Design and Implementation of a File Transfer System Using Python Socket Programming

Chenrui Zhu 2361387 **Qi Chen** 2363203 **Shengyang Yin** 2361841 **Tianchen Niu** 2361034 **Youming Yang** 2361217

***Abstract*—This report presents the design and implementation of a client-server file transfer system based on Python, which follows a custom STEP protocol. This system is equipped with user authentication function, supports the upload of block files with MD5 verification, and can improve performance through a multi-threading mechanism. A series of functional tests and performance tests have verified that this implementation scheme can support data transmission of different file types and has a stable upload speed. This study demonstrates the practical application of concepts related to socket programming and protocol design.**

I. Introduction

A. *Task Specification & Practice relevance*

This project mainly achieved three goals. The first one is to debug and correct the syntax error of the provided server code. The second task focuses on implementing client login, including user identity verification and acquisition authorization token. The ultimate goal is to develop one stable file upload function that enables tcp to split files into blocks for transmission, and MD5 verifies their integrity upon receipt.

The STEP protocol implemented in our project has potential applications across many fields. For example, Jagadeeswari *et al.* [1] indicate that organizations which have a high priority on data security and privacy require secure file-sharing technologies, and the token-based authentication we design can encrypt uploaded files.

B. *Challenges & Contributions*

During the development process, we discovered a series of difficulties, mainly as follows. The first is to understand and be familiar with the provided code. It is necessary to understand the project requirements and identify all bugs. Secondly, how to achieve user login and token return functions is a challenge. Finally, it is necessary but insufficient to implement the core function of uploading and verify it with MD5. On this basis, whether to expand additional functions such as deletion, parallelism, multi-threading, etc. is essential to consider.

This study implements a practical system of the STEP protocol. This method includes handling basic syntax errors, multi-threaded block file transfer functions, and its functionality and reliability have been verified through tests.

II. Related Work

This project builds on the socket-based TCP client–server model of Maata *et al.* [2], which provides the basic bidirectional communication structure. Building on this foundation, we adopt the concurrent-TCP techniques explored by Hiraoka [3], where a file is segmented into blocks and transmitted through multiple independent TCP streams. Such parallelism effectively redirects network traffic by distributing a single logical transfer across several flows, and prior work shows that single-connection approaches cannot replicate the robustness of true parallel TCP [3]. In addition, studies of transmission behaviour indicate that when packet loss occurs, the source must re-establish a route using smooth-start mechanisms [4], which also constitutes a form of traffic redirection.

III. Design

A. *System Architecture*

In this project, we utilize a classic client-server architecture, where the server keeps listening for TCP connections, and the client creates connections when it needs to log in, authenticate, upload, or delete files (Figure 1). On the client side, STEP messages are constructed according to the protocol specification and the target file is cut into fixed-size blocks that are sent in parallel by several threads (four threads by default). The server receives and verifies each message from the client, executes the requested file operation, and stores the data in the specific directory. All control information and file blocks are transmitted over TCP using customised STEP message format.

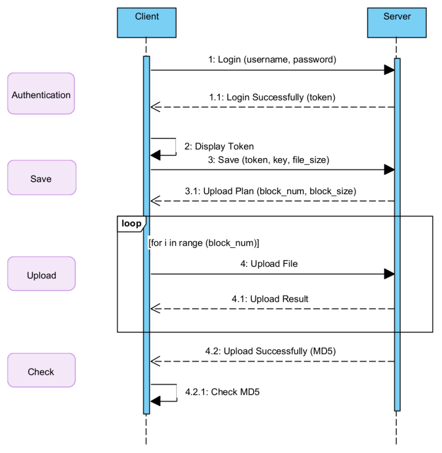

Client
Server
Authentication
1: Login (username, password)
1.1: Login Successfully (token)
2: Display Token
3: Save (token, key, file_size)
Save
3.1: Upload Plan (block_num, block_size)
loop
[for i in range (block_num)]
4: Upload File
Upload
4.1: Upload Result
4.2: Upload Successfully (MD5)
Check
4.2.1: Check MD5

Fig. 1: Client-Server Communication

B. *Protocol Design*

In this project, we implement STEP protocol, which defines how messages are constructed. Each request or response message follows the same structure as below.

1) Packet Structure

Each STEP packet contains three components:

- **Header (8 bytes)**: the first 4 bytes specify the length of the JSON section, and the next 4 bytes specify the length of the binary section, respectively.
- **JSON Section**: contains all control information, including operation type, direction (REQUEST or RESPONSE), username, token, file key, block index, and block size.
- **Binary Section**: an optional payload for file block transmissions during upload and download process.

On the server side, the received data is assembled into an integrated packet by the `get_tcp_packet()` routine, which extracts JSON metadata and executes corresponding operations.

2) Operation Types

In our implementation, we mainly use four STEP operations on the client side: LOGIN, SAVE, UPLOAD, and DELETE. However, the server also supports GET, DOWNLOAD, and BYE as defined in the STEP protocol. All operations except LOGIN require the client to present a token. The server validates this token by decoding it and verifying its MD5 hash.

C. Workflow

1) Authentication Process

Authentication is required before any file operations is been held. The client computes the MD5 hash of the username and sends a LOGIN request with the username and hash. The server validates tge MD5 by a repeat computation. If it is valid, the server generates a token by hashing the username with a random factor, returns it to the client, and requires it for all subsequent requests.

2) File Upload Process

After authentication, the client uploads files in parallel blocks. It sends a SAVE request with the total file size, and the server allocates a temporary file, creates a progress log, and returns the file key, block size(set to the protocol maximum), total blocks, and token. The client splits the file into blocks, with each thread establishing its own TCP connection to send an UPLOAD request with a block index and data, retrying on failure. The server writes each block at the correct offset, updates the progress log, and after all blocks are received, computes the MD5 of the full file, deletes the log, and moves the file to the permanent directory. The client may verify the MD5 to confirm file integrity.

D. Algorithms

The system uses defined algorithms for authentication and file upload, as illustrated in the following pseudocode.

1) Authentication Algorithm

Algorithm 1 Authentication Process

```
Require: Valid username
Ensure: Authentication token for subsequent operations
1: Compute $passwordHash \leftarrow$ MD5($username$)
2: Construct $request \leftarrow$ $operation$ : "LOGIN", $username, passwordHash$
3: $response \leftarrow$ SendRequest($request$)
4: if response.status = SUCCESS then
5:     return $response.token$
6: else
7:     return null
8: end if
```

2) File Upload Algorithm

Algorithm 2 File Upload Process

```
Require: Valid file path and authentication token
Ensure: Successful file transfer with integrity verification
1: $fileSize \leftarrow$ GetFileSize($filePath$)
2: $uploadPlan \leftarrow$ RequestUploadPlan($fileSize, token$)
3: $(blockSize, totalBlocks, fileKey) \leftarrow (plan.blockSize, plan.totalBlocks, plan.key)$
4: for $blockIndex = 0$ to $totalBlocks - 1$ do
5:     $blockData \leftarrow$ ReadBlock($filePath, blockIndex, blockSize$)
6:     UploadBlock($fileKey, blockIndex, blockData, token$)
7: end for
8: $serverMD5 \leftarrow$ VerifyUploadCompletion($fileKey$)
9: $localMD5 \leftarrow$ ComputeFileMD5($filePath$)
10: if $serverMD5 = localMD5$ then
11:     return SUCCESS
12: else
13:     return FAILURE
14: end if
```

IV. Implementation

A. Development Environment

We completed development and testing on both Windows and macOS, ensuring better cross-platform compatibility. A summary of our development devices is provided in Table I.

TABLE I: DEV ENVIRONMENT

Specification	Windows	macOS
OS	Windows 11	macOS Sequoia 15.6
CPU	Intel(R)Core(TM) i9-14900HX	Apple M3
RAM	32GB	8GB @ 6400MHz
IDE	PyCharm 2025.1.3	PyCharm 2024.3.3

We used Git (v2.50.1) to synchronize and modify code.

Python libraries: `socket`, `json`, `struct`, `hashlib`, `threading`, `tqdm`, `logging`.

B. Steps Implementation

1) Server-side Debugging and Bug Fixing

During the debugging phase, several issues in the **original** `server.py` were identified and corrected. The main fixes are summarized as follows:

- **File Handling and Encoding**:
  - Lines 34–40: Fixed file MD5 calculation by opening files in binary mode (`'rb'`) instead of text mode (`'r'`), ensuring correct hashing for non-text data.
  - Lines 443–449: Simplified file log reading using a context manager and ensured line stripping to prevent duplication issues during block completion detection.
- **Logging Consistency**:
  - Throughout Lines 224–371: Replaced all incorrect `logger.error()` calls (used for normal operations) with `logger.info()`, aligning log levels with actual message semantics (e.g., successful save/delete operations).
- **Constant Name and Operation Code Errors**:
  - Line 212: Corrected `op_save` to `OP_SAVE`, fixing a reference inconsistency that caused runtime errors.
  - Lines 351–495: Corrected operation codes in function `make_response_packet()` calls ensuring proper protocol handling for delete / download.
- **Structural and Syntax Fixes**:
  - Line 594: Fixed an incorrect password field reference `json_data['password']` → `json_data[FIELD_PASSWORD]` to prevent authentication failures.
  - Line 651: Corrected socket type from UDP to TCP: `socket(AF_INET, SOCK_DGRAM)` → `socket(AF_INET, SOCK_STREAM)` and added missing `th.start()` to enable threaded connections.
- **Minor Logic and Typo Corrections**:
  - Fixed missing `os.makedirs()` for the temporary upload folder (`tmp/username`) before file creation.
  - Line 686: Fixed function call typo `tcp-listener` → `tcp_listener`.

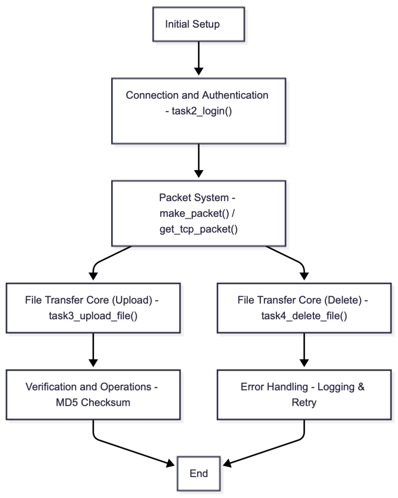
2) Client-side Implementation

The client-side implementation was completed through the following main steps:

a. **Initialization**: Import necessary libraries (such as `socket`, `threading`, etc.) and define basic constants like server address, port, and retry limit.

b. **Command-Line Parsing**: Manually parse command-line parameters using `sys.argv`, supporting both parameterized commands and interactive input modes.

c. **Authentication**: We use the `task2_login()` function to generate MD5-based credentials, send them via `make_packet()`, and parse the server's response using `get_tcp_packet()` to obtain the authentication token.



Initial Setup
Connection and Authentication - task2_login()
Packet System - make_packet() / get_tcp_packet()
File Transfer Core (Upload) - task3_upload_file()
File Transfer Core (Delete) - task4_delete_file()
Verification and Operations - MD5 Checksum
Error Handling - Logging & Retry
End

Fig. 2: Flowchart of implementation

d. **Packet System**: The `make_packet()` and `get_tcp_packet()` functions are reused to handle the packaging and parsing of TCP data.

e. **File Upload**: The `task3_upload_file()` function adopts a three-step upload method:
- First, request an upload plan to get the block size and total number of blocks;
- Second, use `threading.Lock` for safety, read, split, and upload blocks concurrently, with a retry mechanism in place;
- Finally, the client verifies the MD5 and records data such as time consumption.

f. **File Deletion**: `task4_delete_file()` sent structured delete packets and checked responses.

g. **Interactive Mode**: The `interactive_menu()` provided text-based options (upload, delete) for flexible operation without command-line arguments.

h. **Integration**: The `main()` function coordinated all modules, managed login, mode selection, and ensured reliable execution with robust error handling.

C. Programming Techniques

- **Object-Oriented Programming (OOP)**: The program follows OOP principles by organizing the code into well-defined functions and modules, resulting in a clear structure and strong extensibility.
- **Modularity and Code Reusability**: Core functionalities such as login, file upload, and file deletion are encapsulated into independent functions, while existing server-side utility functions are reused (e.g., `get_file_md5()` imported from `server.py`), making maintenance and feature extension more efficient.
- **Multithreading**: The project introduces a multithreading mechanism (e.g., using `threading.Thread`), which enables parallel file block uploads and significantly improves data transfer efficiency.

D. Actual Implementation

1) Authorization Function

This function handles user authentication and obtains the token required for subsequent operations. A TCP client socket is created and connected to the server, and the user-provided username is hashed using MD5 to generate the password. The authentication fields are then packaged into a login packet using `make_packet()` and sent via `sendall(login_packet)`. If the server validates the credentials and returns status 200, the client extracts the token for later requests.

2) File Upload Function

The client first retrieves the file size and path, then sends a save request containing this information along with the previously obtained authentication token. The server response provides the upload plan, including `block_size` and `total_block`.

The file is then read in binary mode(`with open(file_path, 'rb')` and split into blocks according to the specified size. Each block is uploaded parallelly using a request that includes the block index, file key, and token. The client waits for the server response after each block, and a retry mechanism handles any failures.

After all blocks are uploaded, the server returns the MD5(`server_md5`) of the complete file. This is compared with the local MD5 to verify the upload: if they match, the upload is considered successful.

3) File Deletion Function

To satisfy the function`task4_delete_file()`, client first send a deletion request containing the token and file key to the server, and then process the server's response. If the response indicates that the operation is successful, we confirm that the deletion is completed; otherwise, we log the error information.

4) Upload Retry & Multithreaded Support

The upload function includes a retry mechanism that resends any block that fails to upload, ensuring reliable transfer. By Default, multiple blocks can be uploaded in parallel using multithreading, improving efficiency while maintaining data integrity.

E. Difficulties

1) Code Understanding

In order to understand how the code works and Overall structure, we faced a few challenges. The The main task was to understand how TCP transmits data. principles, with a focus on binary packing and unpacking. data and JSON data. We had to explain when functions are used, like `make_packet()` and `get_tcp_packet()` to ensure we wouldn't make mistakes in subsequent coding.

2) Version Control with Git

We faced some difficulties during the collaboration. such as unfamiliarity with Git and disagreements about implementa tion mentalion plans. We did address all these issues through Git usage. guidance, learning, and holding regular meetings.

3) Multiprocessing Implementation

During the development process, we found retransmitting a single file block may block the whole process of uploading. In solving this problem, we introduced multithreading to allow concurrent uploads. Without using the queue module, we used shared list with `threading.Lock` to achieve thread-safe task coordination.

V. Testing and results

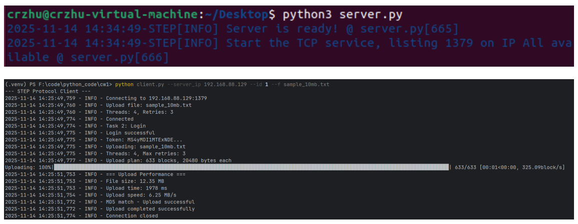
A. Testing environment

	Client	Server (VM)
CPU	Intel(R) Core(TM) i9-14900HX	4 vCPUs
RAM	32GB 3200MHz	4GB
OS	Windows 11 Pro	Ubuntu 22.04 LTS
IDE	PyCharm 2024.2	Terminal
Python Version	Python 3.10.12	Python 3.10.12
Connection	NAT	NAT

TABLE II: Test environment configuration

B. Local-to-Virtual Machine File Transfer Testing

To evaluate the functionality and stability of the file transfer system, we conducted an end-to-end test for task 1 to 3. In the experiment, the server was running on the Ubuntu 22.04 virtual machine and the client was running on the local host. The client uploaded a 10MB text file, and the server successfully received it in its entirety and verified the result through MD5 check. The connection was stable throughout the entire transmission process, with no packet loss, timeout or retransmission. The results show that the system can reliably handle the transmission of medium-sized files, verifying the correctness and robustness of the protocol design and server implementation.



```
crzhu@crzhu-virtual-machine:~/Desktop$ python3 server.py
2025-11-14 14:34:49-STEP[INFO] Server is ready! @ server.py[665]
2025-11-14 14:34:49-STEP[INFO] Start the TCP service, listing 1379 on IP All ava
ilable @ server.py[666]
```

```
(.venv) PS F:\code\python_code\cw1> python client.py --server_ip 192.168.88.129 --id 1 --f sample_10mb.txt
--- STEP Protocol Client ---
2025-11-14 14:25:49,759 - INFO - Connecting to 192.168.88.129:1379
2025-11-14 14:25:49,760 - INFO - Upload file: sample_10mb.txt
2025-11-14 14:25:49,760 - INFO - Threads: 4, Retries: 3
2025-11-14 14:25:49,774 - INFO - Connected
2025-11-14 14:25:49,774 - INFO - Task 2: Login
2025-11-14 14:25:49,775 - INFO - Login successful
2025-11-14 14:25:49,775 - INFO - Token: MS4yMDI1MTExNDE...
2025-11-14 14:25:49,775 - INFO - Uploading: sample_10mb.txt
2025-11-14 14:25:49,775 - INFO - Threads: 4, Max retries: 3
2025-11-14 14:25:49,777 - INFO - Upload plan: 633 blocks, 20480 bytes each
Uploading: 100%|████████████████████████████████████████| 633/633 [00:01<00:00, 325.09block/s]
2025-11-14 14:25:51,753 - INFO - === Upload Performance ===
2025-11-14 14:25:51,753 - INFO - File size: 12.35 MB
2025-11-14 14:25:51,753 - INFO - Upload time: 1978 ms
2025-11-14 14:25:51,754 - INFO - Upload speed: 6.25 MB/s
2025-11-14 14:25:51,772 - INFO - MD5 match - Upload successful
2025-11-14 14:25:51,772 - INFO - Upload completed successfully
2025-11-14 14:25:51,774 - INFO - Connection closed
```

Fig. 3: Server (top) and client (bottom) execution results.

C. Upload Performance Evaluation

Since file size increases the system's upload speed, we suspect that the upload time may not increase linearly as files grow. Therefore, we conducted a series of upload tests on files of different sizes (5,10,15,20 mb) to compare whether the system's upload speed would be affected by file size. In all test cases, the system consistently maintained a throughput of 2.22-3.95 MB/s, and each transmission successfully passed the MD5 integrity verification. The upload time is directly proportional to the file size: 451 milliseconds for a 1 MB file, 2573 milliseconds for a 10 MB file, and 5113 milliseconds for a 20 MB file. This indicates that the larger the file, the multiple of the upload time increases, and it is not affected by the size of the file.

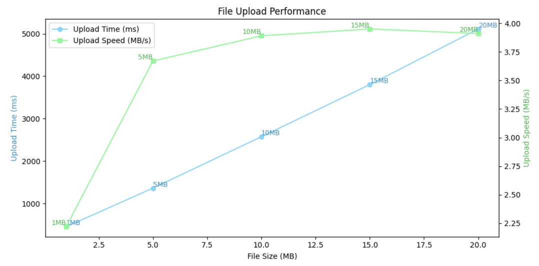


Fig. 4: Upload performance across different file sizes

D. Cross-Format Upload Stability Test

Since the upload form is converted to binary data for upload, different file types may affect the upload speed and stability. Therefore, we created files of different types with a constant size of 10MB to explore whether the upload speed would affect the result under the premise of consistent size. As shown in the figure, the final result - the upload time difference is very small, ranging from 2473 milliseconds to 2756 milliseconds (corresponding to a transfer rate of 3.63–4.04 MB/s). In terms of stability, all files passed the MD5 integrity verification. This experiment not only proves that our system has a stable upload capability for different file types, but also indicates that the file type does not affect the upload speed.

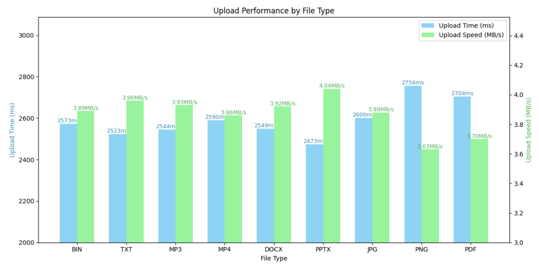


Fig. 5: Upload stability across different file formats

E. Multithreaded Upload Performance Evaluation

To test the performance of the step system in uploading under different threads, we compared four scenarios To examine the upload performance of the step system under different threads, we compared four scenarios, namely 1, 4, 8, and 16 worker threads, and tested files of four sizes ranging from 1 MB to 20 MB. The results show that when the number of threads increases, the transmission time is shortened regardless of the file size, among which the improvement is most significant when there are 4 and 8 threads. When the number of threads further increases, the improvement in transmission speed gradually slows down as coordination and scheduling take the lead. These results indicate that for this system, using a moderate number of threads can achieve the most reliable and efficient performance.

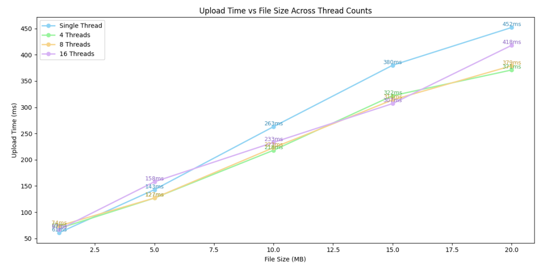


Fig. 6: Multithreaded upload performance

VI. Conclusion

This project builds a file-transfer system on top of the STEP protocol. By implementing a unified packet format, token-based authentication, and block-wise transmission with MD5 verification, the system is able to transfer files reliably from client to server. Experiments that we conducted show that transmissions were very stable and had almost no errors. Moreover, the throughput was mainly constrained by the underlying network bandwidth rather than the project itself. Several improvements on the server side also enhance its ability to handle multiple concurrent requests without crashing. Future work could include resumable uploads, stronger security measures, and optimized concurrency to further improve efficiency.

Acknowledgment

All group members contribute the same percentage.

References

[1] M. Jagadeeswari, P. N. Karthi, V. A. Nitish Kumar, and S. L. S. Ram, "A Secure File Sharing and Audit Trail Tracking Platform with Advanced Encryption Standard for Cloud-Based Environments," in *2023 4th International Conference on Electronics and Sustainable Communication Systems (ICESC)*, Coimbatore, India, 2023, pp. 540-547, doi: 10.1109/ICESC57686.2023.10193389.

[2] R. L. R. Maata, R. Cordova, B. Sudramurthy and A. Halibas, "Design and Implementation of Client-Server Based Application Using Socket Programming in a Distributed Computing Environment," *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, Coimbatore, India, 2017, pp. 1-4, doi: 10.1109/ICCIC.2017.8524573.

[3] J. Funasaka and N. Hiraoka, "Evaluation on Segmented File Download Methods Using Parallel TCP/MPTCP Connections on Multiple Paths," *2023 IEEE International Conference on Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing; Cloud and Big Data Computing; Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, Abu Dhabi, United Arab Emirates, 2023, pp. 0214-0219, doi: 10.1109/DASC/PiCom/CBDCom/Cy59711.2023.10361460.

[4] H. Wang and C. Williamson, "A New Scheme for TCP Congestion Control: Smooth-Start and Dynamic Recovery," *Proceedings of the Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Montreal, QC, Canada, 1998, pp. 69-76, doi: 10.1109/MASCOT.1998.693677.