CPSC 471 Computer Communications Programming Assignment
Simple FTP and Socket Programming

Author:
Jordan Morris

Affiliation:
California State University, Fullerton
Computer Science Department

Course:
Computer Networks/Communications

Programming Language:
Python

Submission Date:
11/29/2024

Abstract

This project implements a simplified client-server architecture to support basic File Transfer Protocol (FTP)-style functionality using Python. The system handles three commands: ls (to list files on the server), put (to upload files from the client to the server), and get (to download files from the server to the client). The primary goal of this assignment was to explore the complexities of designing and implementing communication protocols and to appreciate the nuances of real-world socket programming in a simplified context.

Key objectives included understanding how FTP works, designing a robust communication protocol using control and data channels, and managing real-time file transfer across a network. The project uses Python's socket programming API for handling network communication and the file system.

The methodology involved setting up a server to listen to client requests and handling control and data connections. The client communicates with the server over a control channel to send commands and directs to ephemeral ports for data transfers. Comprehensive error handling and print statement updates ensure reliable operations for all commands.

The results demonstrate successful implementation of the ls, get, and put commands, the understanding of socket programming, simple protocol design, and real-world networking challenges. This project highlights the relevance of Python in rapidly prototyping and deploying networked systems while building foundational knowledge in computer networking concepts.

Keywords

Python, Computer Networks, Socket Programming, Protocol Simulation, FTP, Ephemeral Ports

---

1. Introduction

1.1
Background

Network programming is a cornerstone of modern technology, enabling communication and data exchange between devices across local and global networks. It forms the basis of countless applications, from web browsers to cloud storage and IoT devices. Understanding network programming is crucial for developing scalable, efficient, and reliable systems in an increasingly interconnected world.

Python has emerged as a popular language for network programming due to its simplicity, rich library support, and versatility. Libraries like Socket and higher-level frameworks such as Flask and Django make Python an excellent choice for both low-level and high-level networking tasks. Python's simple syntax makes it an easy-to-understand language for getting into socket programming.

This project focuses on the client-server model. The client-server architecture is used in various protocols, including HTTP, FTP, and SMTP, making it a good topic for understanding network interactions. By implementing a simplified FTP system, this project demonstrates the principles of socket programming, the management of control and data channels, and the intricacies of designing robust communication protocols. Through Python, the project is able to highlight the practical capabilities of these

1.2
Objective

The primary goal of this project was to implement a simple FTP (File Transfer Protocol) client-server communication system. This project aimed to establish a basic understanding of the fundamentals of such systems, including the design and execution of protocols for file transfer operations.

Through this implementation, the focus was on achieving core functionalities like listing files on the server (ls command), uploading files from the client to the server (put command), and downloading files from the server to the client (get command). The project emphasized the use of Python’s socket programming capabilities to create and manage a reliable communication framework between the client and server.

1.3
Scope

The project involves the design and implementation of a simplified FTP client-server system using Python. The primary functionalities include file management operations like listing files (ls), uploading files from the client to the server (put), and downloading files from the server to the client (get).

What the Project Covers:

- Establishing a basic client-server architecture using Python's socket programming.
- Enabling communication between the client and server over TCP.
- Supporting fundamental FTP operations (ls, put, and get) in a secure and reliable manner.
- Managing file transfers efficiently with separate data connections (ephemeral) for file uploads and downloads.
- Simple handling of improper commands, lack of existence of files on both client and server-side
    - Timely ends of protocol communication when file path is not found on either side, resolves correctly.
- Large file transfers are supported
- Response Codes are used but not those specified by the RF 959

What the Project Does Not Cover:

- Implementation of a complete FTP protocol as defined in RFC 959 [1].
    - Advanced FTP features such as authentication, encryption, or secure data transfers.
- Detailed optimization for high-performance or large-scale file transfer systems.

---

2. Related Work

File transfer has been a foundational requirement in networking, driving the development of numerous protocols and technologies. Among these, the **File Transfer Protocol (FTP)**, as defined in **RFC 959** [1], is one of the earliest and most widely implemented standards for transferring files over a network. It laid the groundwork for structured and efficient file exchanges, supporting a range of operations such as listing directories, downloading files, and uploading files.

Other protocols that have influenced or complemented FTP that are notably relevant and discussed in our class include:

1. **Hypertext Transfer Protocol (HTTP):**
    - While primarily designed for web content delivery, HTTP is also used for file transfers, especially in modern web-based systems. Its simplicity and ubiquity make it a popular choice for one-off file downloads or uploads.
2. **Sockets Programming:**

- The use of sockets for implementing custom client-server communication has been a cornerstone of networking. Sockets provide a low-level API that enables developers to design tailored solutions for specific applications, such as the simplified FTP system in this project.

Python libraries and frameworks:

The project utilizes several key Python libraries to implement the simplified FTP system, each playing a critical role in ensuring smooth client-server communication and handling concurrent operations. Below is an overview of the libraries used:

1. **socket:**
   - The cornerstone of the project, the socket library provides low-level access to the system's networking stack. It enables the creation of both server and client sockets for TCP communication. Key features include connection establishment, data transmission, and handling multiple ports for control and data channels in the FTP protocol.
2. **os:**
   - The os library is used for file system interactions. It allows the server to perform file operations such as listing files in the current directory (ls command), checking file existence, and reading or writing files during file transfer operations (get and put commands). This ensures seamless integration with the underlying operating system.
3. **threading:**
   - To support multiple simultaneous clients, the threading library is employed. Each client connection is handled in a separate thread, allowing the server to manage multiple file transfer sessions concurrently without blocking. This ensures scalability and efficient use of server resources.

---

3. System Design and Architecture

3.1 System Overview

The simplified FTP system is built using the client-server model. The architecture comprises two main components: the server-side application and the client-side application. Communication between the client and server is facilitated via TCP sockets. The system uses separate control and

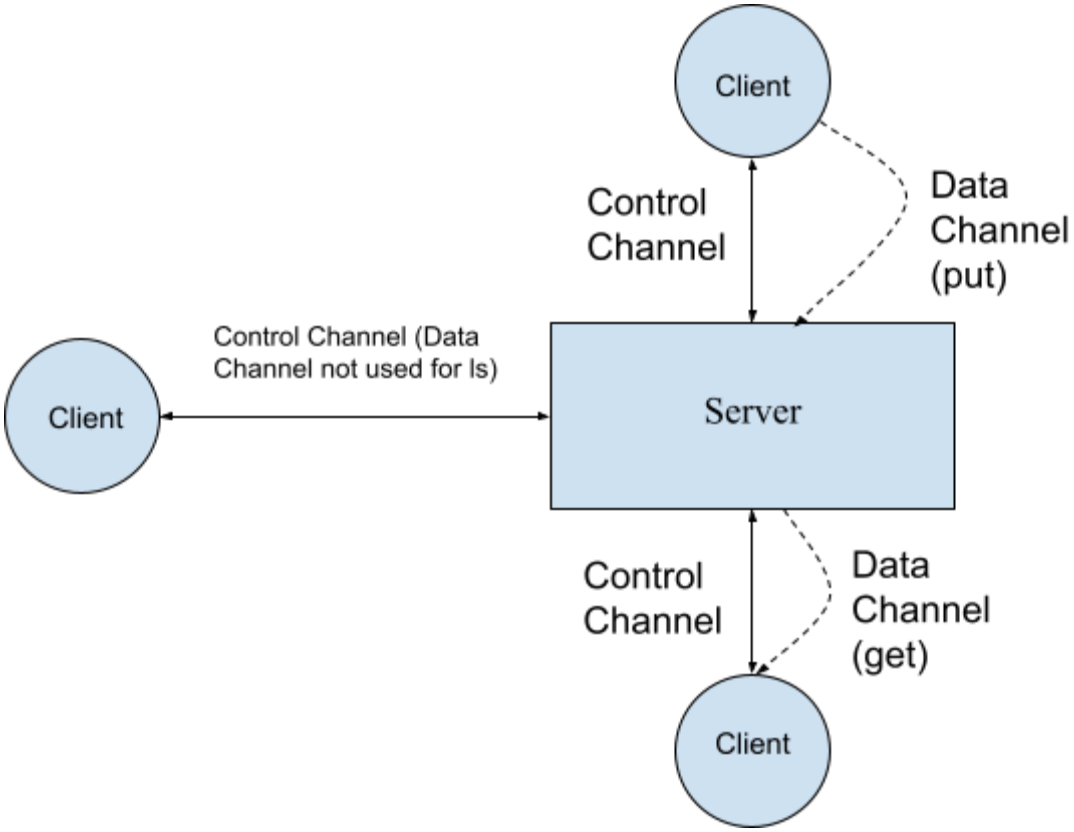
data channels as specified by the guidelines to utilize ephemeral ports so that the two sides don't have to interfere with each other.

The server listens for incoming client connections and processes commands such as ls (list files), get (download a file), and put (upload a file). Each client request is managed in a dedicated thread to allow concurrent operations. The client sends commands to the server, which processes the requests and performs file operations accordingly.

3.2 Architecture Diagram

Diagram Explanation

1. **Client Application:**
    - Initiates requests (ls, get, put, quit) to the server.
    - Establishes a control connection with the server to send commands and receive responses.
    - Creates data connections for file transfer operations (get and put).
2. **Server Application:**
    - Waits for client connections on a predefined port.
    - Spawns a new thread for each connected client to handle operations.
    - Executes commands based on client requests, listing files, or transferring files through a data socket.
3. **Control and Data Channels:**
    - **Control Channel:** Manages command exchange and session control.
    - **Data Channel:** Handles file transfers for get and put commands.



3.3 Modules

1. **Client-Side Application**
    - **Functionality:**
      The client initiates communication with the server. It takes user commands (ls, get, put, quit), sends them to the server, and manages file uploads/downloads using separate data sockets.
    - **Key Functions:**
        - handle_get: Downloads files from the server.
        - handle_put: Uploads files to the server.
        - Command validation and error handling.
2. **Server-Side Application**
    - **Functionality:**
      The server accepts client connections and processes commands in a multithreaded environment. It handles file system operations and manages data connections for file transfer.
    - **Key Functions:**
        - list_files: Lists the files in the server directory for ls commands.
        - handle_get: Sends requested files to the client.
        - handle_put: Receives uploaded files from the client.

3. **Communication Protocol Implementation**
    - **Structure:**
        - The server uses a control socket for command exchange and an ephemeral socket for file transfers.
        - The client synchronizes with the server using a two-step process: command processing (control) and data transfer (data channel).
    - **Key Features:**
        - Error messages for invalid commands or file-related issues.
        - Confirmation messages for successful file transfers.

---

4. Implementation

4.1 Technologies Used

1. Socket Programming:
    a. socket: Provides low-level access to network communication (e.g., creating, binding, connecting sockets for data exchange).
2. Operating System Interactions:
    a. os: Used for file system operations, such as listing files in a directory (ls), checking file existence, and file handling.
3. Multithreading:
    a. threading: Used to handle multiple clients concurrently by creating separate threads for each client connection.
4. Testing:
    a. test.py: Can be used to write tests to validate server and client logic. Written by hand to assert that the operations work as intended and files are placed in proper places.
5. Data Handling:
    a. File I/O: Handled directly using Python's built-in file handling functions (open, read, write), especially for file transfers in binary mode.

4.2
Key Features

1. Establishing a TCP Connection
    - Utilizes the socket library to create, bind, and manage TCP connections between the client and server.

2. Simulating an FTP-like Protocol
    - Implements commands such as ls (list server-side files), get (download files from the server), and put (upload files to server), similar to the functionality of FTP.
3. Control and Data Channels
    - Control Channel: Handles bidirectional command exchange and session management (e.g., parsing ls, get, put, quit requests).
    - Data Channel: Establishes separate, ephemeral connections for file transfers during get and put operations, ensuring efficient and isolated data handling.
4. Concurrent Client Handling
    - Leverages threading to allow multiple clients to connect to the server and operate simultaneously without blocking other connections.
5. File Transfer Operations
    - Supports both uploading (client-to-server) and downloading (server-to-client) of files, with real-time transmission of file data in binary format.
6. Dynamic File Listing
    - Implements the ls command to retrieve and display the list of files available in the server's directory.
7. Error Handling
    - Includes checks for file existence during transfers and appropriate success or failure responses to client commands.

4.3
Code Overview

Server Code

1. **Socket Creation and Listening for Clients**:
    - The server creates a TCP socket, binds it to 0.0.0.0:2121, and listens for incoming connections:

```
# Create a TCP socket for the server

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)



# Bind the server to all interfaces on port 2121

server.bind(("0.0.0.0", 2121))



# Start listening for incoming connections (with a backlog of 5)
```

```
server.listen(5)
```

2. **Handling Client Requests**:
    - Once a connection is accepted, a new thread is spawned to handle client commands (ls, get, put, quit), with different functions for each operation:

```
# Accept a new client connection

    client_socket, addr = server.accept()

    print(f"Accepted connection from {addr}")



    # Start a new thread to handle the client connection

    client_handler = threading.Thread(

      target=handle_client, args=(client_socket,))

    client_handler.start()
```

    - handle_client() receives commands (ls, get, put), interacts with the filesystem, and manages file transfers.
3. **File Upload (put) and Download (get)**:
    - For put, the server listens for incoming data over a separate data connection and writes it to a file:

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

  data = data_conn.recv(BUFFER_SIZE)

  while data:

    f.write(data)

    data = data_conn.recv(BUFFER_SIZE)
```

    - For get, the server sends file data in chunks over a new data socket:

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

  data = f.read(BUFFER_SIZE)

  while data:
```

```
data_conn.send(data)

data = f.read(BUFFER_SIZE)
```

Client Code

4. **Socket Creation and Listening for Clients**:
    - The client creates a TCP socket and connects to the server's IP and port:

```
# Create a TCP socket for the client

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the client socket to the server's IP and port

client.connect((server_ip, server_port))
```

5. **Command Handling**:
    - The client sends commands to the server and handles file transfers based on user input (get, put, ls, quit):

```
# Prompt the user for a command

command = input("ftp> ")


# Handle the 'get' command for downloading a file

if command.startswith("get"):

    # Extract the filename from the command

    filename = command.split()[1]

    # Call the function to handle file download

    handle_get(client, filename)


# Handle the 'put' command for uploading a file

elif command.startswith("put"):
```

```
        # Extract the filename from the command
        filename = command.split()[1]
        # Call the function to handle file upload
        handle_put(client, filename)

    # Handle the 'ls' command to list files on the server
    elif command == "ls":
        # Send the ls command to the server
        client.send(command.encode())
        # Receive the list of files from the server and decode it
        file_list = client.recv(BUFFER_SIZE).decode()
        print("Files on server:")
        print(file_list)  # Display the list of files
        # Send confirmation back to the server that files have been listed
        client.send("FILES LISTED".encode())

    # Handle the 'quit' command to exit the client
    elif command == "quit":
        # Inform the server of the disconnection
        client.send(command.encode())
        client.close()  # Close the client socket
        break  # Exit the loop and end the client session
```

6. **File Upload (put) and Download (get):**
    - For put, the client reads a file in chunks and sends it to the server:

```
with open(filename, 'rb') as f:
    data = f.read(BUFFER_SIZE)
    while data:
        data_socket.send(data)
        data = f.read(BUFFER_SIZE)
```

- For get, the client requests a file and saves it by receiving data from the server:

```
with open(filename, 'wb') as f:
    data = data_socket.recv(BUFFER_SIZE)
    while data:
        f.write(data)
        data = data_socket.recv(BUFFER_SIZE)
```

4.4
Algorithms

1. File Transfer Algorithm
    - Back-to-Back Chunk Transfer: Files are transmitted in fixed-size chunks (using BUFFER_SIZE) to balance memory usage and network bandwidth. This prevents overwhelming the system with large data reads or sends.
    - Stop-and-Wait Flow Control: Each chunk is sent, and the system waits for confirmation of completion (e.g., get or put) before proceeding with additional data transmission or closing the connection. This is handled in the handle_client function of the server to be executed at the end of each respective handle_”command” function. (ex: only after handle_put is executed, does the server send a response confirmation through the data channel to appease the client waiting for a confirmation of uploaded files).
2. Multithreading for Concurrency
    - Thread-per-Client Model: A new thread is spawned for each client connection, allowing the server to handle multiple clients simultaneously without blocking other connections. This approach is simple yet effective for moderate traffic loads which is much simpler to implement in controlled practice than the large scale considerations on paper.

3. Control and Data Channel Management
    - Basic Command Parsing: The server parses incoming commands from the control channel to identify the requested operation (ls, get, put, or quit). Based on the parsed command, it routes the workflow to the appropriate handling logic.
    - Port Allocation for Data Channel: During file transfers (get or put), a new ephemeral port is dynamically allocated on the server side for the data channel, ensuring clean separation from the control channel.
4. Error Handling and Feedback Loop
    - Response Checking: Both the client and server exchange success or failure acknowledgments for commands and transfers, ensuring that both sides are synchronized.
    - File Existence Check: Before initiating a file transfer, the server verifies the existence of requested files and informs the client of success or failure, avoiding unnecessary resource use.
5. Basic Congestion Control
    - Buffer Management: The system limits the size of each transmission to avoid overwhelming the network. Although simple, this aligns with the principles of congestion control by minimizing large bursts of traffic.

---

5. Results and Discussion

5.1
Testing Scenarios

The following test cases focus on different file operations and server-client interactions:

- **File Uploads (PUT)**: The tests include uploading a valid file, a non-existent file, an existing file, a large file, multiple files in quick succession, and files with special characters or the maximum allowed filename length. They also cover uploading during high load and checking server disk space limits during uploads.
- **File Downloads (GET)**: Tests for downloading files include retrieving a valid file, attempting to download a non-existent file, downloading while another client uploads, and checking if files can still be downloaded after a client disconnects during upload.
- **File Listings and Other Operations (LS & Others)**: These tests involve listing files on the server, checking the server's behavior when the directory is empty, and verifying server responses to invalid commands. Additional scenarios test client disconnections during file uploads, file deletion from the server, client file cleanup after upload, and server restart persistence.

5.2
Performance Metrics

The client-server structure demonstrated reliable and efficient operation, successfully transferring files of varying sizes between the server and the requesting client under diverse scenarios. Key observations include:

1. Accuracy and Robustness:
    - File transfers are consistently completed without errors, even for large files or during high-frequency requests from multiple clients.
    - Server responses to commands (ls, get, put, quit) were accurate, ensuring smooth communication between the client and server.
2. Scalability:
    - The multithreaded server efficiently handled concurrent connections from multiple clients, maintaining responsiveness and stability under simultaneous requests.
3. Transfer Speed:
    - File transfer operations were conducted within expected timeframes, leveraging the BUFFER_SIZE configuration to optimize transmission while avoiding network congestion.
4. Error Recovery:
    - The system gracefully managed edge cases, such as non-existent files (get requests) and interrupted transfers, providing appropriate feedback to the client without crashing or halting operations.
5. Basic Congestion Control:
    - The system limits the size of each transmission (buffer) to avoid overwhelming the network. Although simple, this aligns with the principles of congestion control by minimizing large bursts of traffic.

5.3
Challenges and Resolutions

One of the key challenges in this project stemmed from the lack of a clear, documented protocol design early on. This oversight led to issues in the handle_put function on both the client and server sides, where the sequence of operations did not align correctly between the two. In the initial iterations of the client program, the client inadvertently sent the put command twice—once in the main() function and again when handle_put was called. As a result, the server would begin the process of receiving the file, while the client, still one step behind, continued interacting through the control channel. This mismatch caused a situation where the server sent

the ephemeral data port number back to the client, but the client did not properly process it due to the command duplication. The client would display the data port in response to the server's output, rather than executing the intended command.

This issue was resolved by restructuring the command send functions to ensure each command was executed on a per-command basis. By making the transmission of commands properly sequential, the client and server were better synchronized. Additionally, another problem was identified: if the client attempted to upload a file that did not exist in the local directory, it would still send the command to the server. This led to unnecessary resource consumption. To address this, the handle_put function was modified to check for the file's existence before initiating the upload. If the file was absent, the command would not be sent, thus preventing the server from engaging in an unnecessary and incomplete file transfer process.

A more minor issue that arose was the need for confirmation to properly document the interactions between the client and server. Each function required an acknowledgment to ensure that the command was processed successfully. Without these confirmations, the system could not reliably track whether a command had executed properly. This was addressed by implementing response codes sent through the control channel after each handle function executed, allowing for clearer communication and ensuring the success or failure of each operation was documented appropriately.

5.4
Discussion

Strengths

- Clear Separation of Control and Data Channels: One of the strengths of this project is the clear distinction between the control and data channels. The control channel handles the communication of commands, while the data channel is dedicated to file transfers. This design mirrors protocols discussed in class like FTP, ensuring efficiency in managing different types of data. By using separate channels for commands and file transfer, the solution maintains organization and avoids congestion in the communication process.
- Concurrency and Scalability: The use of threading to handle multiple clients simultaneously enhances the scalability of the server if needed. Each client connection is handled in a separate thread, allowing for concurrent file transfers and command execution without interfering with other clients. This is essential for real-world applications where multiple users would be interacting with the server at once.
- Error Handling and Confirmation: The usage of simple response codes and command confirmations between the client and server ensures reliable error handling. It allows both sides to track the status of the operations, confirming success or failure after each command. This transparency provides a foundation for debugging and ensures that clients are aware of the system's current state should problems arise.
- Simplicity and Modularity: The design is simple yet modular, with functions dedicated to specific tasks like file listing (ls), file downloading (get), and file uploading (put). This

makes the system easy to extend or modify. New commands can be added without disrupting existing functionality, promoting maintainability.

Limitations

- File Size and Transfer Speed: While the system handles file transfers effectively, there is no built-in mechanism for optimizing transfer speeds, especially for larger files. As the buffer size is fixed and the system uses basic file chunking, transfers could be slow for large files, especially over high-latency networks.
- Lack of Security: The system lacks encryption or any security mechanisms like its SFTP [2] counterparts that operate with SSH, which poses a significant limitation for real-world applications. Sensitive data could be intercepted during transmission since all communication, including the control and data channels, happens in normal text. To be suitable for real-world use, the system would need to incorporate security protocols like TLS to protect the integrity and confidentiality of data.
- Limited Command Set: The system currently supports only a basic set of commands (e.g., ls, get, put, quit). While sufficient for a simple file transfer system, it is not as feature-rich as established protocols like FTP or SFTP, which offer additional commands for advanced file management. Expanding the command set would make the system more versatile and closer to being used in the real world.
- No Automatic Recovery from Failure: In the current implementation, if a transfer fails (e.g., due to network interruption), there is no built-in mechanism to automatically retry or resume the transfer upon detection of failure. This can lead to incomplete file transfers or the need for manual intervention, which reduces the reliability of the system in a real-world scenario.
- File Path Handling: The system assumes that files are located in the current directory, without any support for navigating file directories or managing paths. For a more versatile solution, features like relative/absolute path handling or file browsing capabilities would be necessary to make the system more flexible and user-friendly so that they don't have to place files in one singular folder for them to work with the client-server connection.

---

6. Conclusion and Future Work

6.1 Conclusion

In this project, a functional client-server file transfer system was developed using Python’s socket library. The system efficiently handles file transfers over TCP connections by

establishing separate control and ephemeral data channels, mimicking the behavior of established protocols like FTP. Key operations, such as listing files on the server, downloading, and uploading files, were implemented and successfully tested. The server can handle multiple clients at the same time using threads, and responses are provided after every command, ensuring a smooth and reliable interaction between the client and server. This implementation offers a simple but effective example of client-server communication in computer networks and can serve as the basis for learning how file transfer protocols operate in practice.

The project is significant to the field of computer networks as it highlights the importance of basic protocol design, network communication models, and error handling in systems. By emulating real-world behaviors like file listing and transfer using the control and data channel architecture, the project demonstrates foundational concepts of network communication and lays the groundwork for building more complex and secure systems.

6.2 Future Work

1. **Support for More Protocols**: Expanding the system to support other protocols, such as SFTP or FTPS [3], could enhance its versatility. These protocols add features such as encryption, authentication, and better error handling, making the system more secure and usable in real-world applications.
2. **Security Enhancements**: The system currently lacks security features like encryption, making it unsuitable for transferring sensitive data. Implementing TLS (Transport Layer Security) over the control and data channels would secure communication and protect the integrity and confidentiality of files during transfer.
3. **Error Handling and Fault Tolerance**: Enhancing error handling to support automatic retries and recovery from failures (e.g., network interruptions during file transfers) would make the system more reliable. Implementing mechanisms for resume transfers and detailed logging of the system services could also improve the system's reliability.
4. **File Path Handling**: Extending the system to support complex file paths (including directories) would make the client-server interaction more flexible. Allowing clients to specify full file paths or navigate directories would enhance usability and make the system more adaptable to real-world scenarios.

---

7. References

[1] "File Transfer Protocol (FTP)," RFC 959, Internet Engineering Task Force (IETF), Oct. 1985. [Online]. Available: https://www.rfc-editor.org/rfc/rfc959

[2] GeeksforGeeks, "SFTP | File Transfer Protocol," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/sftp-file-transfer-protocol/

[3] GeeksforGeeks, "Difference Between FTPS and SFTP," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/difference-between-ftps-and-sftp/

---

8. Appendices

Server-Side Code (serv.py)

This part contains the core functionality to handle client connections, process commands like ls, get, and put, and manage file transfers.

1. **handle_client function**
    a. Handles the communication with a connected client by processing commands sent from the client.

```
def handle_client(client_socket):
    """
    Handle communication with a connected client.
    This function runs in a separate thread for each client.
    """
    while True:
        # Receive a command from the client
        command = client_socket.recv(BUFFER_SIZE).decode()
        print(f"Received command: {command}")

        if command.startswith("ls"):
            list_files(client_socket)  # Call the function to list files
            confirmation = client_socket.recv(BUFFER_SIZE).decode()
            if confirmation == "FILES LISTED":
                print("Client confirms listing of files.")

        elif command.startswith("get"):
            filename = command.split()[1]
            handle_get(client_socket, filename)

        elif command.startswith("put"):
            filename = command.split()[1]
            handle_put(client_socket, filename)
```

```
            client_socket.send("UPLOAD SUCCESS".encode())

        elif command == "quit":
            print("Client disconnected.")
            client_socket.close()
            break
```

2. **handle_get function**
    a. Handles the process of sending a file from the server to the client over a data connection.

```
def handle_get(client_socket, filename):
    """
    Handle a request to download a file from the server to the client.
    """
    if os.path.exists(filename):
        client_socket.send("SUCCESS".encode())
        data_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        data_socket.bind(('', 0))  # Bind to an ephemeral port
        data_port = data_socket.getsockname()[1]
        client_socket.send(f"DATA_PORT {data_port}".encode())
        data_socket.listen(1)

        data_conn, addr = data_socket.accept()
        with open(filename, 'rb') as f:
            data = f.read(BUFFER_SIZE)
            while data:
                data_conn.send(data)
                data = f.read(BUFFER_SIZE)

        data_conn.close()
        print(f"File {filename} sent successfully.")
    else:
        client_socket.send("FAILURE: File not found".encode())
```

3. **handle_put function**
    a. Handles the process of receiving a file from the client and saving it on the server

```
def handle_put(client_socket, filename):
    """
    Handle file upload requests from the client.
    """
```

```
    data_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    data_socket.bind(('', 0))
    data_port = data_socket.getsockname()[1]
    client_socket.send(f"DATA_PORT {data_port}".encode())

    data_socket.listen(1)
    data_conn, addr = data_socket.accept()
    with open(filename, 'wb') as f:
        data = data_conn.recv(BUFFER_SIZE)
        while data:
            f.write(data)
            data = data_conn.recv(BUFFER_SIZE)

    data_conn.close()
    print(f"File {filename} uploaded successfully.")
```

4. **main function**
    a. Sets up the main server to listen for incoming client connections and starts a new thread for each client

```
def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(("0.0.0.0", 2121))
    server.listen(5)
    print("Server listening on port 2121")

    while True:
        client_socket, addr = server.accept()
        print(f"Accepted connection from {addr}")
        client_handler = threading.Thread(target=handle_client, args=(client_socket,))
        client_handler.start()
```

Client-Side Code (cli.py)

This part contains the logic for the client to send commands to the server, handle file downloads and uploads, and manage communication.

1. **main function**
    a. Handles user input for commands (ls, get, and put) and sends them to the server

```
def main():
    server_ip = "127.0.0.1"
    server_port = 2121
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
client.connect((server_ip, server_port))

while True:
    command = input("ftp> ")

    if command.startswith("get"):
        filename = command.split()[1]
        handle_get(client, filename)

    elif command.startswith("put"):
        filename = command.split()[1]
        handle_put(client, filename)

    elif command == "ls":
        client.send(command.encode())
        file_list = client.recv(BUFFER_SIZE).decode()
        print("Files on server:")
        print(file_list)
        client.send("FILES LISTED".encode())

    elif command == "quit":
        client.send(command.encode())
        client.close()
        break
```

2. **handle_get function**
    a. Handles downloading a file from the server over a data connection

```
def handle_get(client_socket, filename):
    client_socket.send(f"get {filename}".encode())
    response = client_socket.recv(BUFFER_SIZE).decode()

    if response == "SUCCESS":
        data_port_response = client_socket.recv(BUFFER_SIZE).decode()
        data_port = int(data_port_response.split()[1])

        data_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        data_socket.connect((client_socket.getpeername()[0], data_port))

        with open(filename, 'wb') as f:
            data = data_socket.recv(BUFFER_SIZE)
            while data:
```

```
            f.write(data)
            data = data_socket.recv(BUFFER_SIZE)

        data_socket.close()
        print(f"File {filename} downloaded successfully.")
        client_socket.send("FILE(S) RECEIVED".encode())
    else:
        client_socket.send("FILE(S) NOT RECEIVED".encode())
        print(response)
```

3. **handle_put function**
    a. Handles uploading a file to the server over a data connection

```
def handle_put(client_socket, filename):
    if os.path.exists(filename):
        client_socket.send(f"put {filename}".encode())
        data_port_response = client_socket.recv(BUFFER_SIZE).decode()
        data_port = int(data_port_response.split()[1])

        data_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        data_socket.connect((client_socket.getpeername()[0], data_port))

        with open(filename, 'rb') as f:
            data = f.read(BUFFER_SIZE)
            while data:
                data_socket.send(data)
                data = f.read(BUFFER_SIZE)

        data_socket.close()
        print(f"File {filename} uploaded successfully.")
        confirmation = client_socket.recv(BUFFER_SIZE).decode()
        if confirmation == "UPLOAD SUCCESS":
            print("Server confirms the file has been received.")
    else:
        print(f"FAILURE: File '{filename}' not found. No command sent to the server.")
```

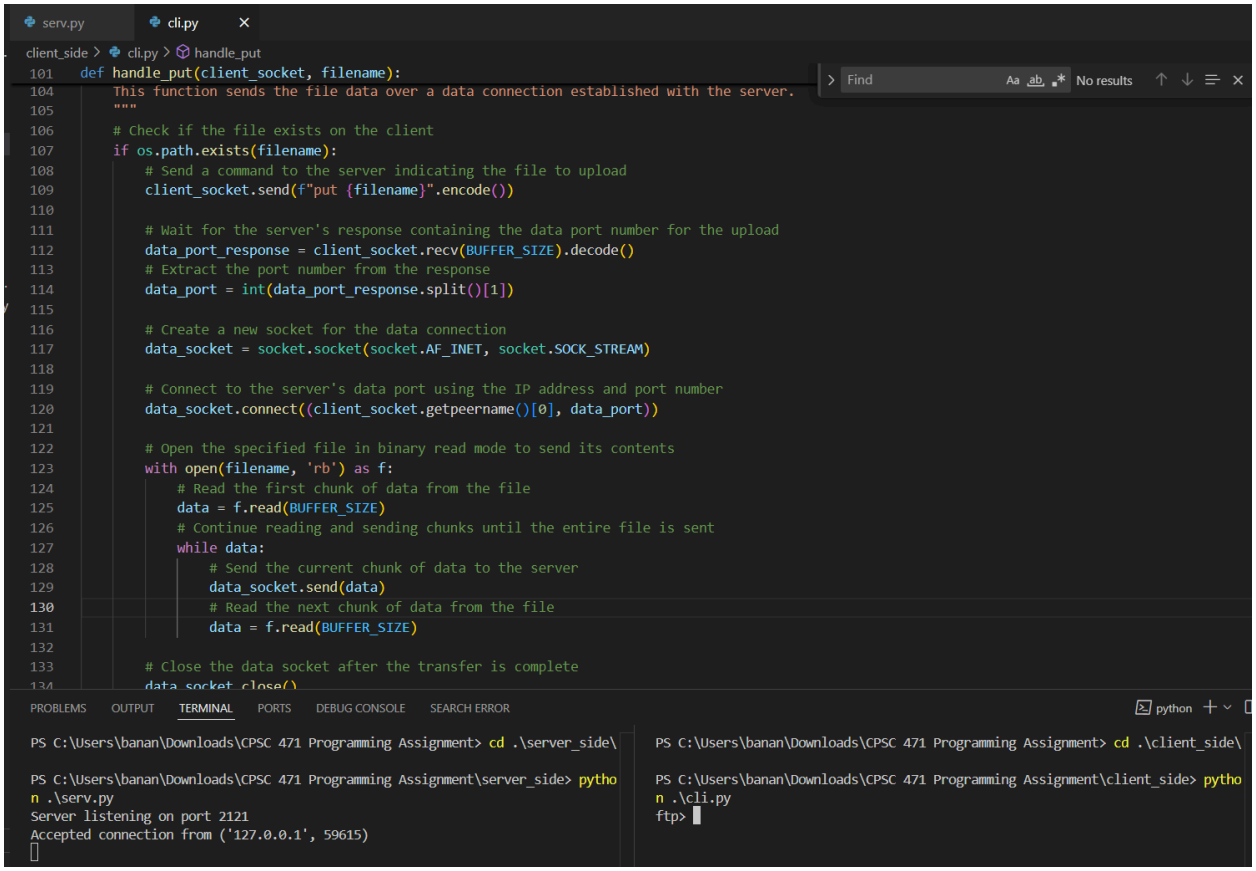
1. Network Configuration
    - Server IP Address:
        i. Default: 0.0.0.0 (binds to all available network interfaces)
        ii. Update server.bind() in serv.py if using a specific network interface.
    - Server Port:

i. Default: 2121
ii. Modify this port in serv.py as needed. Ensure the port is open and not blocked by a firewall.

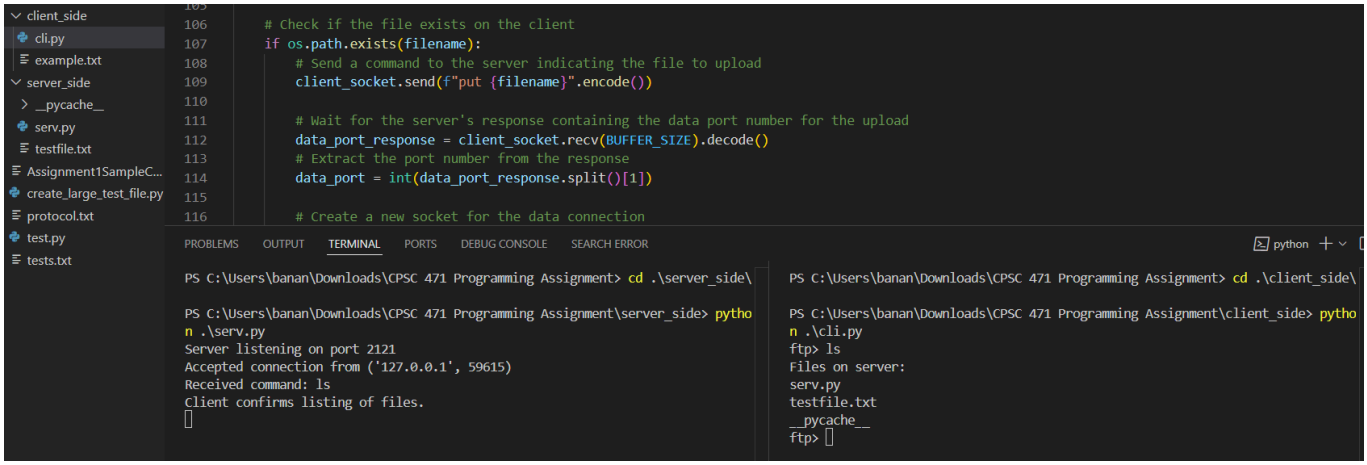
- Data Port:
  i. Dynamically assigned by the operating system (ephemeral port).
- Client IP Address:
  i. Default: 127.0.0.1 (localhost). This can be changed in cli.py to the server's IP address for remote communication.

2. Directory Setup
   - Server Working Directory:
     i. The server will operate in its current working directory.
     ii. Ensure the directory contains files you want to make available for download or is writable for uploaded files.
   - Client Working Directory:
     i. Files will be downloaded to the directory where the client script is executed.

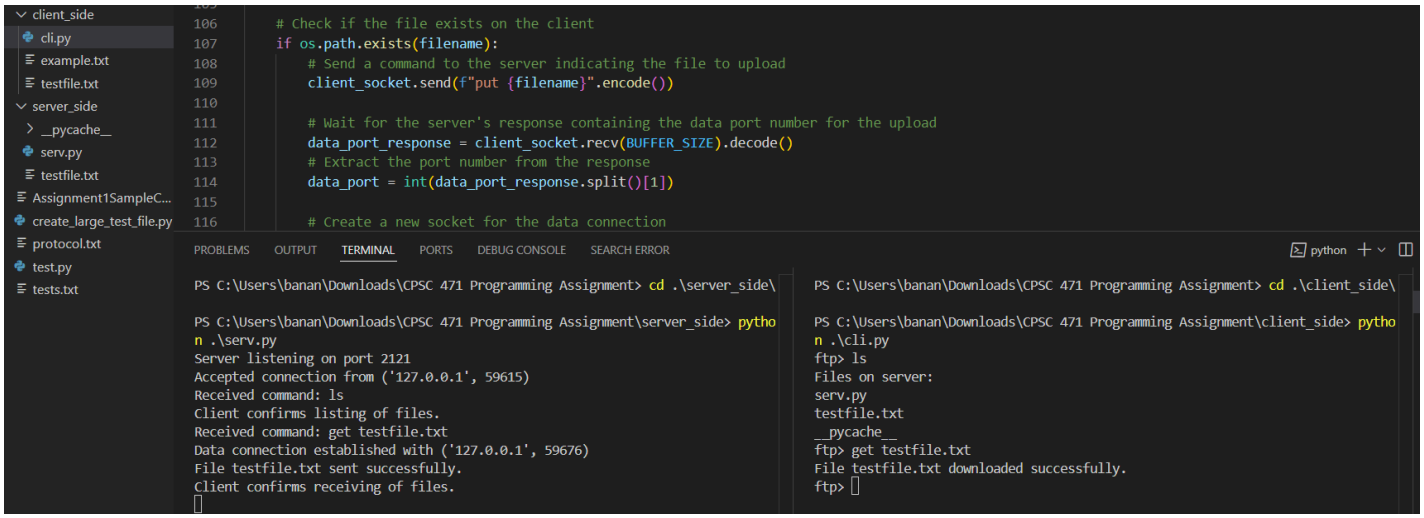
Start of Connection:



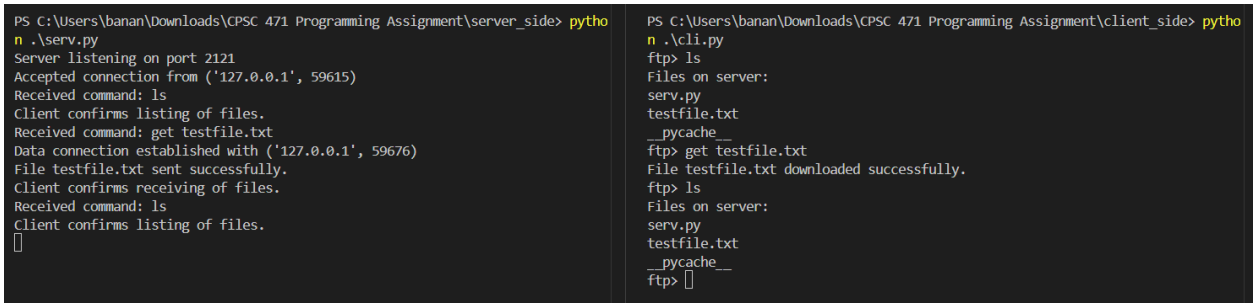
ls Command properly displays the server files in server_side directory:



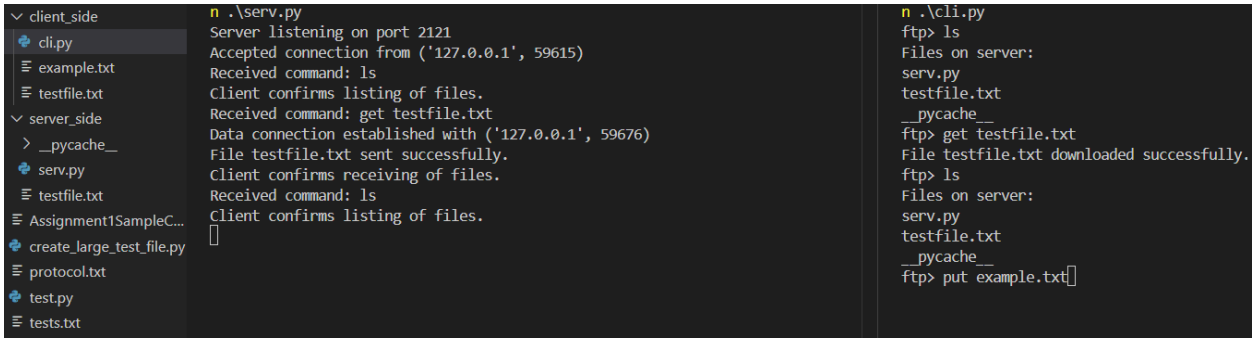
get Command retrieves testfile.txt from server_side and downloads it over data channel into client_side directory:

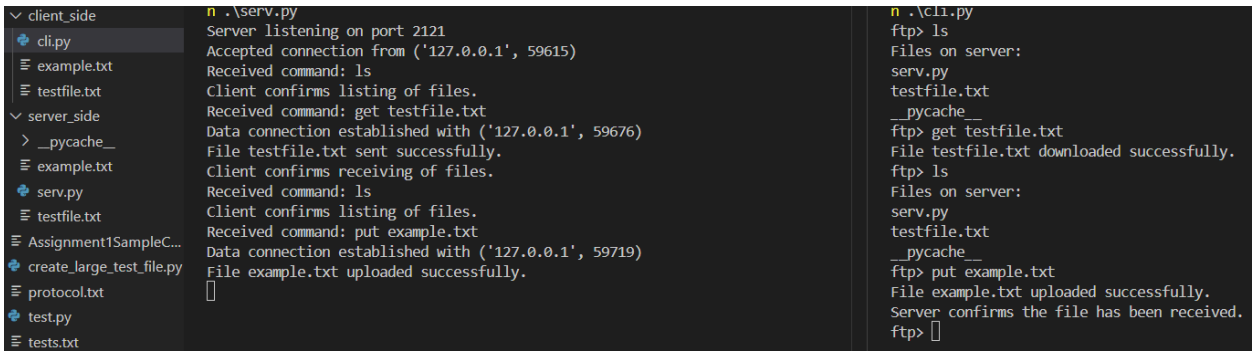


ls is run again to confirm that no files have been added to server_side:



put command takes example.txt in client_side and uploads it into server_side:





The quit command closes the control channel and ends communication between this client and the server: