FiTA: A File Transfer Application Based on STEP Protocol is All you need

MingXuan Hu 2252534 Rui Sang 2251576 ZiQi Liu 2253863

ZhiXin Li 2254577 ZhengHao Zhou 2257629

***Abstract*—This report presents the development of a client-server application for file uploading and downloading using Python Socket programming and the Simple Transfer and Exchange Protocol (STEP). The project focuses on debugging server-side code, implementing a client application, and ensuring reliable data transmission through TCP-based communication and JSON data representation. Key contributions include server code debugging, secure client login, and efficient file transfer mechanisms such as chunked transfer, multithreading, and MD5 checksum validation. Testing shows that multithreading improves performance for larger files, while single-threaded transfers are more efficient for smaller files. Future work may include adaptive thread management to further optimize efficiency.**

***Index Terms*—File Transfer, TCP, STEP Protocol, Python Socket Programming, C/S Architecture, Multithreading**

I. Introduction

A. Background

File uploading and downloading are fundamental web-based applications that play a vital role in our daily digital interactions. From backing up important data files to sharing documents and media files, these actions facilitate convenient communication and storage. The advent of cloud storage services and remote servers further magnifies the importance of efficient and reliable file transfer mechanisms [1]. Given this context, this project aims to develop a client application to explore the complexities of file uploading and downloading by using Python Socket programming, based on predefined protocols.

B. Challenge

The challenge of this project involves several key issues. Firstly, the server-side code provided contains some syntax errors that must be identified and corrected to ensure that the server operates correctly. Secondly, a client application must be implemented to interact with the server according to a specific protocol. In this project, the Simple Transfer and Exchange Protocol (STEP) is used, which is a TCP-based protocol that guarantees reliable data transmission and uses JSON as the data representation format. Addressing these challenges requires not only the ability to effectively debug Python code, but also a thorough understanding of server and client architectures. Additionally, careful attention to detail and strict adherence to protocols are essential to ensure successful communication and data transfer.

We sincerely thank all members of the CAN201 faculty team.

C. Practice Relevance

The implementation of this project has important practical significance. In the contemporary digital era, secure and efficient file transfer is critical for a variety of network applications, including cloud storage services, collaborative work environments, and data backup solutions [2]. By developing robust client applications for file uploading and downloading, the project helps to improve the reliability and performance of such services. In addition, the skills and knowledge gained from this project can be applied to other web-based applications, thereby improving the overall efficiency of data communication and management.

D. Contribution

In this coursework, several key contributions have been made. Firstly, the server-side code is carefully debugged to ensure it runs error-free, displaying "Server is ready" upon successful execution. Secondly, a client application is developed to interact with the server according to the STEP protocol, which includes a login process that secures a token for further interactions and ensures reliable data transmission. Thirdly, the client application efficiently uploads files to the server by first requesting permission and receiving an uploading plan, then transferring the file block by block. Finally, the client verifies the file's status on the server, checking the MD5 hash to ensure proper reception. These contributions demonstrate a comprehensive understanding of network programming and the ability to address practical challenges in file transfer applications.

II. Relate Work

In the design of efficient and secure file transfer applications, the research on network traffic redirection has made an important contribution to its development. Network traffic redirection involves rerouting data packets through alternative paths to optimize performance, enhance security, and balance server loads. This technique ensures that file transfers are conducted smoothly, securely, and efficiently by preventing congestion, mitigating potential threats, and improving overall data transmission speed.

Many researchers have delved into various aspects of network traffic redirection, contributing valuable insights and advancements. For instance, Kumar and Mishra explored load balancing techniques to distribute file transfer requests across

multiple servers, preventing bottlenecks and enhancing performance [3]. Subashini and Kavitha focused on the security implications, demonstrating how traffic redirection can route data through security devices to filter out malicious content [4]. Additionally, Liao et al. investigated optimization strategies, showing that intelligent traffic redirection can reduce latency and improve transmission speeds by avoiding congested network paths [5]. These studies collectively highlight the multifaceted benefits of network traffic redirection in enhancing file transfer applications.

III. Design

A. C/S Network Architecture

This project employs a **TCP**-based client-server (C/S) architecture for file uploads and downloads using the **STEP** protocol. The server manages data and supports multiple concurrent client connections, allowing simultaneous uploads and downloads. As shown in Figure 1, the server listens on port **1379**, and clients initiate requests over TCP, ensuring reliable data transfer.

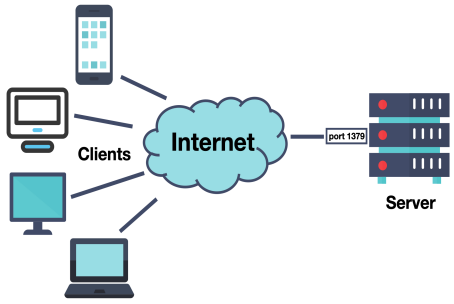

Fig. 1. Client and Server Architecture

B. Workflow of Program

Figure 2 shows a sequence diagram illustrating the STEP protocol-based interaction between the client and server.

1) **Authorization:** The client initiates a login request with user ID and encrypted password. The server validates and returns a token, which serves as a credential for subsequent requests.
2) **File Upload:** For file uploads, the client first sends an upload request. The server responds with an upload plan, including key, block_size, and total_block. The client then uploads the file in chunks using multithreading, verifying the integrity upon completion with an MD5 check.
3) **Additional Functions:** Delete, Save, and Download functionalities are also implemented. Only the Save sequence is shown here; Delete is similar to Save, and Download resembles Upload. Details of these functions will be provided in the implementation section.

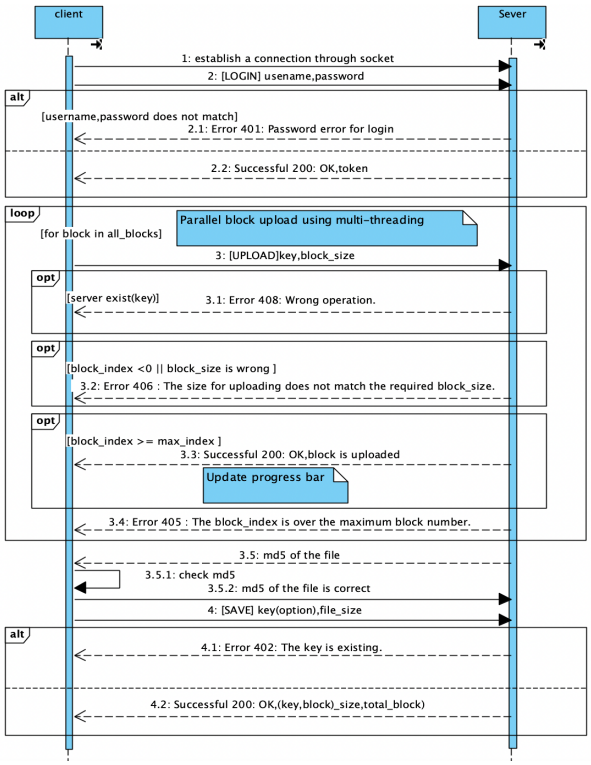

Fig. 2. Sequence Diagram of Program

C. Key Functional Design

1) **Efficient File Transfer Control**
   a) **Chunked Transfer and Validation:** Files are transferred in chunks, with each chunk verified by the client using the server's MD5 checksum. This reduces retransmission in case of errors, enhancing efficiency.
   b) **Dynamic Block Size:** The client adjusts to the server's block_size during upload, while the server calculates optimal block_size based on file size.
   c) **Multithreaded Upload:** A thread pool uploads chunks in parallel, with `thread.join()` ensuring synchronization for data integrity and stability.
   d) **File Format Adaptation:** The client adjusts processing based on file format (e.g., PDF, JPG) to ensure efficient uploads.
2) **User-Friendly Debugging and Feedback**
   a) **Transfer Progress Display:** Real-time progress display during file transfer keeps users informed, improving experience.

D. Algorithm

The two pseudo codes respectively represent the algorithms to achieve the login function and upload function in the program.

Algorithm 1 Login Algorithm

Input: `sock`: Socket object for communication `username`: Username for login
Output: `Token`: Authentication token received from server upon successful login, or

```
Try:                                                          ▷ Start of try block
FIELD_OPERATION: OP_LOGIN
FIELD_USERNAME: username
FIELD_PASSWORD: md5_hash(username)          ▷ MD5 hash of username used as password
sock.sendall(make_packet(login_data))
print("Login request sent.")
response_json, _ ← receive_packet(sock)
if FIELD_TOKEN exists in response_json then
    token ← response_json[FIELD_TOKEN]
    return token
else
    status_message ← response_json.get(FIELD_STATUS_MSG, "Unknown error")
    return None
end if
Catch: Exception as e                                  ▷ Catch block for exceptions
return None
```

Algorithm 2 Upload File to Server

Input: `sock`: Socket object for communication
`token`: Authentication token for server access
`file_path`: Path to the file being uploaded
Output: Coordinates the upload of a file to the server in blocks, with a completion message upon success.

```
Try:                                                          ▷ Start of try block
key ← os.path.getsize(file_path)
sock.sendall(make_packet(upload_request))          ▷ Send upload plan request to server
response_json, _ ← receive_packet(sock)            ▷ Receive upload plan response from server
If response_json[FIELD_STATUS] != 200
  print("Failed to get upload plan")
  Return
key ← response_json[FIELD_KEY]
block_size ← response_json.get(FIELD_BLOCK_SIZE, MAX_PACKET_SIZE)
total_blocks ← response_json.get(FIELD_TOTAL_BLOCK, 1)
print("Upload plan received: key={key}, block_size={block_size},
total_blocks={total_blocks}")
for each block_index from 0 to total_blocks - 1 do     ▷ Initialize empty list threads for parallel upload
    upload_block(sock, token, file_path, key, block_index, block_size)
    Add thread to threads list
end for
for each thread in threads do
    thread.join()
end for
print("File upload completed.")
Catch Exception as e                                   ▷ Catch block for exceptions
print("An error occurred during file upload: " + str(e))
```

IV. Implementation

A. The Host Environment

The *json* library encodes and decodes transmitted data, while *hashlib* performs verification tasks, specifically by computing the MD5 checksum for files and accounts. The *os* library facilitates operations related to the file system, such as retrieving file sizes. The *threading* library improves the efficiency of uploading and downloading file chunks by handling them in multiple threads. Meanwhile, *tkinter* implements the graphical user interface, providing an interactive window for managing server connections and file operations.

TABLE I
DEVELOPMENT ENVIRONMENT

Field	Description
Operating System	Microsoft Windows 11
CPU	12th Gen Intel(R) Core(TM) i5-12500H
GPU	NVIDIA GeForce RTX 3050 Ti Laptop GPU
RAM	16G
IDE	JetBrains PyCharm Community 2023.3
Python Version	Python 3.8
Python Library	'socket', 'json', 'hashlib', 'struct', 'os', 'argparse', 'threading', 'time', 'logging', 'math', 'shutil', 'tkinter'

B. Steps of Implementation

1) Analyze the requirements document to clarify the core functions and goals of the system.
2) Debug and fix errors in the server code before writing the client code to ensure it handles requests correctly.
3) Set up the development environment, including using JetBrains PyCharm as the IDE, configuring Python 3.8, and installing the required Python libraries.
4) Design the client architecture, focusing on module structure and interaction logic.
5) Implement command-line parsing, authentication, and file upload/download modules to ensure the client can interact correctly with the server and handle responses.
6) To improve the efficiency of file uploads and downloads, add multi-threading to handle file blocks in parallel.
7) Design and implement a user interface to facilitate interaction.
8) Add exception handling mechanisms to ensure that the client can handle issues properly in case of network interruptions or server response errors, and provide appropriate user prompts.
9) Deploy the client and server on two virtual machines for testing to verify system stability and complete the report.

C. Flow Chart

The flow chart and the entire structure is shown in Figure 3 and Figure 4.

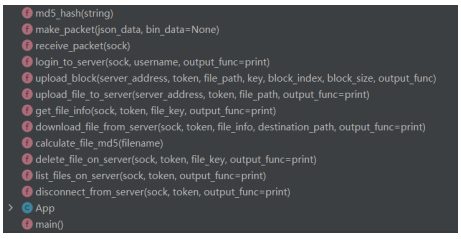

Fig. 3. Client's Methods

D. Programming Skills

1) **Object-Oriented Programming (OOP):** Used OOP principles to design the sever and client, making the code more modular, reusable, and easier to maintain. Different functionalities, such as authentication, file transfer, and error handling, were encapsulated in independent classes.
2) **Modular Design:** Applied modular design by breaking down the client into smaller, self-contained components, each responsible for a specific functionality. This approach improved code readability and allowed easier testing and debugging.
3) **Multi-threading:** Implemented multi-threading to enhance the efficiency of file transfers by uploading or downloading file blocks concurrently. This reduced wait times and improved overall system performance.
4) **JSON Handling:** Utilized the *json* library for encoding and decoding data packets, ensuring a standardized format for communication between the client and server. This made the data exchange process more reliable and easier to debug.

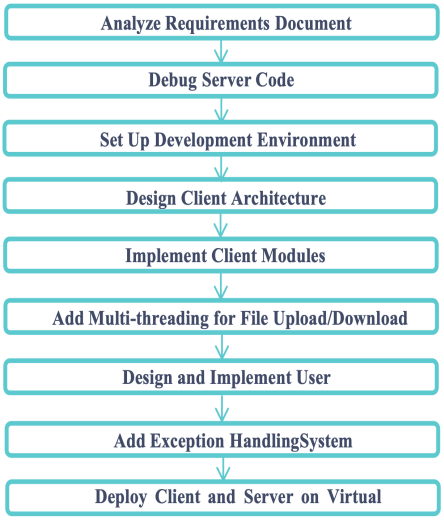

Fig. 4. Flowchart of Implementation

E. Actual Implementation

1) **Authorization Function:** The *login_to_server* function establishes a TCP connection, sends login requests, and calculates an MD5 hash of the username as a password. A login packet is created using the *make_packet* function and sent to the server. The client then extracts the authorization token from the server's response for future operations.
2) **File Uploading Function:** The *upload_file_to_server* function follows these steps: (a) The client requests an upload plan from the server. (b) The client receives the server's response to verify if the upload plan request was successful and obtains a unique key for the file from the server, which will be used to identify the file on the server. (c) The file is split into blocks on the client side. (d) Each block is read and an upload packet is created using *make_packet*. (e) The *upload_block*

function uploads each block, either sequentially or in parallel using multiple threads. (f) After all blocks are uploaded, the MD5 hash of the complete file is verified to ensure data integrity. (g) Check success/failure, and output the corresponding message on both client and server side.

F. Difficulties and Solutions

In the process of multi-threaded uploading, we encountered a specific challenge: the client requires multiple connections, leading to several simultaneous connections on the server attempting to write to the same log file. This concurrent write operation from multiple threads to a single file can cause read-write conflicts, resulting in the loss of certain data blocks and, ultimately, upload failure.

To address this common issue, we implemented a thread-lock mechanism on the server. By applying a thread lock around the log write operations, we ensure that only one thread can access the log file for writing at any given time. This approach effectively prevents read-write conflicts, preserving the integrity and consistency of the log file. As a result, it eliminates data block loss due to log write conflicts, thereby improving both the success rate of file uploads and the overall stability of the system.

V. Test and Result

A. Test Environment

TABLE II
Test Environment 1

Field	Description
Operating System	Microsoft Windows 11
CPU	12th Gen Intel(R) Core(TM) i5-12400F
GPU	NVIDIA GeForce RTX 2060 SUPER
RAM	32G
IDE	Visual Studio Code 2023.3
Python Version	Python 3.11

TABLE III
Test Environment 2

Field	Description
Operating System	Microsoft Windows 11
CPU	12th Gen Intel(R) Core(TM) i5-12500H
GPU	NVIDIA GeForce RTX 3050 Ti Laptop GPU
RAM	16G
IDE	JetBrains PyCharm Community 2023.3
Python Version	Python 3.8

B. Test Steps

1) **Server Debugging:** Enter "python server.py" in the terminal. Once the server is ready, the message "Server is ready" is displayed in the terminal, as shown in Figure 5.
2) **Authorization:** Enter "python client.py –server_ip 127.0.0.1 –id 1379 –f test.txt" in the terminal. Enter the user's name in the UI window, then press the "Connect and Login" button. The server will send a token to the client, which is displayed in the terminal upon successful receipt, as shown in Figure 6.

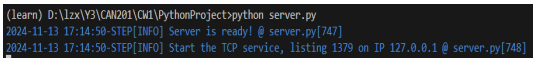

Fig. 5. Server is Ready

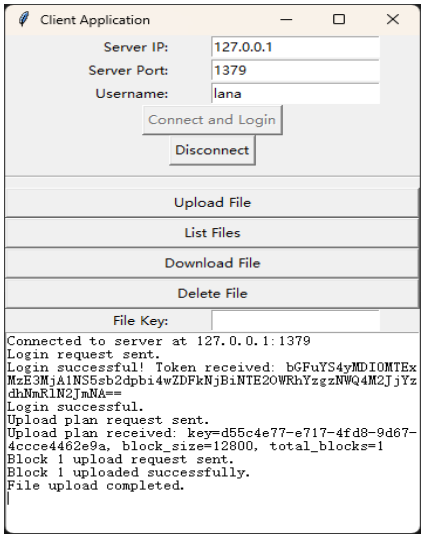

Fig. 6. Client Running Result

3) **File Uploading:**
   a) Press the "Upload File" button and select the file you wish to upload.
   b) The client sends an UPLOAD plan request and receives a response from the server.
   c) The client automatically proceeds with the UPLOAD request, sending each block of the file sequentially.
   d) Simultaneously, the client monitors the server's responses to ensure each block is successfully sent. The client terminal upon successful file upload is shown in Figure 6.
   e) After all blocks are transferred, the file's MD5 hash is verified to ensure data integrity, and a success or failure message is communicated to both server and client, as shown in the server terminal in Figure 7.

Fig. 7. Server Running Result

4) **File Listing:** Press the "List Files" button. The client sends a GET request, and the server responds with a list of all files the user has uploaded.

5) **File Downloading:**
   a) Enter the file key in the UI window and press the "Download File" button.
   b) The client sends a GET request and receives a response from the server.
   c) The server transmits each block of the file to the client.
   d) Upon completion, the MD5 hash of the file is verified, and a success or failure message is sent to both server and client.
6) **File Deletion:** Enter the file key in the UI window and press the "Delete File" button. The client sends a DELETE request, and the server removes the file log, sending a response indicating successful or unsuccessful deletion.

C. Test Results

We tested ten files of varying sizes (1KB-40MB), each transferred ten times, and recorded the average transfer time. As illustrated in Figure 8, file size is linearly correlated with transfer time. When uploading larger files, the transfer time increases accordingly. Additionally, hardware and network connection protocols can also impact transfer rates.

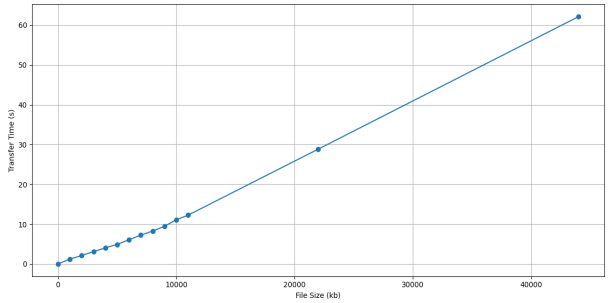

Fig. 8. Transfer Time for Different Sizes of Files

To enhance the efficiency of large file transfers, we incorporated multithreading to transmit the blocks. Multithreading divides the file into chunks, with each thread responsible for transmitting a portion of these chunks, enabling concurrent processing of multiple data blocks. The transfer rates of different file sizes with multithreading are shown in Figure 9.

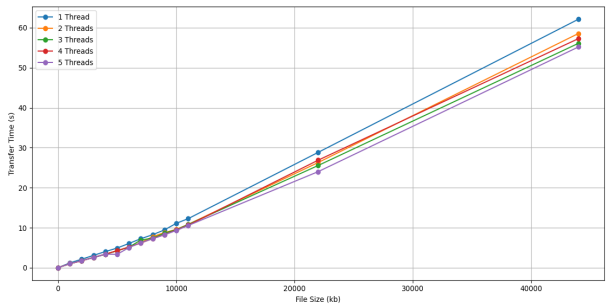

Fig. 9. Transfer Time for Different Sizes of Files with Multithreading

Our experiments demonstrated that this approach improves overall transfer speed to a certain extent, with significant benefits for larger files (over 10MB). When files are distributed across multiple threads, the client can complete transfers more quickly. Compared to single-threaded sequential transmission, multithreading allows a greater volume of data transfer within the same timeframe.

However, the benefits of multithreading are influenced by file size. For smaller files, the overhead associated with thread creation, management, and synchronization can lead to resource contention, diminishing performance improvements and sometimes even increasing latency. Additionally, network fluctuations may affect transfer speed, as multithreading relies on consistent network conditions for optimal performance. Based on these results, we conclude that single-threading is more cost-effective for files smaller than 10MB, while multithreading offers substantial advantages for larger files. Consequently, an adaptive thread management mechanism could be implemented in the client program, dynamically adjusting the number of threads based on file size and network conditions to optimize transfer efficiency.

VI. Conclusion

This project successfully implemented a reliable file transfer system using the STEP protocol, incorporating key features such as multithreading, chunked uploads, and MD5 verification to ensure data integrity and efficient transfer. Through extensive testing, we observed that multithreading significantly improved performance for larger files, while single-threaded transfer was more efficient for smaller files. This highlights the importance of selecting the appropriate transfer method based on file size to optimize performance.

Looking ahead, there are several avenues for further enhancement. One promising direction is the development of an adaptive thread management mechanism. Such a system could dynamically adjust the number of threads based on file size and current network conditions, thereby optimizing transfer efficiency in real-time. By continuing to refine and expand upon the foundational work presented in this project, we can develop a more versatile and resilient file transfer system that meets the evolving needs of modern digital communication.

References

[1] Zahid Ghaffar et al. "A lightweight and efficient remote data authentication protocol over cloud storage environment". In: *IEEE Transactions on Network Science and Engineering* 10.1 (2022), pp. 103–112.
[2] Jun-Wen Chan, Swee-Huay Heng, and Syh-Yuan Tan. "Secure data sharing in a cyber-physical cloud environment". In: *Journal of Telecommunications and the Digital Economy* 11.4 (2023), pp. 94–112.
[3] Nitin Kumar Mishra and Nishchol Mishra. "Load balancing techniques: need, objectives and major challenges in cloud computing-a systematic review". In: *International Journal of Computer Applications* 131.18 (2015), pp. 0975–8887.
[4] Subashini Subashini and Veeraruna Kavitha. "A survey on security issues in service delivery models of cloud computing". In: *Journal of network and computer applications* 34.1 (2011), pp. 1–11.
[5] Daqiang Liao, Du Zou, and Gautam Srivastava. "Improved design of load balancing for multipath routing protocol". In: *Journal of High Speed Networks* 29.4 (2023), pp. 265–278.