# NEW-NIT-DELHI-LOGO1.png

# CSB 351: Network Programming

**File Transfer protocol (FTP) implementation**

**Submitted By:**

**Nishant Raj (211210042)**

**Submitted To: Dr. Preeti Mehta**

**Department of**  **Computer science and engineering**

***Introduction***

**File Transfer Protocol (FTP)** is a standard network protocol used for transferring files between a client and a server on a computer network. An FTP server is a software application that runs on a server and provides access to files, allowing users to upload, download, and manage files remotely.

FTP servers are commonly used for various purposes, including:

1. \*\*File Sharing\*\*: FTP servers allow users to share files with others over a network or the internet. This is particularly useful for businesses that need to exchange large files or collaborate on projects remotely.

2. \*\*Website Hosting\*\*: Many websites are hosted on FTP servers, where webmasters can upload and manage website files such as HTML pages, images, and scripts.

3. \*\*Backup and Storage\*\*: FTP servers can serve as a centralized location for storing backup files and data. Users can upload files to the server for safekeeping and access them from anywhere with an internet connection.

4. \*\*Software Distribution\*\*: Software developers often use FTP servers to distribute their software packages and updates to users. This allows for efficient distribution of large files to a wide audience.

5. \*\*Remote File Access\*\*: FTP servers enable users to access their files remotely from any location with internet access. This is particularly useful for accessing important documents or data while traveling or working from home.

FTP servers typically require authentication for access, meaning users need to provide a username and password to connect to the server. Some FTP servers also support secure connections using protocols like FTPS (FTP over SSL/TLS) or SFTP (SSH File Transfer Protocol) to encrypt data transmission and enhance security.

Overall, FTP servers play a crucial role in facilitating file transfer and management for individuals and organizations across various industries.

**Problem Statement**

Design and implement a simplified File Transfer Protocol (FTP) using the Socket API in

Python. The goal is to create a basic FTP system that allows users to upload and download

files to and from a central server. Focus on the functionality of transferring files securely and

efficiently.

Requirements:

**1. Server-Client Architecture:**

Implement a server that manages file storage and multiple client connections. Clients

should be able to connect to the server to upload and download files.

**2. User Authentication:**

Implement a user authentication mechanism. Users must provide a valid username

and password to access the FTP server. Store user credentials securely on the server.

**3. File Upload and Download:**

Enable clients to upload files to the server and download files from the server.

Implement commands such as PUT for uploading and GET for downloading files.

Ensure data integrity during file transfers.

**4. Directory Listing:**

Implement a command to list the contents of the server&#39;s file directory. Clients should

be able to request a list of available files on the server.

**5. File Deletion (Optional):**

Extend the application to support file deletion. Clients with the appropriate

permissions should be able to delete files from the server.

**6. Permissions and Access Control (Optional):**

Implement a basic permissions system to control which users have access to specific

directories or files. Consider implementing roles such as read-only and read-write.

**7. Error Handling:**

Implement robust error handling mechanisms to deal with issues such as

authentication failures, invalid commands, or file transfer interruptions. Provide

informative error messages to users.

**8. User Interface (Optional):**

Develop a basic command-line or graphical user interface for the client to enhance the

user experience. The interface should include options for connecting to the server,

uploading and downloading files, and managing user sessions.

**9. Logging and Auditing (Optional):**

Implement a logging mechanism to record important events, such as user logins, file

transfers, and errors. Ensure that the logs are informative and easy to interpret.

**10. Documentation:**

Provide detailed documentation explaining the system architecture, communication

protocol used, and instructions for setting up and running the server-client application.

**11. Testing:**

Conduct thorough testing to ensure the reliability and security of the FTP application.

Include test cases for various scenarios, including successful file transfers,

authentication processes, and error handling.

**Implementation**

Connection:

The first step to start working with FTP server using Python is to

connect to the FTP server.

Serverside code:

import socket

import os

# Define server parameters

HOST = 'localhost'

PORT = 5000

BUFFER\_SIZE = 1024

FILES\_DIRECTORY = 'files'

# Function to handle file upload

def upload\_file(conn, filename):

    with open(os.path.join(FILES\_DIRECTORY, filename), 'wb') as f:

        while True:

            data = conn.recv(BUFFER\_SIZE)

            if not data:

                break

            f.write(data)

    print(f"File '{filename}' uploaded successfully")

# Function to handle file download

def download\_file(conn, filename):

    file\_path = os.path.join(FILES\_DIRECTORY, filename)

    if os.path.exists(file\_path):

        with open(file\_path, 'rb') as f:

            data = f.read(BUFFER\_SIZE)

            while data:

                conn.send(data)

                data = f.read(BUFFER\_SIZE)

        print(f"File '{filename}' downloaded successfully")

    else:

        print(f"File '{filename}' does not exist on server")

# Main server logic

def main():

    os.makedirs(FILES\_DIRECTORY, exist\_ok=True)

    with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

        s.bind((HOST, PORT))

        s.listen()

        print("Server listening...")

        while True:

            conn, addr = s.accept()

            print(f"Connected by {addr}")

            with conn:

                while True:

                    data = conn.recv(BUFFER\_SIZE).decode()

                    if not data:

                        break

                    command, filename = data.split()

                    if command == 'PUT':

                        upload\_file(conn, filename)

                    elif command == 'GET':

                        download\_file(conn, filename)

                    else:

                        print("Invalid command")

            print(f"Connection with {addr} closed")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Clientside code:

import socket

# Define server parameters

SERVER\_HOST = 'localhost'

SERVER\_PORT = 5000

BUFFER\_SIZE = 1024

# Function to connect to the server

def connect\_to\_server():

    s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    s.connect((SERVER\_HOST, SERVER\_PORT))

    return s

# Function to upload file

def upload\_file(sock, filename):

    try:

        with open(filename, 'rb') as f:

            sock.sendall(f"PUT {filename}".encode())

            data = f.read(BUFFER\_SIZE)

            while data:

                sock.sendall(data)

                data = f.read(BUFFER\_SIZE)

        print(f"File '{filename}' uploaded successfully")

    except Exception as e:

        print(f"Error uploading file '{filename}': {e}")

# Function to download file

def download\_file(sock, filename):

    try:

        sock.sendall(f"GET {filename}".encode())

        with open(filename, 'wb') as f:

            while True:

                data = sock.recv(BUFFER\_SIZE)

                if not data:

                    break

                f.write(data)

        print(f"File '{filename}' downloaded successfully")

    except Exception as e:

        print(f"Error downloading file '{filename}': {e}")

# Function to list files

def list\_files(sock):

    try:

        sock.sendall(b"LIST")

        files = sock.recv(BUFFER\_SIZE).decode()

        print("Files on server:")

        print(files)

    except Exception as e:

        print(f"Error listing files: {e}")

# Function to delete file

def delete\_file(sock, filename):

    try:

        sock.sendall(f"DELETE {filename}".encode())

        response = sock.recv(BUFFER\_SIZE).decode()

        print(response)

    except Exception as e:

        print(f"Error deleting file '{filename}': {e}")

# Main client logic

def main():

    try:

        while True:

            print("\nSelect an option:")

            print("1. Connect to server")

            print("2. Upload file")

            print("3. Download file")

            print("4. List files on server")

            print("5. Delete file")

            print("6. Exit")

            choice = input("Enter choice: ")

            if choice == '1':

                sock = connect\_to\_server()

                print("Connected to server")

            elif choice == '2':

                filename = input("Enter filename to upload: ")

                upload\_file(sock, filename)

            elif choice == '3':

                filename = input("Enter filename to download: ")

                download\_file(sock, filename)

            elif choice == '4':

                list\_files(sock)

            elif choice == '5':

                filename = input("Enter filename to delete: ")

                delete\_file(sock, filename)

            elif choice == '6':

                sock.sendall(b"exit")

                break

            else:

                print("Invalid choice")

    except Exception as e:

        print(f"Client error: {e}")

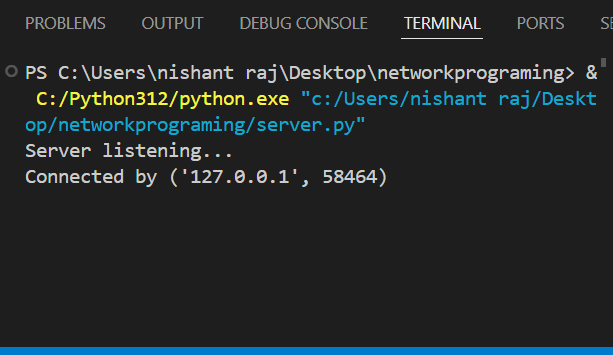
    finally:

        sock.close()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Output:-



**Authentication:**

In order to connect to the FTP server, you will need to know some

credentials:

\* Host

\* Username

\* Password

**Serverside code;**

import socket

import os

import hashlib

# Define server parameters

HOST = 'localhost'

PORT = 5000

BUFFER\_SIZE = 1024

FILES\_DIRECTORY = 'files'

USERS = {'user1': 'password1', 'user2': 'password2'}

# Function to handle file upload

def upload\_file(conn, filename):

    with open(os.path.join(FILES\_DIRECTORY, filename), 'wb') as f:

        while True:

            data = conn.recv(BUFFER\_SIZE)

            if not data:

                break

            f.write(data)

    print(f"File '{filename}' uploaded successfully")

# Function to handle file download

def download\_file(conn, filename):

    file\_path = os.path.join(FILES\_DIRECTORY, filename)

    if os.path.exists(file\_path):

        with open(file\_path, 'rb') as f:

            data = f.read(BUFFER\_SIZE)

            while data:

                conn.send(data)

                data = f.read(BUFFER\_SIZE)

        print(f"File '{filename}' downloaded successfully")

    else:

        print(f"File '{filename}' does not exist on server")

# Function to authenticate user

def authenticate(conn):

    conn.sendall(b"Username: ")

    username = conn.recv(BUFFER\_SIZE).decode().strip()

    conn.sendall(b"Password: ")

    password = conn.recv(BUFFER\_SIZE).decode().strip()

    hashed\_password = hashlib.sha256(password.encode()).hexdigest()

    if username in USERS and USERS[username] == hashed\_password:

        conn.sendall(b"Authenticated")

        return True

    else:

        conn.sendall(b"Invalid username or password")

        return False

# Main server logic

def main():

    os.makedirs(FILES\_DIRECTORY, exist\_ok=True)

    with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

        s.bind((HOST, PORT))

        s.listen()

        print("Server listening...")

        while True:

            conn, addr = s.accept()

            print(f"Connected by {addr}")

            if authenticate(conn):

                while True:

                    data = conn.recv(BUFFER\_SIZE).decode()

                    if not data:

                        break

                    command, filename = data.split()

                    if command == 'PUT':

                        upload\_file(conn, filename)

                    elif command == 'GET':

                        download\_file(conn, filename)

                    else:

                        print("Invalid command")

                print(f"Connection with {addr} closed")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Client Side code;**

import socket

# Define server parameters

SERVER\_HOST = 'localhost'

SERVER\_PORT = 5000

BUFFER\_SIZE = 1024

# Function to authenticate user

def authenticate(sock):

    while True:

        response = sock.recv(BUFFER\_SIZE).decode()

        print(response)

        if response == "Authenticated":

            return True

        elif response == "Invalid username or password":

            return False

# Main client logic

def main():

    with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

        s.connect((SERVER\_HOST, SERVER\_PORT))

        if not authenticate(s):

            print("Authentication failed.")

            return

        while True:

            command = input("Enter command (PUT/GET <filename>): ")

            s.sendall(command.encode())

            if command == 'exit':

                break

            elif command.startswith('GET'):

                filename = command.split()[1]

                with open(filename, 'wb') as f:

                    while True:

                        data = s.recv(BUFFER\_SIZE)

                        if not data:

                            break

                        f.write(data)

                print(f"File '{filename}' downloaded successfully")

            elif command.startswith('PUT'):

                filename = command.split()[1]

                with open(filename, 'rb') as f:

                    data = f.read(BUFFER\_SIZE)

                    while data:

                        s.sendall(data)

                        data = f.read(BUFFER\_SIZE)

                print(f"File '{filename}' uploaded successfully")

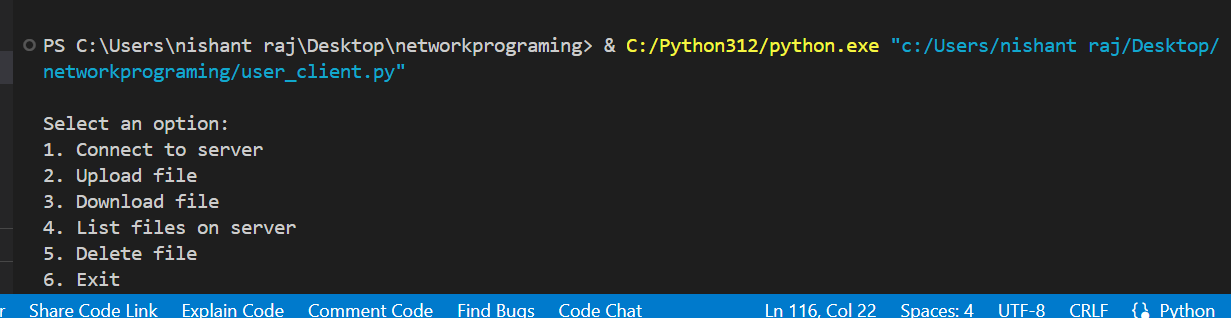
            else:

                print("Invalid command")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Output:**

****

Now the connection is established and we can verify it by

requesting a welcome message from the FTP server

**File Upload :**

In this section we will explore how to upload a file to the FTP server.

So far we have create a new folder in FTP called “pyshark” and set

it as the current working directory.

Server code:-

import socket

import os

# Define server parameters

HOST = 'localhost'

PORT = 5000

BUFFER\_SIZE = 1024

FILES\_DIRECTORY = 'files'

# Function to handle file upload

def upload\_file(conn, filename):

    with open(os.path.join(FILES\_DIRECTORY, filename), 'wb') as f:

        while True:

            data = conn.recv(BUFFER\_SIZE)

            if not data:

                break

            f.write(data)

    print(f"File '{filename}' uploaded successfully")

# Function to handle file download

def download\_file(conn, filename):

    file\_path = os.path.join(FILES\_DIRECTORY, filename)

    if os.path.exists(file\_path):

        with open(file\_path, 'rb') as f:

            data = f.read(BUFFER\_SIZE)

            while data:

                conn.send(data)

                data = f.read(BUFFER\_SIZE)

        print(f"File '{filename}' downloaded successfully")

    else:

        print(f"File '{filename}' does not exist on server")

# Main server logic

def main():

    os.makedirs(FILES\_DIRECTORY, exist\_ok=True)

    with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

        s.bind((HOST, PORT))

        s.listen()

        print("Server listening...")

        while True:

            conn, addr = s.accept()

            print(f"Connected by {addr}")

            while True:

                data = conn.recv(BUFFER\_SIZE).decode()

                if not data:

                    break

                command, filename = data.split()

                if command == 'PUT':

                    upload\_file(conn, filename)

                elif command == 'GET':

                    download\_file(conn, filename)

                else:

                    print("Invalid command")

            print(f"Connection with {addr} closed")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

client code:-

import socket

# Define server parameters

SERVER\_HOST = 'localhost'

SERVER\_PORT = 5000

BUFFER\_SIZE = 1024

# Main client logic

def main():

    with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

        s.connect((SERVER\_HOST, SERVER\_PORT))

        while True:

            command = input("Enter command (PUT/GET <filename>): ")

            s.sendall(command.encode())

            if command == 'exit':

                break

            elif command.startswith('GET'):

                filename = command.split()[1]

                with open(filename, 'wb') as f:

                    while True:

                        data = s.recv(BUFFER\_SIZE)

                        if not data:

                            break

                        f.write(data)

                print(f"File '{filename}' downloaded successfully")

            elif command.startswith('PUT'):

                filename = command.split()[1]

                with open(filename, 'rb') as f:

                    data = f.read(BUFFER\_SIZE)

                    while data:

                        s.sendall(data)

                        data = f.read(BUFFER\_SIZE)

                print(f"File '{filename}' uploaded successfully")

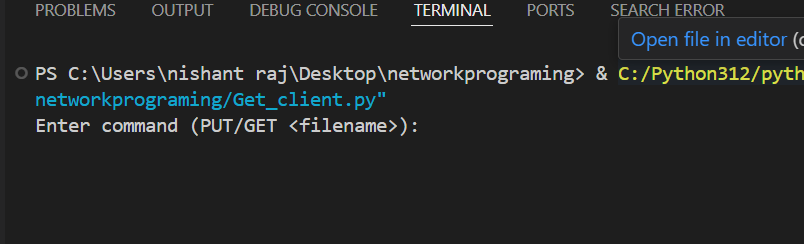
            else:

                print("Invalid command")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**OUTPUT**

****

In order to upload the file to FTP server, we will use

the .storbinary() method of the FTP class. This method initiates a

file transfer from FTP client to FTP server using the FTP command

STOR.

First we will open the sample file “file1.txt” as a binary file with read

mode and then perform the FTP file transfer (we will name the

transferred file on the server as “uploaded\_file”):

**Libraries Used:**

● Socket Library: We used the socket library to establish network connections between

the FTP server and clients. With this library, we were able to create sockets for

communication, allowing clients to connect to the server and transfer files securely.

● os Library: Leveraging the os library, we managed file operations within the FTP

server. This library provided us with functions to handle file uploads, downloads, and

directory listings. We could interact with the file system, ensuring smooth file management

processes.

● struct Library: The struct library played a crucial role in encoding and decoding data

structures for network communication. By using this

library, we could pack and unpack binary data, facilitating the transmission of file details

such as file names and sizes between the server and clients.

● json Library: With the json library, we stored and retrieved user credentials securely

in a JSON file. This library allowed us to parse JSON data, ensuring that user authentication

information was managed efficiently and safely.

● datetime Library: Utilizing the datetime library, we generated timestamps for

logging events within the FTP server. This library enabled us to record the exact date and

time of user actions, providing valuable insights into system activities.

● time Library: We employed the time library to measure the elapsed time during file

transfers. By utilizing functions from this library, we could calculate the duration of file

uploads and downloads, optimizing performance and user experience.

● threading Library: Leveraging the threading library, we implemented

multithreading support in the FTP server. This library allowed us to handle multiple client

connections concurrently, enhancing the scalability and responsiveness of the server.

**Implementation Steps:**

1) Setting Up Server-Client Architecture: Server Configuration:

● We started off by creating a Python script named server.py.

● Using Python's socket library, we established a server socket to listen for incoming

connections.

● Defined constants for the server IP address, port number, and buffer size to facilitate

communication.

● With threading, we managed to handle multiple client connections simultaneously,

ensuring efficient server operation.

**Client Configuration:**

● In parallel, we developed a client script named client.py.

● Leveraging the socket library, we enabled clients to connect to the server and

perform file transfer operations.

● We set up constants for the server IP address, port number, and buffer size to ensure

seamless communication.

● We also implemented functions in the client script to facilitate connection

establishment and command transmission to the server.

2) Implementing User Authentication:

● We created an authentication module named auth.py to handle user authentication.

● Utilizing Python's json library, we securely stored user credentials in a JSON file

named users.json.

● We implemented functions within the module to authenticate users, add new users,

and delete existing users.

● Before storage, we ensured that the passwords were hashed to enhance security.

**3) Handling File Transfer Operations:**

Uploading and Downloading Files:

● We developed functions in the server script (server.py) to manage file upload (upld)

and download (dwld) operations.

● We utilized struct for encoding and decoding data structures during file transfer.

● We also ensured that files were transmitted in chunks, guaranteeing efficient and

reliable transfer between the server and clients.

Deleting Files:

● We implemented a function (delf) in the server script to handle file deletion requests

from clients.

● We also validated file existence and performed deletion operations after

authentication and permission checks.

**4) Implementing Directory Listing:**

● Within the server script (server.py), we created a function (list\_files) to list the

contents of the server's file directory.

● We also sent the list of available files, along with their sizes, to clients, allowing them

to view and select files for download or deletion.

**5) Logging and Auditing:**

● We developed a logging module named log.py to record important events such as

user logins, file transfers, and deletions.

● We also utilized Python's datetime library to generate timestamps for logging

entries, aiding in tracking and auditing system activities

.

**6) Error Handling:**

● Throughout the development process, we implemented robust error handling

mechanisms to address authentication failures, invalid commands, and file transfer

interruptions.

● We also ensured that informative error messages were provided to users, guiding

them through troubleshooting procedures.

**7) Testing:**

● We conducted comprehensive testing to verify the reliability and security of the FTP

application.

● We also designed test cases covering various scenarios, including successful file

transfers, authentication processes, error handling, and logging functionalities, ensuring the

robustness of the system.

**Conclusion:** The implemented FTP system offers a secure and efficient file transfer

solution, suitable for small-scale file management tasks. Future enhancements could include

additional features such as permission management, enhanced logging

capabilities, and improved user interfaces to further enhance usability and functionality.

**Future Scope**: Although not explicitly implemented yet, we considered the development

of a user interface (CLI or GUI) to enhance the user experience. We planned features such as

connecting to the server, managing file transfers, sessions, and viewing logs to provide users

with a more intuitive interaction environment.

Thank You