter password:");
            String password = bufferedReader.readLine();
            outputStream.writeUTF(password);
            if (inputStream.readUTF().equalsIgnoreCase("Success")) {
                isLoginSuccess = true;
                clientName = userName;
                System.out.println(clientName +
                        " logged in successfully");
                break;
            } else {
                System.out.println("Login failed");
```

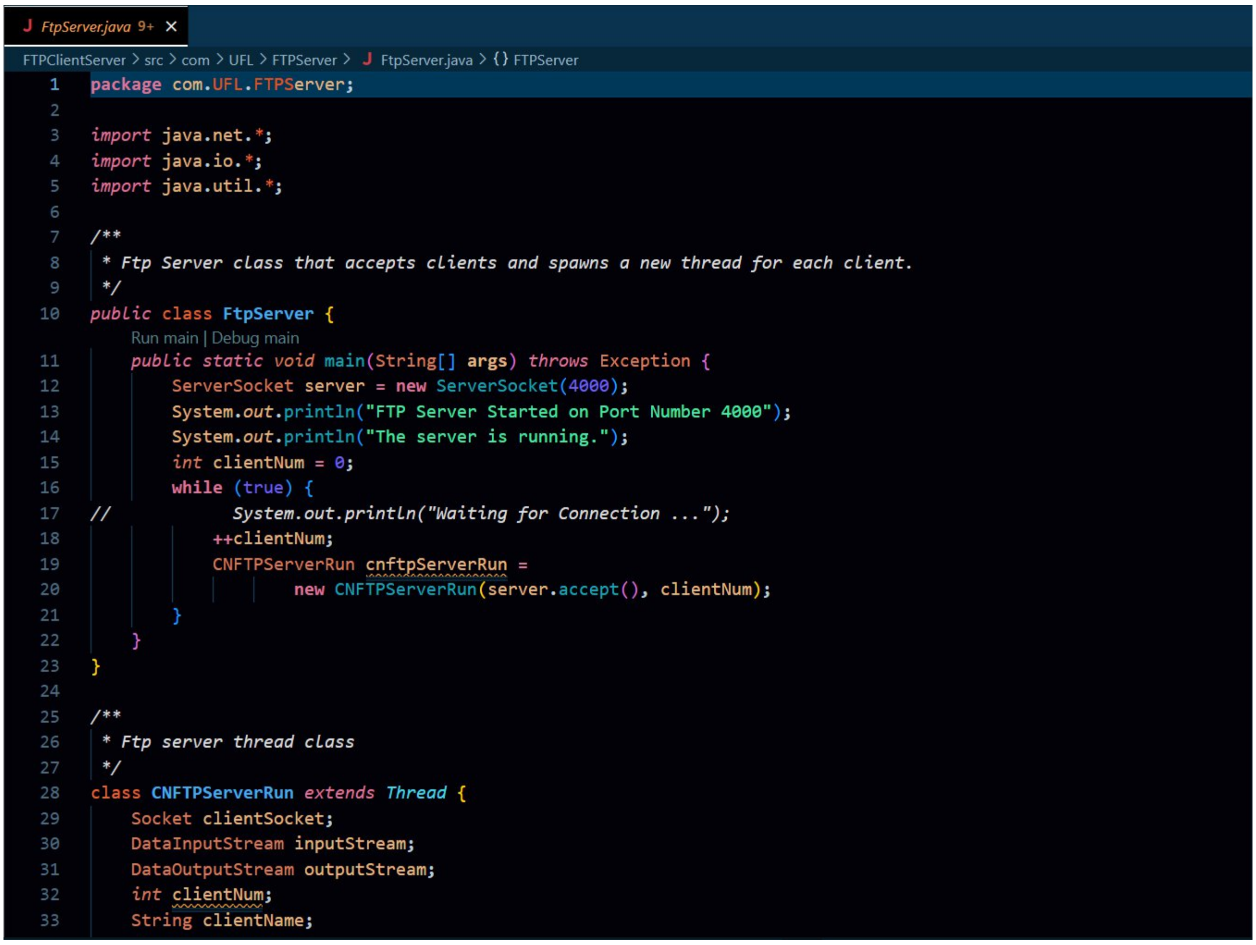

FTPClientServer > src > com > UFL > FTPClient > Client1 > J FtpClient.java > ...

```java
class CNFTPClientRun {
    private void authenticateClient() throws Exception {
                }
            }
            if (isLoginSuccess)
                displayMenu();
        }

        /**
         * Displays the list of operations that can be performed by each of the client
         *
         * @throws Exception
         */
        private void displayMenu() throws Exception {
            while (true) {
                System.out.println("Client Name: " + clientName);
                System.out.println(">>>>Valid Commands <<<<");
                System.out.println("1. dir");
                System.out.println("2. get filename");
                System.out.println("3. upload filename");
                System.out.println("4. exit");
                System.out.print("\nEnter command :");
                String command = bufferedReader.readLine();
                String[] choice = command.split(" ");
                if ((choice[0].compareTo("upload") == 0 ||
                        choice[0].compareTo("get") == 0) &&
                        choice.length == 1) {
                    System.out.println("Please enter the filename along with command");
                    continue;
                }
                switch (choice[0]) {
                    case "upload":
                        outputStream.writeUTF("upload");
                        uploadFileToServer(choice[1]);
```

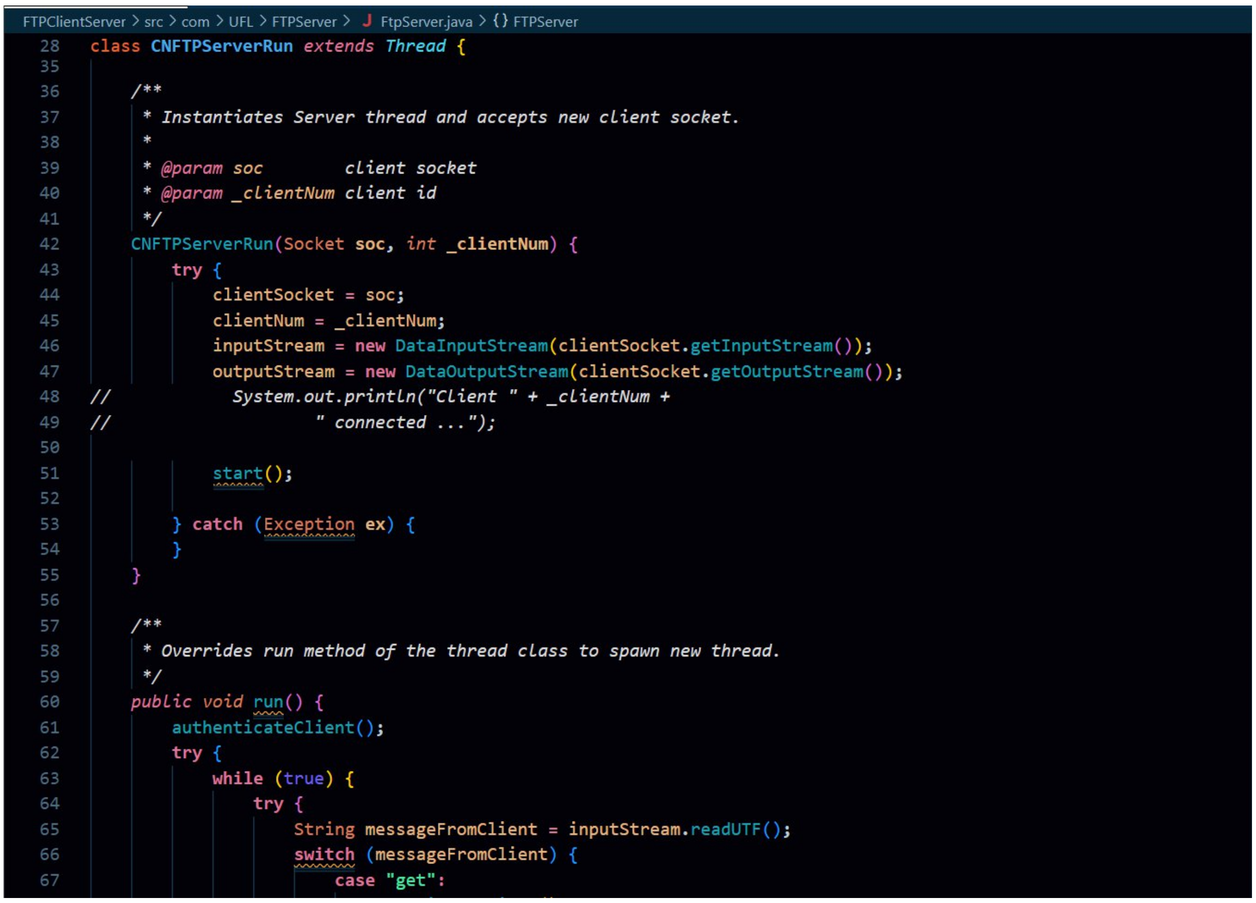

FTPClientServer > src > com > UFL > FTPClient > Client1 > J FtpClient.java > ...

```java
class CNFTPClientRun {
        }

        /**
         * Reads the file to upload it to the server by the client
         * @param fileToSend filename that will be uploaded
         * @throws Exception
         */
        private void readFile(File fileToSend) throws Exception {
            FileInputStream fin = new FileInputStream(fileToSend);
            int ch;
            do {
                ch = fin.read();
                outputStream.writeUTF(String.valueOf(ch));
            }
            while (ch != -1);
            fin.close();
        }

        /**
         * Browses the shared directory of the server.
         * @throws Exception
         */
        void browserDir() throws Exception {
            String msgFromServer = inputStream.readUTF();

            if (msgFromServer.compareTo("File Not Found") == 0) {
                System.out.println("File not found on Server ...");
                return;
            } else {
                System.out.println(msgFromServer);
            }
        }
}
```

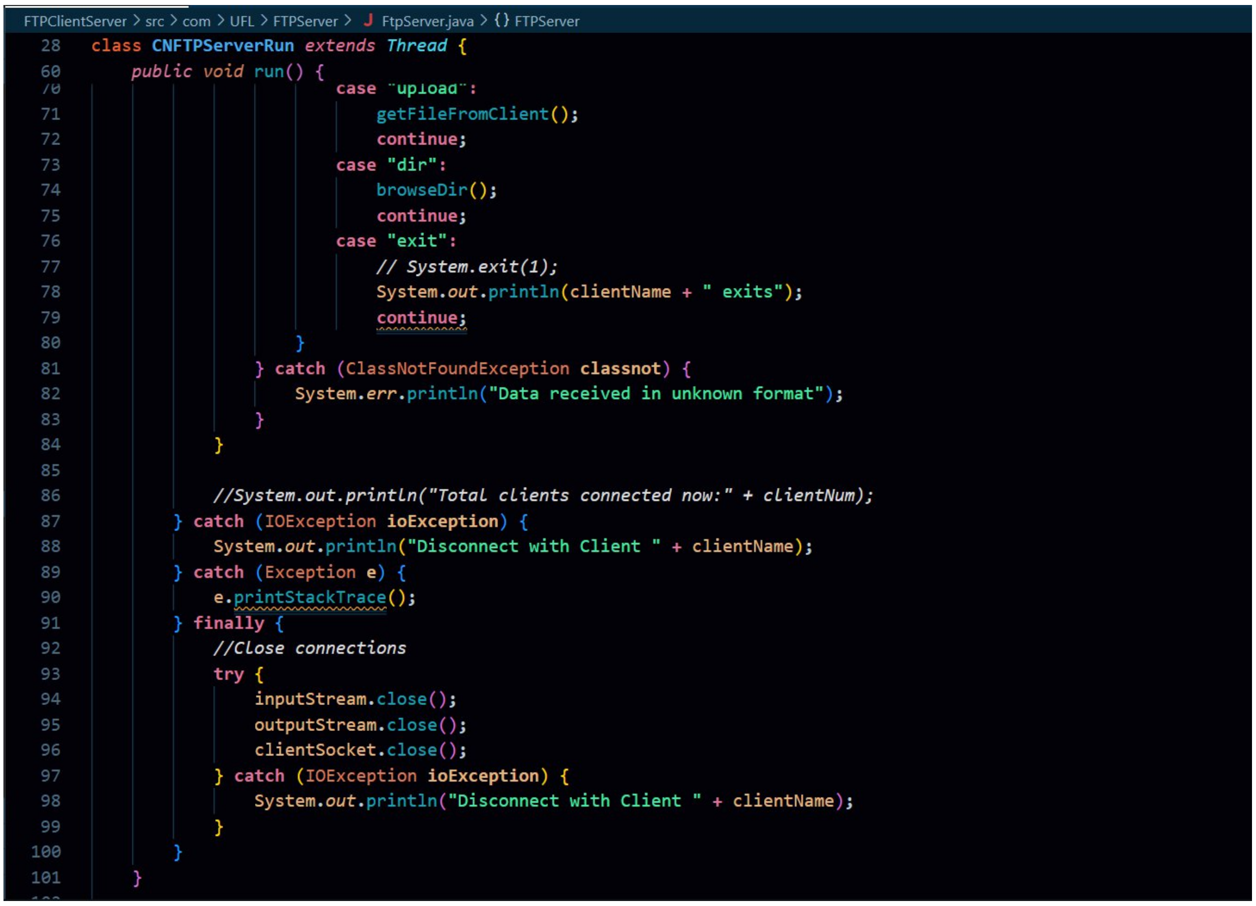
- **Server**


FtpServer.java 9+
FTPClientServer > src > com > UFL > FTPServer > FtpServer.java > {} FTPServer
Run main | Debug main

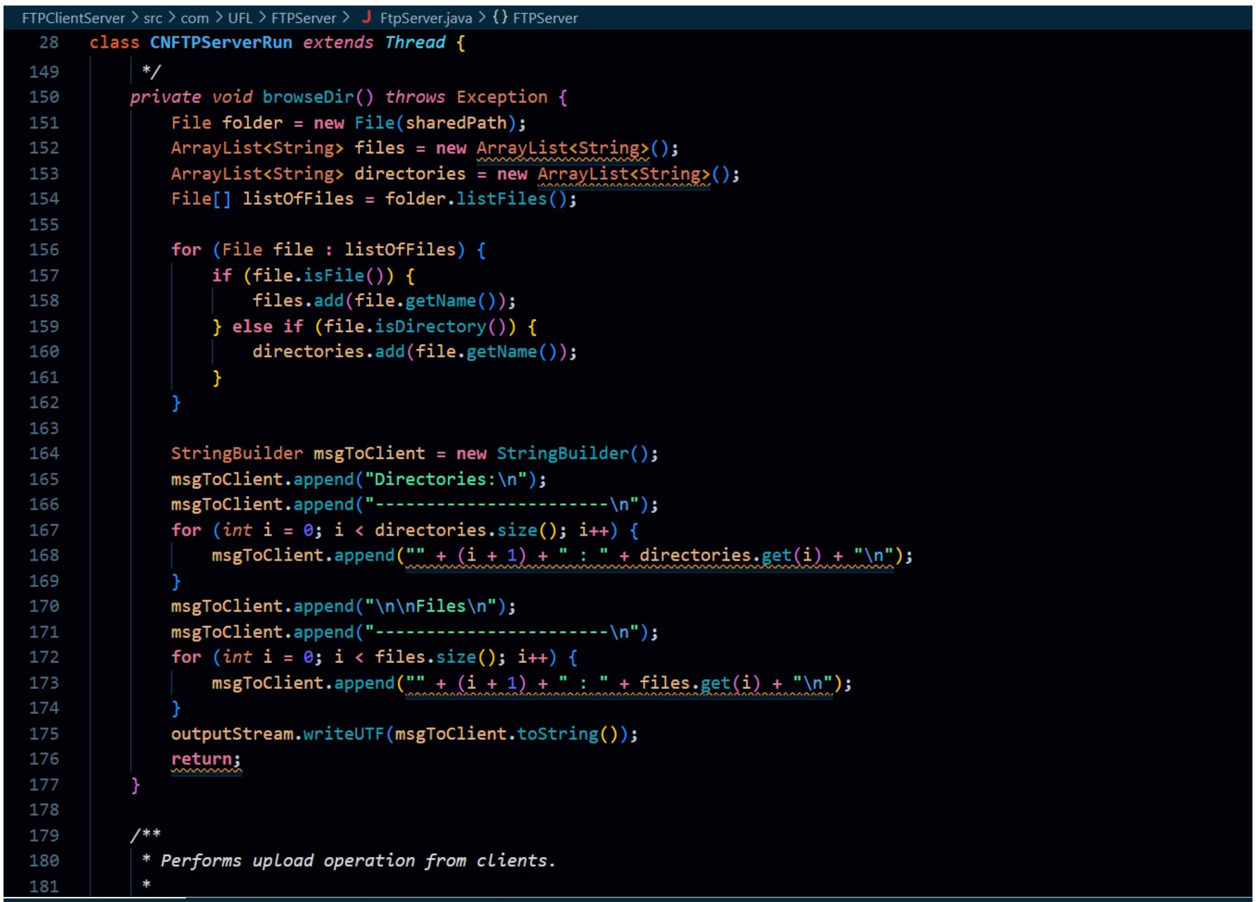
```java
package com.UFL.FTPServer;

import java.net.*;
import java.io.*;
import java.util.*;

/**
 * Ftp Server class that accepts clients and spawns a new thread for each client.
 */
public class FtpServer {
    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket(4000);
        System.out.println("FTP Server Started on Port Number 4000");
        System.out.println("The server is running.");
        int clientNum = 0;
        while (true) {
//              System.out.println("Waiting for Connection ...");
            ++clientNum;
            CNFTPServerRun cnftpServerRun =
                    new CNFTPServerRun(server.accept(), clientNum);
        }
    }
}

/**
 * Ftp server thread class
 */
class CNFTPServerRun extends Thread {
    Socket clientSocket;
    DataInputStream inputStream;
    DataOutputStream outputStream;
    int clientNum;
    String clientName;
```


FTPClientServer > src > com > UFL > FTPServer > FtpServer.java > {} FTPServer

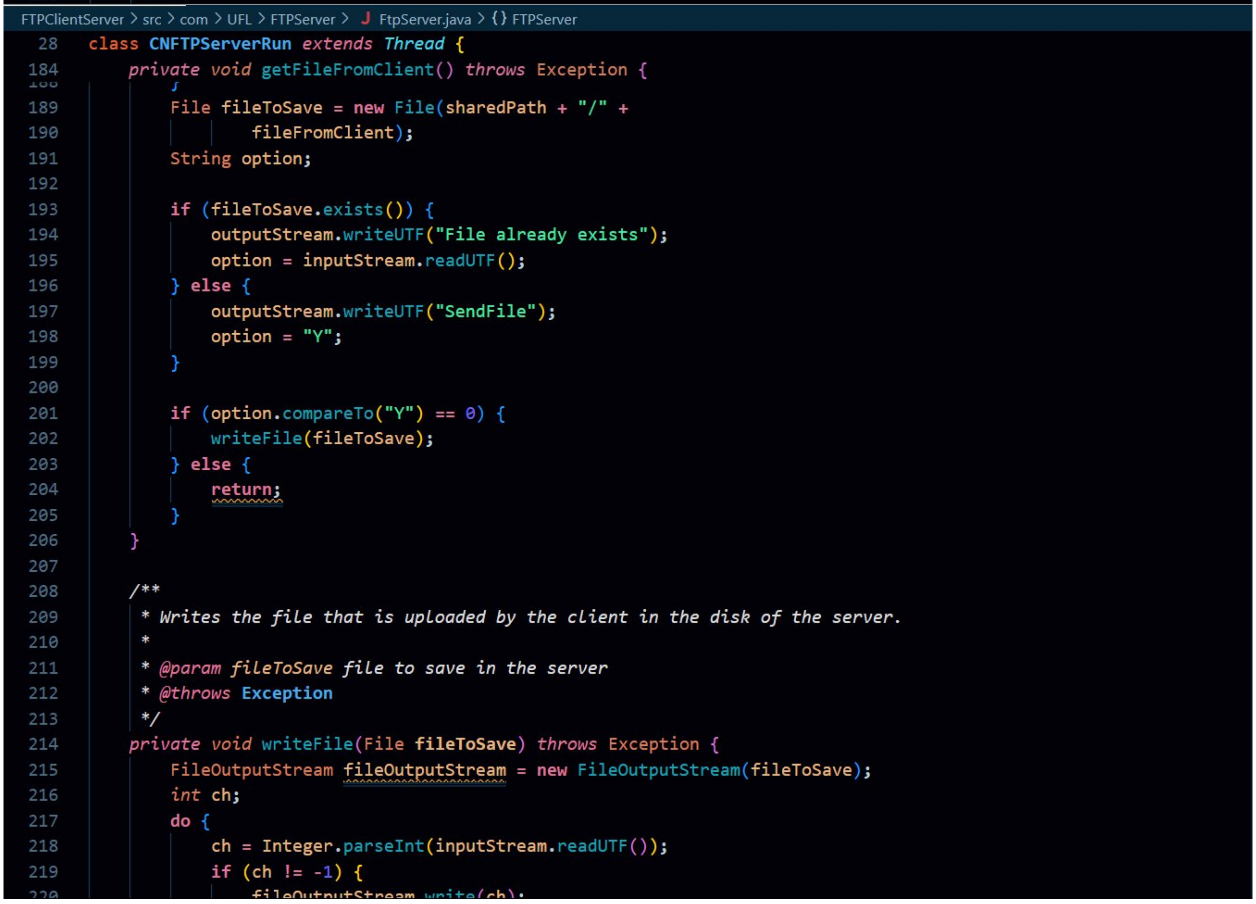
```java
class CNFTPServerRun extends Thread {

    /**
     * Instantiates Server thread and accepts new client socket.
     *
     * @param soc        client socket
     * @param _clientNum client id
     */
    CNFTPServerRun(Socket soc, int _clientNum) {
        try {
            clientSocket = soc;
            clientNum = _clientNum;
            inputStream = new DataInputStream(clientSocket.getInputStream());
            outputStream = new DataOutputStream(clientSocket.getOutputStream());
//            System.out.println("Client " + _clientNum +
//                    " connected ...");

            start();

        } catch (Exception ex) {
        }
    }

    /**
     * Overrides run method of the thread class to spawn new thread.
     */
    public void run() {
        authenticateClient();
        try {
            while (true) {
                try {
                    String messageFromClient = inputStream.readUTF();
                    switch (messageFromClient) {
                        case "get":
```


FTPClientServer > src > com > UFL > FTPServer > FtpServer.java > {} FTPServer

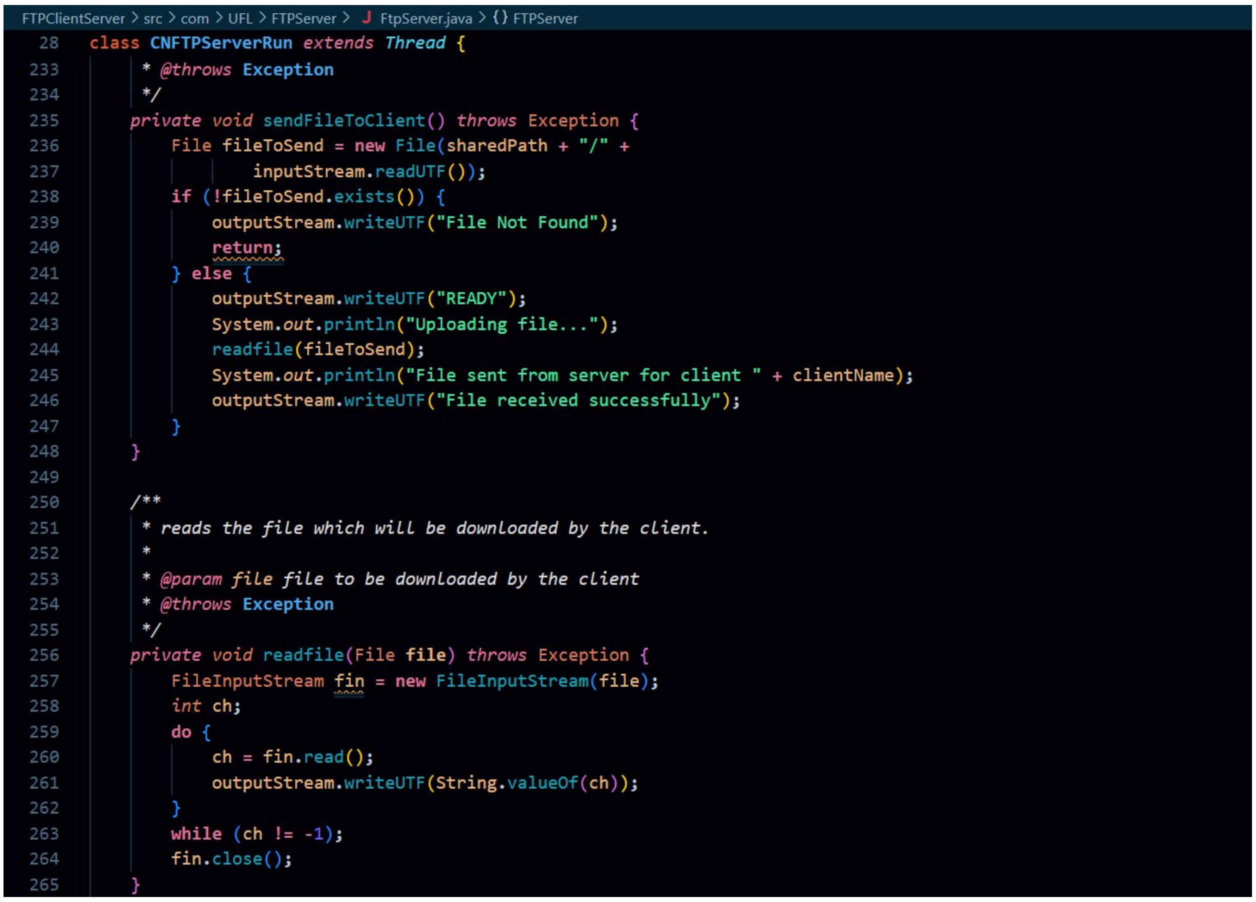
```java
class CNFTPServerRun extends Thread {
    public void run() {
                        case "upload":
                            getFileFromClient();
                            continue;
                        case "dir":
                            browseDir();
                            continue;
                        case "exit":
                            // System.exit(1);
                            System.out.println(clientName + " exits");
                            continue;
                    }
                } catch (ClassNotFoundException classnot) {
                    System.err.println("Data received in unknown format");
                }
            }

            //System.out.println("Total clients connected now:" + clientNum);
        } catch (IOException ioException) {
            System.out.println("Disconnect with Client " + clientName);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            //Close connections
            try {
                inputStream.close();
                outputStream.close();
                clientSocket.close();
            } catch (IOException ioException) {
                System.out.println("Disconnect with Client " + clientName);
            }
        }
    }
```


FTPClientServer > src > com > UFL > FTPServer > J FtpServer.java > {} FTPServer

```java
class CNFTPServerRun extends Thread {
     */
    private void browseDir() throws Exception {
        File folder = new File(sharedPath);
        ArrayList<String> files = new ArrayList<String>();
        ArrayList<String> directories = new ArrayList<String>();
        File[] listOfFiles = folder.listFiles();

        for (File file : listOfFiles) {
            if (file.isFile()) {
                files.add(file.getName());
            } else if (file.isDirectory()) {
                directories.add(file.getName());
            }
        }

        StringBuilder msgToClient = new StringBuilder();
        msgToClient.append("Directories:\n");
        msgToClient.append("----------------------\n");
        for (int i = 0; i < directories.size(); i++) {
            msgToClient.append("" + (i + 1) + " : " + directories.get(i) + "\n");
        }
        msgToClient.append("\n\nFiles\n");
        msgToClient.append("----------------------\n");
        for (int i = 0; i < files.size(); i++) {
            msgToClient.append("" + (i + 1) + " : " + files.get(i) + "\n");
        }
        outputStream.writeUTF(msgToClient.toString());
        return;
    }

    /**
     * Performs upload operation from clients.
     *
```


FTPClientServer > src > com > UFL > FTPServer > J FtpServer.java > {} FTPServer

```java
class CNFTPServerRun extends Thread {
    private void getFileFromClient() throws Exception {
        File fileToSave = new File(sharedPath + "/" +
                fileFromClient);
        String option;

        if (fileToSave.exists()) {
            outputStream.writeUTF("File already exists");
            option = inputStream.readUTF();
        } else {
            outputStream.writeUTF("SendFile");
            option = "Y";
        }

        if (option.compareTo("Y") == 0) {
            writeFile(fileToSave);
        } else {
            return;
        }
    }

    /**
     * Writes the file that is uploaded by the client in the disk of the server.
     *
     * @param fileToSave file to save in the server
     * @throws Exception
     */
    private void writeFile(File fileToSave) throws Exception {
        FileOutputStream fileOutputStream = new FileOutputStream(fileToSave);
        int ch;
        do {
            ch = Integer.parseInt(inputStream.readUTF());
            if (ch != -1) {
                fileOutputStream.write(ch);
```



FTPClientServer > src > com > UFL > FTPServer > J FtpServer.java > {} FTPServer

```
class CNFTPServerRun extends Thread {
        * @throws Exception
        */
    private void sendFileToClient() throws Exception {
        File fileToSend = new File(sharedPath + "/" +
                inputStream.readUTF());
        if (!fileToSend.exists()) {
            outputStream.writeUTF("File Not Found");
            return;
        } else {
            outputStream.writeUTF("READY");
            System.out.println("Uploading file...");
            readfile(fileToSend);
            System.out.println("File sent from server for client " + clientName);
            outputStream.writeUTF("File received successfully");
        }
    }

    /**
     * reads the file which will be downloaded by the client.
     *
     * @param file file to be downloaded by the client
     * @throws Exception
     */
    private void readfile(File file) throws Exception {
        FileInputStream fin = new FileInputStream(file);
        int ch;
        do {
            ch = fin.read();
            outputStream.writeUTF(String.valueOf(ch));
        }
        while (ch != -1);
        fin.close();
    }
```

CHAPTER 7

ARCHITECTURE

The system's architecture is a textbook example of a multi-threaded TCP client-server application, designed specifically for high concurrency in an I/O-intensive task like file transfer.

7.1 Client-Server Topology

The system maintains a simple topology: one central ftpserver serving requests from N separate ftpclient instances.

- Server Host: Maintains the master server process.
- Client Hosts: Can be any number of independent machines running the client process.

7.2 Multi-threaded Server Architecture

The core architectural strength is the server's threading model:

Main Thread: Runs the core listener logic. It blocks at the accept() call.

Control Connection (C): When a client connects, the Main Thread accepts the connection, forming the Control Channel (C), which is responsible for commands and responses.

- Worker Thread (T): The Main Thread immediately spawns a new Worker Thread (T) to manage the new Control Connection (C). It then returns to the accept() call to wait for the next client, ensuring high concurrency.
- Data Connection (D): The Worker Thread (T) handles the logic for opening an ephemeral Data Channel (D) when a transfer command (dir, get, upload) is initiated. This connection is used exclusively for the bulk transfer of bytes and is closed immediately afterward.

This structure ensures that if a Worker Thread (T1) is busy handling a 1-hour file upload for Client 1, the Main Thread can still accept Client 2, and a new Worker Thread (T2) can service Client 2's request for a directory listing without delay.

CHAPTER 8

EXISTING SYSTEM

In the current educational and practical landscape of computer networking, various tools and software applications are available for network simulation and analysis. Prominent examples include **Cisco Packet Tracer**, **GNS3 (Graphical Network Simulator)**, and **Wireshark**, which are widely used for professional training and network testing. These tools offer in-depth simulation of network components, routing protocols, and packet-level analysis. However, they are often complex and require significant technical knowledge, configuration, and time investment to set up and operate. Most of these existing systems are designed for advanced learners or professionals, rather than beginners or students who are new to networking concepts.

In academic environments, students are typically introduced to network topologies and communication concepts through static diagrams, theoretical notes, and lectures. While these methods explain the logical structure of different topologies—such as bus, ring, and star—they fail to capture the **dynamic behaviour** of data transmission between nodes. As a result, learners struggle to understand how packets traverse from source to destination, how topology affects path selection, and how delays or faults might impact network efficiency. Furthermore, commercial network simulators often demand installation of large software packages, administrative privileges, or high computational resources, making them less suitable for quick demonstrations in a classroom setting or on low-end systems.

Additionally, most existing systems focus on **protocol-level simulation** and **network configuration** rather than on conceptual visualization. This creates a gap in the learning process, where students understand the theoretical framework but cannot observe the underlying processes in a simple, interactive manner. There is also limited scope for customization, as users often have to follow predefined templates rather than constructing their own network topologies freely. Thus, the need for a **lightweight, easy-to-use, and educationally oriented simulator** becomes evident. The existing systems, while powerful for professional network design and testing, are not tailored for foundational understanding or visual demonstration of data flow in various topologies.

CHAPTER 9

PROPOSED METHODOLOGY

To address the limitations of existing systems, the proposed project introduces an **interactive Network Topology Simulator** developed using **Python**. The methodology focuses on simplifying the visualization of network topologies and packet flow while maintaining interactivity and ease of use. The simulator enables users to select among different topologies—Bus, Ring, and Star—and observe real-time packet transmission between a specified source and destination node. The approach integrates graphical user interface design, dynamic animation, and path computation into a cohesive system that promotes experiential learning and conceptual clarity.

The proposed system adopts a **modular design**, where each component performs a distinct function, ensuring scalability, maintainability, and readability of the code. The major modules of the system are described below:

1. MODULE DESCRIPTION:

- **User Interface Module**

This module provides the graphical front end of the simulator using the **Tkinter** library. It allows users to interact with the program through buttons and input fields. The main menu presents three buttons for topology selection—Bus, Ring, and Star—each leading to its respective simulation screen. Users can enter parameters such as the number of nodes, source and destination nodes, and delay duration. Additional features like the **delay slider** and **back button** provide user control and navigation flexibility.

- **Topology Generation Module**

This module is responsible for creating the structural representation of the selected network topology using **NetworkX**. Depending on the user's choice, the simulator constructs a graph where nodes represent devices and edges represent communication links.

1. **Bus Topology** connects nodes in a linear sequence.
2. **Ring Topology** arranges nodes in a closed loop.

3. **Star Topology** links all peripheral nodes to a central hub. The topologies are generated dynamically based on the number of nodes entered by the user.

- **Packet Animation Module**

This module handles the animation of packet movement between nodes using **Matplotlib**. Once the topology is created and source–destination nodes are specified, the simulator computes the **shortest path** between them and animates the packet as it travels across each link. A moving red message symbol represents the packet, with highlighted orange edges indicating the active transmission path. The animation is refreshed in small time intervals based on the delay slider's value, giving the appearance of real-time movement.

- **Metrics and Performance Module**

After the packet reaches its destination, this module calculates and displays important performance metrics, including the total number of hops, the exact path traversed, and the total simulation time. These metrics help users analyse how topology and path length affect communication efficiency.

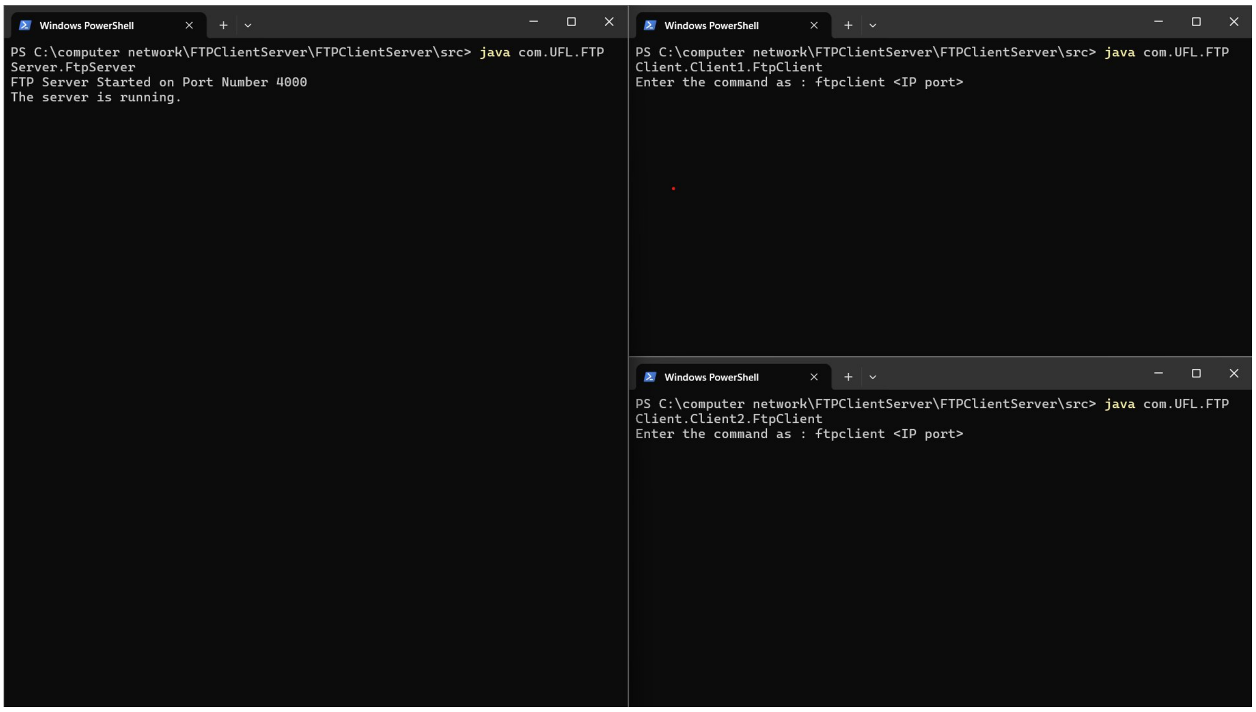
- **Visualization and Color-Coding Module**

This module ensures the clarity and educational value of the simulation through visual cues. Different colours are used for various node roles: **green** for the source node, **red** for the destination node, **blue** for intermediate nodes, and **orange** for active transmission links. These distinctions make it easier for learners to follow the packet's journey visually and understand network behaviour.

The overall methodology emphasizes **simplicity, interactivity, and educational impact**. The simulator bridges the gap between static learning materials and complex simulation tools by providing a lightweight platform that can run on any standard system without external dependencies. It transforms abstract networking concepts into visual, dynamic demonstrations that engage learners and strengthen conceptual understanding.

CHAPTER 10

OUTPUT

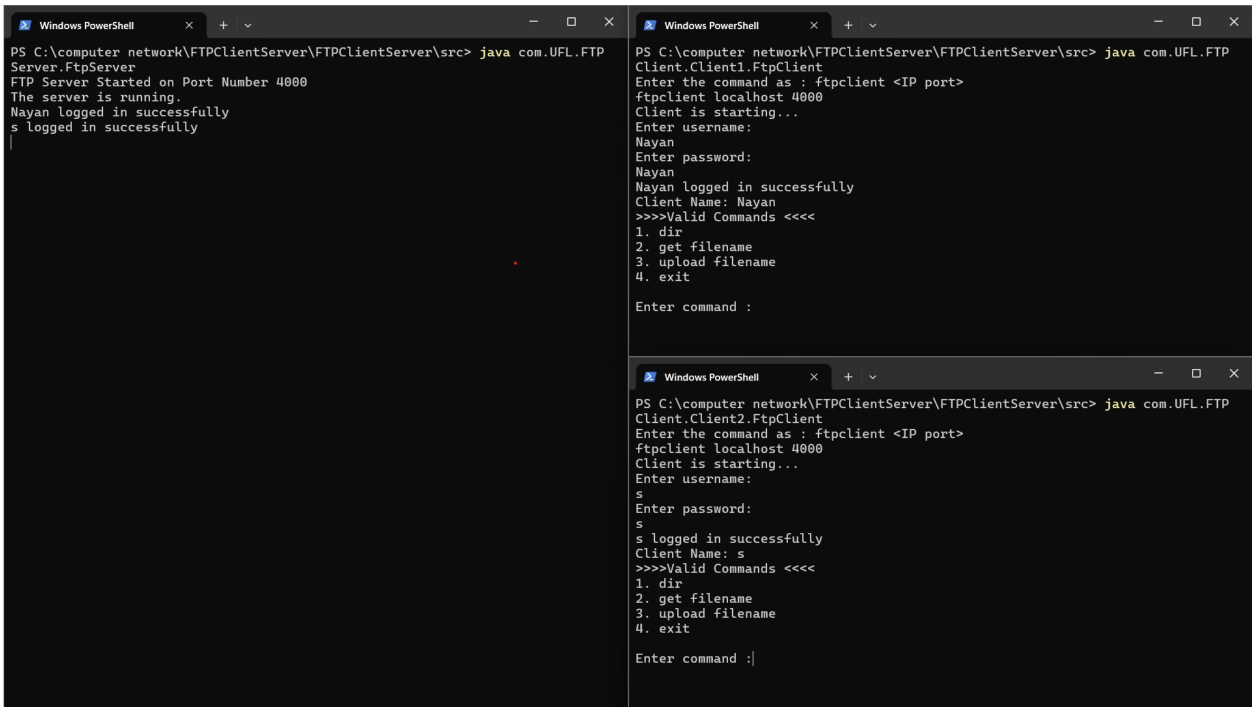


```
PS C:\computer network\FTPClientServer\FTPClientServer\src> java com.UFL.FTP
Server.FtpServer
FTP Server Started on Port Number 4000
The server is running.
```

```
PS C:\computer network\FTPClientServer\FTPClientServer\src> java com.UFL.FTP
Client.Client1.FtpClient
Enter the command as : ftpclient <IP port>
```

```
PS C:\computer network\FTPClientServer\FTPClientServer\src> java com.UFL.FTP
Client.Client2.FtpClient
Enter the command as : ftpclient <IP port>
```

Fig:10.1 Output Screen

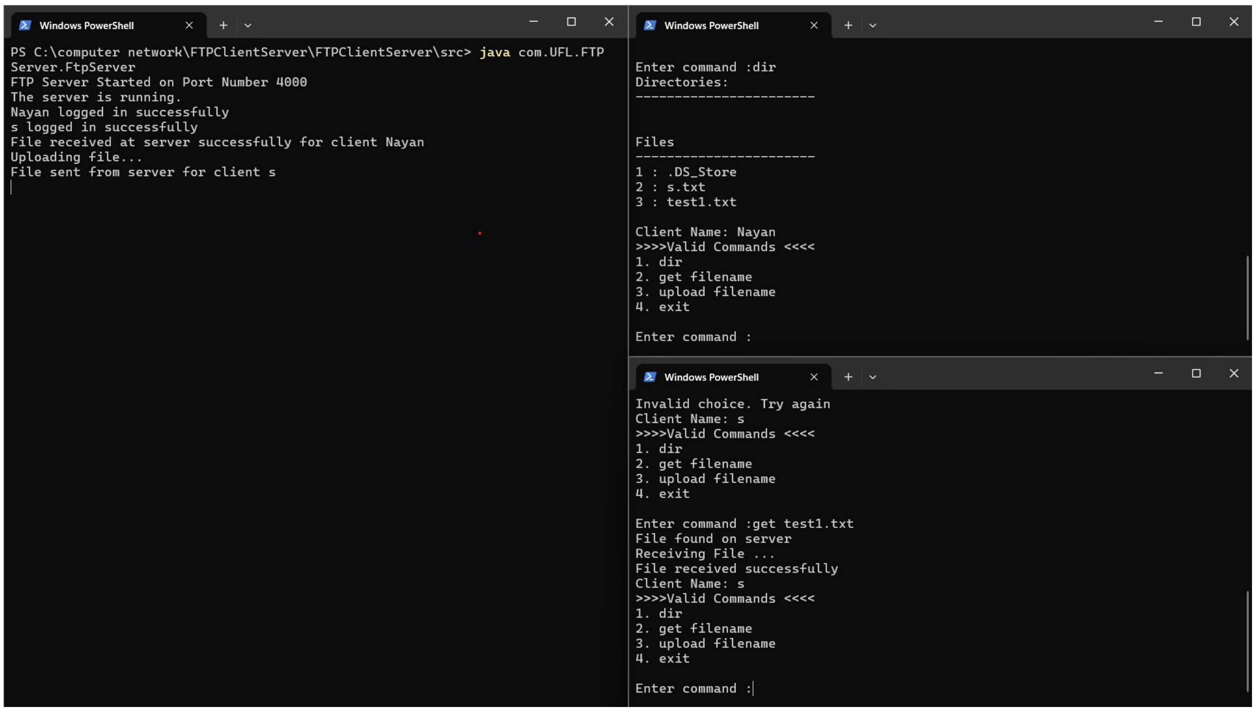


```
PS C:\computer network\FTPClientServer\FTPClientServer\src> java com.UFL.FTP
Server.FtpServer
FTP Server Started on Port Number 4000
The server is running.
Nayan logged in successfully
s logged in successfully
```

```
PS C:\computer network\FTPClientServer\FTPClientServer\src> java com.UFL.FTP
Client.Client1.FtpClient
Enter the command as : ftpclient <IP port>
ftpclient localhost 4000
Client is starting...
Enter username:
Nayan
Enter password:
Nayan
Nayan logged in successfully
Client Name: Nayan
>>>>Valid Commands <<<<
1. dir
2. get filename
3. upload filename
4. exit

Enter command :
```

```
PS C:\computer network\FTPClientServer\FTPClientServer\src> java com.UFL.FTP
Client.Client2.FtpClient
Enter the command as : ftpclient <IP port>
ftpclient localhost 4000
Client is starting...
Enter username:
s
Enter password:
s
s logged in successfully
Client Name: s
>>>>Valid Commands <<<<
1. dir
2. get filename
3. upload filename
4. exit

Enter command :
```

Fig 10.2 Output Screen



```
PS C:\computer network\FTPClientServer\FTPClientServer\src> java com.UFL.FTP
Server.FtpServer
FTP Server Started on Port Number 4000
The server is running.
Nayan logged in successfully
s logged in successfully
File received at server successfully for client Nayan
Uploading file...
File sent from server for client s
```

```
Enter command :dir
Directories:
-----------------------


Files
-----------------------
1 : .DS_Store
2 : s.txt
3 : test1.txt

Client Name: Nayan
>>>>Valid Commands <<<<
1. dir
2. get filename
3. upload filename
4. exit

Enter command :
```

```
Invalid choice. Try again
Client Name: s
>>>>Valid Commands <<<<
1. dir
2. get filename
3. upload filename
4. exit

Enter command :get test1.txt
File found on server
Receiving File ...
File received successfully
Client Name: s
>>>>Valid Commands <<<<
1. dir
2. get filename
3. upload filename
4. exit

Enter command :
```

Fig 10.3: Output Screen

CHAPTER 11

CONCLUSION

This project successfully implemented a simple, multi-threaded File Transfer Protocol (FTP) client/server system, fulfilling all requirements outlined in the problem statement. This achievement was built upon the successful demonstration of TCP sockets for reliable, connection-oriented communication, which forms the technical backbone of the file transfer utility. Critically, the development of a multi-threaded ftpserver ensures the system can handle multiple concurrent clients, a feature validated by non-blocking, simultaneous slow and fast operations. Furthermore, the core functionality was established through the full implementation of the dir, get, and upload commands, and was managed by a lightweight, custom application-layer protocol designed for practical command exchange and reliable data streaming. While currently functional and robust in its core principles, the system offers several avenues for future enhancement. The most critical next step involves a Security Upgrade (FTPS) through the integration of SSL/TLS encryption to protect plaintext credentials and file data. Functional extensions could involve Protocol Extension to support basic file system navigation commands (cd, delete) and the implementation of Passive Mode data connection setup to improve client robustness behind various firewalls and NAT configurations. Finally, further development can focus on Performance Optimization (e.g., using larger buffers) and the implementation of a Graphical User Interface for improved client usability.