Team Members

Name: Qi Cao
Student ID: 2360908

Name: Chengyang Du
Student ID: 2360818

Name: Haoyan Xu
Student ID: 2359977

Name: Jingchen Ding
Student ID: 2253158

File Transfer System Based on STEP

Qi Cao (2360908), Chengyang Du (2360818), Haoyan Xu (2359977), Jingchen Ding (2253158)
Information and Computer Science
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
Suzhou, Jiangsu, China

***Abstract*—In recent years, with the rapid development of the Internet, a large number of files and messages are exchanged globally. In this project, we design and implement a client-server (C/S) network file transfer system based on the STEP protocol, enabling secure and reliable data exchange. The system supports user authentication, file upload/download, and MD5 integrity verification. This paper begins with an overview of the project and a brief introduction to the STEP protocol. Then, we present a detailed design of the system architecture, including its core components and communication flow. Furthermore, we analyze the transfer performance under various file sizes, evaluating throughput and efficiency trends. Finally, we conclude with a summary of project and potential directions for future work.**

***Index Terms*—C/S Architecture, TCP, STEP**

I. Introduction

A. Background

File transfer has been a foundational network application, dating back to early protocols like FTP [1]. As reliance on secure data transfer grows, modern application-layer protocols like the Simple Transfer and Exchange Protocol (STEP) plays a critical role.

STEP is a TCP-based protocol. Its core strength lies in balancing speed and reliability: it enables fast file transfers without compromising data integrity, making it ideal for scenarios where both efficiency and accuracy are critical.

B. Task Specification

There are three primary tasks in this project:

1) Debug the provided server code, so it runs and prints "Server is ready".
2) Implement client-side authentication using the specified student ID and MD5-hashed student ID in order to obtain a valid authentication token.
3) Create file-level upload functionality (e.g., read the file upload plan from the server, upload file blocks, and check integrity using MD5) in a way that uses Python's socket, hashlib, struct and tqdm libraries and meets the specification requirements for STEP protocol message formats, required fields, and status codes.

Additionally, we modify the original server and add functionality to client to achieve secure multi-threading transfer. The original log tracking in server is replaced with in-memory "upload_states", guarding each (username, key) with a **Lock**.

C. Challenge

We identified two primary challenges during implementation: performance bottlenecks and data integrity.

The baseline single-threaded implementation was severely constrained by I/O bottlenecks, failing to meet efficiency objectives for large files. Concurrently, the server's non-atomic, log-based state management for tracking received blocks lacked mutual exclusion, creating a significant risk of race conditions and data corruption.

To address these deficiencies, we engineered a reliable multi-threaded architecture, which will be detailed in the following sections.

D. Practice Relevance

This project's architecture serves as a foundation for robust, high-throughput file transfer systems. Its design offers the scalability required for institutional use, such as a university-wide file service, and the extensibility to integrate with modern data-processing pipelines, including database ingestion or data analysis via LLM APIs.

E. Contributions

The group work together seamlessly, successfully finishing all four tasks mentioned ahead. A Client-Server file upload system is constructed with reliable multi-threaded options and clear output information during operation.

II. Releated Work

Apart from STEP, prior innovations in network traffic redirection have largely targeted specific optimization goals. For instance, Topology-aware Resource Adaptation (TARA) [2] dynamically reroutes traffic to minimize energy consumption and alleviate congestion. Similarly, Software-Defined Edge (SDE) [3] leverages SDN and TCP duplication to redirect traffic specifically for high availability during server failures. However, these approaches are constrained by their specific use cases (energy or failover). In contrast, STEP represents a critical advancement by offering a more generalized mechanism that decouples the connection state from the network path, thereby addressing the limitations of these traditional redirection schemes.

III. Design

A. Server Debugging

We find 8 mistakes in total.

1) The `struct` package was imported to support the `make_package` function.
2) Renamed `tcp-listener` function to `tcp_listener`.
3) Changed SOCK_DGRAM in `server_socket = socket(AF_INET, SOCK_DGRAM)` to SOCK_STREAM.
4) Changed `op_save` in block `if request_operation == op_save:` to uppercase `OP_SAVE`.
5) Added `th.start()` in the `tcp_listener(server_ip, server_port)` function.
6) Changed `'r'` to `'rb'` in `get_file_md5(filename)` function with `with open(filename, 'r') as fid`.
7) Changed 16 to 8 in `while len(bin_data) < 16:` to satisfy the STEP protocol format requirement.
8) Modified the `file_process` function: changed the `if FIELD_KEY not in json_data.keys():` block to include error logging and send a response packet with status code 410, then return.

B. C/S network Architecture Design

Fig. 1 shows an example of client-server network architecture. The system uses a classic C/S architecture over TCP port 1379 (STEP-specified):

1) **Server:** Passively listens for client requests, accepts connections, processes requests, and returns responses.
2) **Client:** Proactively requests services (login, upload plan, block transfer, integrity verification).

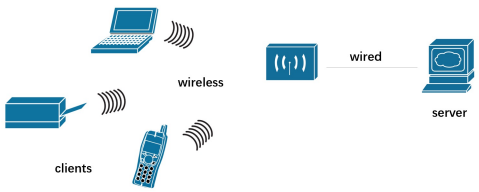

Fig. 1. Example of a client-server network architecture.

Fig. 2 presents the sequence diagram illustrating the interaction between the client and the server. The workflow consists of five main steps:

1) TCP connection established (port 1379)
2) Authentication: Client sends AUTH-type request to log in; Server replies with return token and status code.
3) Initiate upload: Client sends FILE/SAVE request to send upload plan; Server responds with a key, block size, and total blocks.

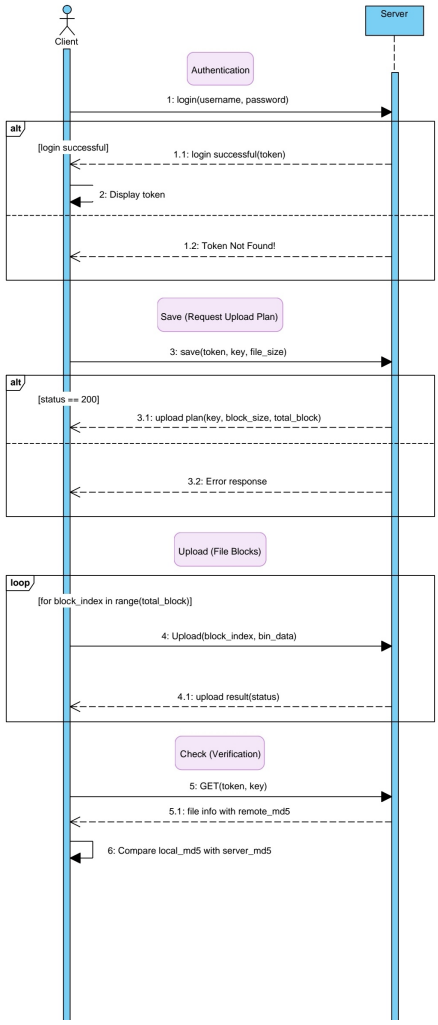

Fig. 2. Sequence diagram of the interaction between client and server.

4) Block uploads: Client sends UPLOAD-type requests individually for every block; the server replies to acknowledge the request with a status code.
5) Integrity check: Client sends FILE/GET request to receive the checksum for MD5 verification.

C. Algorithm

The pseudo codes for the authorization and upload function is presented below. The **save and verify** procedures in the upload workflow which are omitted here will be detailed in the next section.

IV. Implementation

A. Develop Environment

Throughout the project, several team members participated in the coding process. The majority of the development work was performed on two operating systems shown in the Table I and II below.

Algorithm 1: Authorization Algorithm

```
Input: student_id
password ← make_password(student_id)
login_json ← { FIELD_TYPE: TYPE_AUTH,
  FIELD_OPERATION: OP_LOGIN,
  FIELD_DIRECTION: DIR_REQUEST,
  FIELD_USERNAME: student_id,
  FIELD_PASSWORD: password }
message ← make_packet(login_json)
send_packet(socket, message)
resp ← recv_packet(socket)
if resp is None or resp_json.get(FIELD_STATUS) ≠ 200 then
    print("Login failed: " +
      resp.get(FIELD_STATUS_MSG, "Unknown error"))
end
else
    token ← resp[FIELD_TOKEN]
    print("Login successful. Token: " + token)
end
```

TABLE I
MACOS DEV ENVIRONMENT

	Specification
OS	macOS Sequoia 15.3.1
CPU	Apple M4
RAM	16GB
IDE	VS Code & Neovim
Python	3.12
PM	uv

TABLE II
WINDOWS DEV ENVIRONMENT

	Specification
OS	Windows 11
CPU	Intel 13-i7
RAM	16GB
IDE	VS Code
Python	3.13.9
PM	anaconda

B. Steps of implementation

- Carefully read the coursework requirements and make a to-do list, then assign tasks.
- Draw a sequence diagram to illustrate the interaction between the client and the server, and construct a flowchart to clarify the overall workflow of the system.
- Comprehend the logic of the provided server code. Identify and fix the existing bugs.
- Start developing the client code based on the sequence diagram. Conduct unit tests emphasizing on each request-response cycle.
- Complete the first version of the project and upload it to GitHub.

Algorithm 2: Upload Algorithm (Client-Side Kernel)

```
Input: socket, token, file_path
Function upload_file_workflow(file_path, token):
    file_key ← os.path.basename(file_path)
    file_size ← os.path.getsize(file_path)
    save_request ← make_message(OP_SAVE,
      TYPE_FILE, file_size, file_key)
    Send save_request To server
    plan ← Response From server
    if plan[status] ≠ 200 then
        Print("SAVE failed:" + plan[status_msg])
        return False
    end
    key ← plan[FIELD_KEY]
    block_size ← plan[FIELD_BLOCK_SIZE]
    total_block ← plan[FIELD_TOTAL_BLOCK]
    Open file_path as file in binary mode
    for block_index ← 0 to total_block - 1 do
        bin_data ← file.read(block_size)
        upload_request ←
          make_message(OP_UPLOAD, TYPE_FILE,
          block_index, bin_data)
        Send upload_request To server
        result ← Receive response From server
        if result[status] ≠ 200 then
            Print("Block upload failed")
            return False
        end
    end
    server_md5 ← verify_upload(socket, token, key)
    if server_md5 == get_file_md5(file_path) then
        Print("Upload verified")
        return True
    end
    else
        Print("MD5 mismatch!")
        return False
    end
```

- Optimize the first completed version and introduce multi-threading.
- Refactor the final code for better readability and maintainability, and add explanatory inline comments where necessary.
- Collect and analyze all test results, and finalize the project report.

C. Programming skills

1) Parallel Programming: The system utilizes parallel programming for multi-threaded file uploads. The client establishes multiple concurrent sockets for simultaneous block transfer, while the server employs lock mechanisms to ensure thread safety during concurrent data access.

2) Functional Programming and Modular Design: The solution adopts a modular, functional design, composing logic from discrete, single-purpose functions rather than classes. This approach enhances maintainability, simplifies unit testing, and promotes code clarity through a clear separation of concerns.

D. Program flow chart

Fig. 3 illustrates the program flow chart.

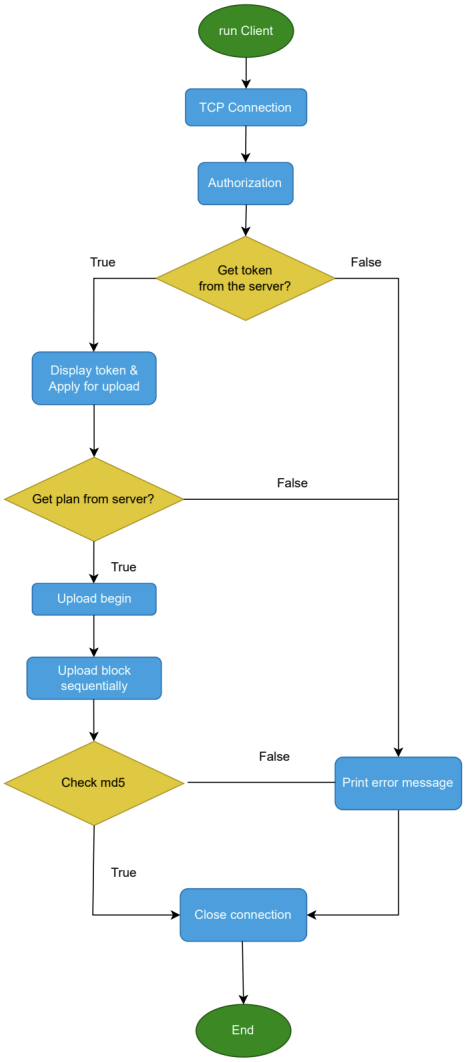

Fig. 3. Program flow chart

E. Actual implementation

1) Authorization function: The authorization function is implemented through the following steps. First, a client socket is created and connected to the server. Then the user input the student id, then `make_password` function calculate the MD5 value of the id and return it as password. After that, a packet form of login request is created by `make_packet` function, and then `send_packet` function sends it to the server. After receiving the response from the server by `recv_packet` function, the `validate_response` function check the response is correct or not. If the response is right, the token will be extracted from the response and printed.

2) File uploading function: The uploading function is executed after authorization. First, the user provides the file path, from which the file size and the file key are derived. A JSON-formatted save request is then constructed and converted into a packet using `make_packet`, before being sent to the server via `send_packet` function. After receiving the response from the server by `request_save` function, the upload plan is obtained, and the `upload_blocks` function uploads file accordingly. Finally, the `verify_upload` function returns the MD5 checksum of the uploaded file which will be compared with that of the local file. If the two values match, the message "Upload verified successfully!" is displayed. In addition, the `upload_blocks` function supports configurable multi-threading through the *block_workers* parameter. When multiple workers are used, each thread opens its own TCP connection and uploads different file blocks in parallel. A shared block index counter, guarded by a lock, assigns block indices to workers without duplication.

F. Difficulties

1) Workflow Establishment: A primary challenge was establishing an efficient workflow for the newly formed team. This was resolved by implementing **Github** for version control and **Microsoft Teams** for communication and task management. This toolset streamlined development and centralized project artifacts.

2) Vulnerable Multi-threaded Transfers: Intermittent failures during multi-threaded transfers were traced to a server-side **race condition**. The cause was a non-thread-safe, file-based state tracking system in the original server. This was resolved by implementing a dynamic, in-memory state model protected by a **lock**, which eliminated the vulnerability with negligible performance overhead. Testing results will be demonstrated in the next section.

V. Testing and Results

A. Testing environment & Settings

All the results below were tested the macOS environment mentioned above. It is important to note that there are **two versions of server**: the original debugged one(server.py) and the one with safe thread control(safe_server.py). **All tests except the multi-threaded test were performed on the original debugged server, following the project requirement.** To offer a more comprehensive analysis, the upload throughput is also recorded during testing, throughput is calculated by:

$$\text{Throughput (MB/s)} = \frac{\text{File Size (Bytes)}}{\text{Upload Time (s)} \times 1024^2} \quad (1)$$

B. Basic flow test

Here, we demonstrate "Task 1-3" with snapshots: Starting the server:

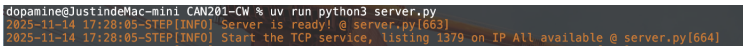

Fig. 4. Start server

Launching the client to upload a test file(around 10MB) and checked the received file successfully by MD5(Fig.5):

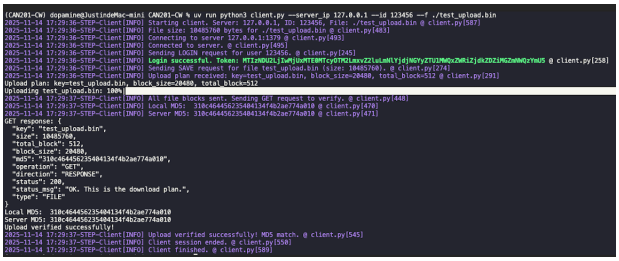

Fig. 5. Client upload

C. Single-threaded benchmark test

The following content demonstrates the average performance in the single-threaded, multiple file size condition with uploading time and throughput as metrics. File sizes: 256(KB), 1, 5, 10, 20, 50(MB).

1) Transfer Time: As shown in Fig. 6, the growth of time related to file size exhibits a approximately linear relationship, demonstrating the stability of transfer speed.

2) The Influence of Fixed Protocol Overhead: As observed in Fig. 7, the 256K file size exhibits a notably high throughput coefficient of variation (61%). This variability is attributed to fixed protocol overheads (e.g., connection handshakes), which constitute a significant portion of the transaction time for small data payloads. As the file size increases, this fixed setup cost is effectively amortized over a larger data transfer, leading to progressively more stable and predictable performance.

3) Throughput Inflection and Saturation: Unlike the linear growth in transfer time, throughput exhibits non-linear behavior, reaching a peak of 106.05 MB/s with the 20MB file size (1024 blocks). This peak likely signifies optimal TCP buffer utilization and overhead amortization. The subsequent marginal decline in throughput observed with larger file sizes suggests the system is encountering resource contention, potentially due to memory pressure or network buffer saturation. **(Insight goes to conclusion)**

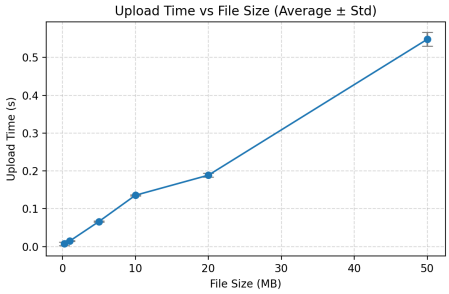

Fig. 6. single-threaded benchmark time

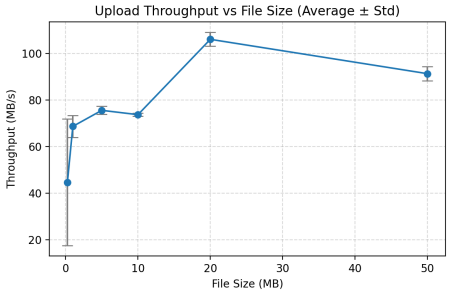

Fig. 7. single-threaded benchmark throughput

D. Multi-threaded benchmark test

In order to find the performance boundary, we also test the multi-threaded setting. Here we demonstrate average performance in the multi-threaded, single file size condition with uploading time and throughput as metrics. We measure the performance with different numbers of threads. The original server and the thread-safe server are compared in performance. File size: 50MB.

1) General view: similar performance of two versions & 1 to 2 leap : Demonstrated in Fig. 8 and 9, the safe version and original exhibits a similar performance, showing the efficient implementation and right design choice of our safe version. The 35% throughput increase and 18% time reduction from 1 to 2 threads represents the largest marginal benefit. This suggests the effectiveness of our multi-threaded implementation, demonstrating that parallel TCP connections effectively mask client-side I/O stalls and protocol overhead.

2) Convergence Indicates System-Level Saturation: Both implementations converge to a peak throughput of approximately 173MB/s at 16 threads. This saturation point indicates that performance becomes limited not by thread management overhead, but by a system-level bottleneck (e.g., network bandwidth, disk I/O, or CPU). Interestingly, **this result contrasts markedly with identical tests on Intel-based hardware**, which yielded significantly longer time and slower convergence. The performance disparity strongly suggests the observed throughput is amplified by **Apple Silicon's architectural optimizations**, such as its Unified Memory Architecture (UMA) and custom storage controller, which

are highly effective at mitigating I/O bottlenecks under high concurrency.

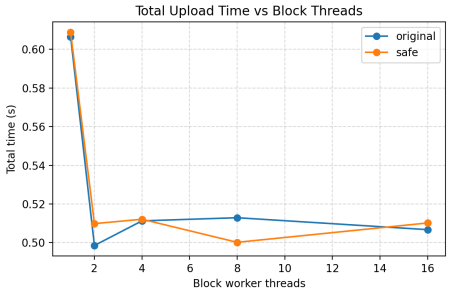

Fig. 8. multi-threaded benchmark time

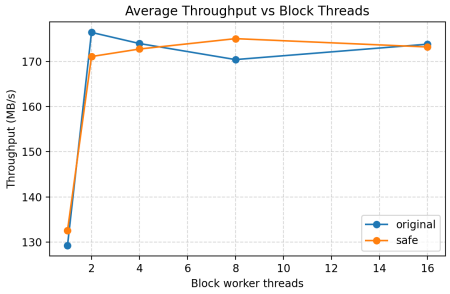

Fig. 9. multi-threaded benchmark throughput

VI. Conclusion

In conclusion, this project successfully fulfilled its three primary objectives. Our empirical evaluation confirmed that system-level I/O saturation, rather than threading overhead, is the primary performance bottleneck at high concurrency. Future work should focus on two key areas: mitigating the performance drop in small-file scenarios and conducting a deeper architectural analysis to explain the significant performance disparities observed between different chip architectures.

VII. Acknowledgment

An even proportion of 25% for all members.

References

[1] N. A. John, “File sharing and the history of computing: Or, why file sharing is called ‘file sharing’,” *Critical Studies in Media Communication*, vol. 31, no. 3, pp. 198–211, 2013, doi: 10.1080/15295036.2013.824597.

[2] J. Kang, Y. Zhang, and B. Nath, “TARA: topology-aware resource adaptation to alleviate congestion in sensor networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 7, pp. 919–931, 2007.

[3] V. Gramoli, G. Jourjon, and O. Mehani, “Can SDN mitigate disasters?” *arXiv preprint arXiv:1410.4296*, 2014.