A High-Performance STEP Protocol Implementation for Secure Client-to-Server File Uploads

1st Yize Liu *2254472* | 2nd Shengtian Huang *2254461* | 3rd Qing Qin *2254084* | 4th Xu Chen *2257453* | 5th Zichen Qiu *2252705*

***Abstract*—With the continuous development of computer technology and communication technology, the transmission of information has become more and more convenient, but it is also often accompanied by security risks and inefficient handling of large files. In this report, we present a high-performance implementation of the Simple Transport and Exchange Protocol (STEP) for secure client-to-server file uploads. Our system implements a three-stage workflow including authentication, upload preparation, and block-based transfer, with real-time progress tracking and MD5 integrity verification. Performance testing shows linear growth in file size and significant efficiency gains through multi-threading while maintaining consistent performance across different file types. This implementation provides a strong and reliable foundation for secure file transfer applications.**

I. INTRODUCTION

Efficient and reliable file transfer mechanisms are critical for the success of numerous network applications, particularly those utilizing the client-server architecture [1]. With the exponential growth of data volumes across industries, the demand for robust file transfer protocols has become increasingly urgent. The client-server model, as a widely adopted architecture, forms the backbone of many modern applications, including cloud storage, content distribution, and enterprise file sharing [2].

Despite the availability of numerous file transfer protocols, many existing solutions face challenges such as inefficient handling of large files and poor fault tolerance in the face of network failures. In addition, many protocols fail to provide the necessary encryption and authentication mechanisms to secure the transfer of sensitive data [1]. Thus, the challenge lies in developing a protocol that balances efficiency, security, and robustness in the context of modern network demands.

This project proposes the implementation of the Simple Transfer and Exchange Protocol (STEP) to address these challenges. The key tasks with STEP include debugging the server-side application, which eliminates critical syntax errors, and developing the client-side functionalities to handle authorization, file uploads, and data integrity verification.

The STEP protocol has significant potential applications across. In cloud storage, it enables efficient handling of large files through block-based upload mechanisms, while its built-in authentication and secure transfer capabilities make it well-suited for enterprise-level file sharing environments [3]. Additionally, the protocol's versatility can be leveraged in systems such as weather stations for transmitting large sets of data, content distribution networks for optimized data delivery, and backup systems for ensuring the integrity of archived files [3]. By offering a solution that integrates both efficiency and security, STEP can serve as a foundational protocol for a wide range of real-world applications.

Our Contributions:

- **Bug Fixes in Server-Side Code**
- **Client-Side Implementation**
- **File Upload Mechanism**
- **Error Handling**
- **Data Integrity Verification**

II. RELATED WORKS

A. Application Perspective: Cloud Storage Technology

In the domain of cloud storage, Zhao et al. [4] explored the integration of FPGA technology into cloud storage systems, demonstrating its potential to enhance data transfer efficiency. Their work highlights how FPGA's reconfigurability can adapt to varying network demands, which is crucial for optimizing performance in traffic redirection applications.

B. Technical Perspective: Traffic Redirection Protocols

Chand et al. [5] proposed the TRCCTP protocol, which addresses congestion control in wireless sensor networks by intelligently redirecting data traffic. This approach not only alleviates congestion but also improves overall network performance, providing a practical solution for efficient traffic redirection in sensor networks.

C. Research Perspective: Load Balancing and Traffic Redirection

Liu et al. [6] introduced a load balancing-based real-time traffic redirection method for edge networks. By leveraging OpenFlow controllers to dynamically adjust flow paths, their method optimizes network performance and reduces latency, offering a promising framework for traffic redirection in edge computing environments.

III. DESIGN

A. Explanation of Architecture

The system architecture, as depicted in Fig. 1, follows a two-tier client/server structure with the application of STEP. The STEP client is a program that establishes a connection with the STEP server via the Internet to send requests and receive responses. The client sends user requests over a local area network (LAN) using TCP/IP and STEP protocols, and the server processes these requests and provides the corresponding service. A SOCKS-compatible proxy handles redirected

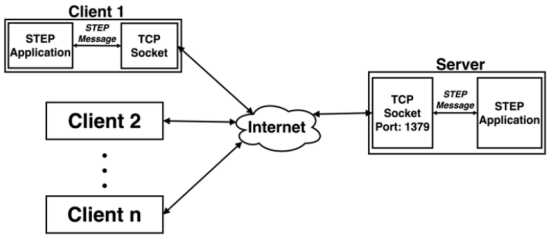

Fig. 1. C/S architecture

connections intercepted by the client node's redirector. After the server processes the client's request, it sends the data back to the client, which then computes the data and displays the results to the user. In addition, multiple clients (no more than 20) can connect to one server simultaneously based on the multithreading functionality provided by the server.

B. Workflow of Program

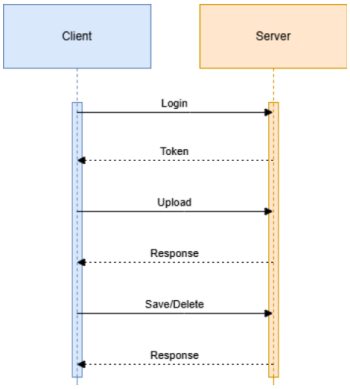

Fig. 2. C/S communication

Our file transfer system implements a comprehensive three-phase workflow for secure and reliable file transmission over TCP/IP networks. The process begins with the authentication phase, where the client establishes a TCP connection to the server on port 1379 and initiates a login request. This request follows the STEP protocol format, containing a JSON structure with the student ID as username and its MD5-hashed value as password. Upon successful authentication, the server issues a unique token that serves as a security credential for all subsequent operations.

The second phase involves file upload preparation, where the client analyzes the target file to determine its size and sends a SAVE request to the server. This request includes essential file metadata (filename, file size) along with the previously obtained authentication token. The server processes this information and responds with a detailed upload plan, specifying the optimal block size and calculating the total number of blocks required for the transfer based on a maximum packet size of 20KB.

In the final phase, the client executes the block-based file transfer by systematically dividing the file into blocks according to the server's specifications. Each block is transmitted sequentially with its corresponding block index and the authentication token, while the system maintains real-time progress tracking using the tqdm library to provide visual feedback on the upload status. After all blocks are successfully transmitted, the server generates an MD5 hash of the received file, which the client uses to verify the integrity of the transferred data, ensuring that no corruption occurred during transmission.

C. Algorithms

Algorithm 1: Authorization and Token Acquisition

Input: server_ip, server_port, student_id
Output: token, socket
socket ← CreateTCPSocket();
Connect(socket, server_ip, server_port);
username ← ToString(student_id);
password ← MD5Hash(username);
user_data ← {"type": "AUTH", "operation": "LOGIN", "username": username, "password": password};
login_packet ← MakeRequestPacket(user_data);
Send(socket, login_packet);
json_data, bin_data ← GetTCPPacket(socket);
token ← json_data["token"];
return token, socket;

In the client code, we designed three main functions to handle the STEP protocol workflow. The connection_and_token() function establishes TCP connection and obtains authorization token using MD5-hashed student ID. The save_file() function prepares upload by exchanging file metadata with server. The upload_file() function performs the actual file transfer in 20KB blocks with real-time progress tracking. These main functions are supported by utility functions make_packet(), make_request_packet(), and get_tcp_packet() for message handling.

IV. IMPLEMENTATION

A. Development Environment

To validate the designed STEP client code, two virtual machines with identical specifications (Ubuntu 20.04.4 LTS 64-bit, AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx 2.30GHz CPU, 2GB RAM, and Intel PRO/1000 MT desktop VirtualBox Host-Only Ethernet Adapter) are used as prototypes. The corresponding host environment consists of Windows 11, AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx 2.30GHz, 8GB RAM, and Intel PRO/1000 MT desktop. The implementation was carried out using Python 3.8.10, leveraging its extensive standard library and third-party package ecosystem. PyCharm Professional 2023.2 served as our primary integrated development environment, offering advanced debugging and code analysis capabilities. Git 2.25.1

Algorithm 2: File Save and Upload

```
Input: token, socket, file_path
Output: md5 or error
file_size ← GetFileSize(file_path);
save_request ← {"type": "FILE", "operation":
 "SAVE", "key": file_path, "size": file_size, "token":
 token};
Send(socket, MakeRequestPacket(save_request));
json_data ← GetTCPPacket(socket);
if json_data.status = 200 then
    file_content ← ReadBinaryFile(file_path);
    for i = 0 to json_data.total_block - 1 do
        start_index ← i * MAX_PACKET_SIZE;
        end_index ← min((i+1) *
         MAX_PACKET_SIZE, file_size);
        block_data ←
         file_content[start_index:end_index];
        upload_request ← {"type": "FILE",
         "operation": "UPLOAD", "key":
         json_data.key, "block_index": i, "token":
         token};
        Send(socket,
         MakeRequestPacket(upload_request,
         block_data));
        response ← GetTCPPacket(socket);
        if response.status ≠ 200 then
            return Error("Upload failed at block " + i);
        end
        UpdateProgressBar(i, json_data.total_block);
    end
    return response.md5;
end
return Error("Save request failed");
```

was employed for version control, enabling efficient code management and collaboration.

Additionally, the implementation relied on several key Python libraries to achieve its functionality. The socket library formed the foundation for TCP/IP communication, while hashlib was utilized for MD5 hash calculation to ensure file integrity. Data serialization and deserialization were handled through the json library, with struct managing binary data structure operations. For user interface elements, we incorporated tqdm to provide visual progress feedback during file transfers. The system's concurrent processing capabilities were implemented using the threading library, and comprehensive system monitoring was achieved through the logging module.

B. Steps of Implementation

First, review the server code to identify any bugs:

- import struct
  The struct module is used in the code for binary data packaging and unpacking, and this module needs to be imported.
- def getfile_md5(filename) → def get_file_md5(filename)
  The function calls in the code should be consistent before and after, and they should adhere to Python's PEP 8 coding standards.
- def Tcp_Listener(server_port, server_ip) → def tcp_listener(server_port,server_ip)
  The case of function calls should be consistent before and after, meaning that the capitalization of function names should be uniform throughout the code.
- Add "server_socket.listen(20)"
  In the code, the server should activate the listening state when establishing the connection.
- Add "connection_socket, addr = server_socket.accept()"
  To enable the server to accept client connections, this line must be added to the code:
- tcp_listener(server_ip, server_port)
  Uncomment this code.

The client-side implementation was completed through the following key steps: (1)Initial Setup: Imported necessary libraries and constants, defined max packet size, and created argparse function for command line argument parsing. (2)Connection Management: Established server connections using connection_and_token function for authentication handling and token retrieval. (3)Packet System: Implemented make_packet and make_request_packet functions for packet generation, while maintaining segment boundaries with get_tcp_packet function. (4)File Transfer Core: Used save_file function to handle file operations, implemented block upload system with progress tracking and automatic retry functionality. Verification System: Added MD5 checksum calculation and file integrity checks. Each component underwent thorough testing before integration, ensuring robust error handling and data integrity throughout the system.

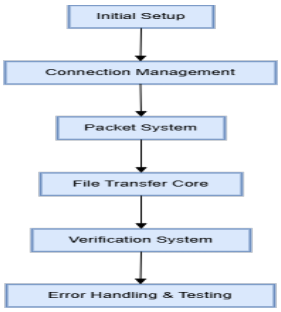

Fig. 3. Flowchart of implementation

C. Programming Skills

- **OOP:** The code utilizes the encapsulation principle of OOP within the STEP protocol by structuring object properties as global variables and arranging their associated methods beneath them. This object-oriented methodology aids in minimizing code redundancy while simultaneously promoting code reuse. By compartmentalizing functionality to address distinct tasks, it enhances code readability and maintainability.

- **Parallelism:** In the context of the file upload function, the client can adopt a multi-threading strategy, utilizing the computing power of multiple cores or processors, and performing multiple tasks simultaneously can significantly improve program performance and efficiency to facilitate file uploads. This aspect will be further elaborated upon in the forthcoming Testing and Results section.

D. Actual Implementation

1) The authorization function: The authentication function is implemented through the following steps: First, create a client socket and connect to the server (client_socket = socket(AF_INET, SOCK_STREAM); client_socket.connect((server_ip, server_port))); then obtain the username (student ID) and calculate its MD5 value as the password (username = str(args.id); password_md5=hashlib.md5(username.encode()).hexdigest()); next, create a login request packet containing the username and password and send it (login_packet = make_request_packet(OP_LOGIN, TYPE_AUTH, user)); finally, receive the server response and extract the token (json_data, json_bin = get_tcp_packet(client_socket); token = json_data[FIELD_TOKEN]).

2) File Uploading Function: First, obtain the file information and construct a save request (size = os.path.getsize(file_path); save_request = FIELD_KEY: file_path, FIELD_SIZE: size, FIELD_TOKEN: token), send the request to the server and obtain the upload plan (client_socket.send(make_request_packet(OP_SAVE, TYPE_FILE, save_request))); then read the file data (with open(file_path, 'rb') as f: image_data = f.read()) and segment it according to the block size returned by the server; use tqdm to implement progress monitoring, construct an upload request for each data block (upload_request ={FIELD_KEY: json_data[FIELD_KEY], FIELD_BLOCK_INDEX: i, FIELD_TOKEN: token}), send it to the server and process the response; finally, ensure data integrity through MD5 verification (if server_md5.lower() == local_md5.lower(): logger.info("MD5 verification successful")). The entire implementation process includes a complete error handling mechanism and records operation status through the logging system to ensure the reliability of file transfer.

E. Difficulties

During the project, we encountered two main technical challenges. First, we faced the issue of complex and difficult-to-understand server code. To address this challenge, we adopted a solution of drawing sequence diagrams, using visualization to illustrate the interaction process between the client and server. This visual approach not only helped us gain a clearer understanding of the server's operational mechanisms but also provided a more definite direction for subsequent client development. Second, we faced the challenge of file transfer efficiency. We experimented with various optimization methods including single-threading, multithreading, and multiprocessing, though we discovered that multithreading and multiprocessing were not applicable in all scenarios.

V. Testing and Results

A. Testing Environment

The same test environment as the implementation is used to test our implementation and display the results.

B. Testing Steps

Basic Flow Testing: The following test steps and screenshots are based on transferring the image test.jpg(533.90 KB). The test process consists of three main steps: First, debug the server code by fixing six syntax bugs in 'server.py', then verify success by running "python3 server.py" on the server VM terminal, which displays "Server is ready!". Second, perform authorization by checking if the token is obtained after logging in with username; this involves parsing command-line arguments (ID, path, IP, and port), creating STEP protocol packets for requests while maintaining segment boundaries, establishing connection, obtaining and saving token, and uploading files with necessary libraries and constants. This can be verified by running "python3 client.py —server_ip 10.0.2.6 —id 2254472 —f test.jpg" on the client VM terminal, which displays the token. Finally, verify successful file upload from client to server by checking the progress bar in the terminal window, along with MD5 information for file integrity verification, with the server-side page showing successful completion of the operation (setup_logging() is added to the client code to display and log the key information of the uploaded file in order to make the testing data easy to be logged). The graph shows the results of the basic test.

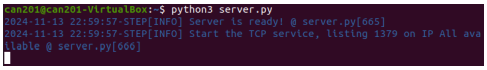

Fig. 4. Server is ready

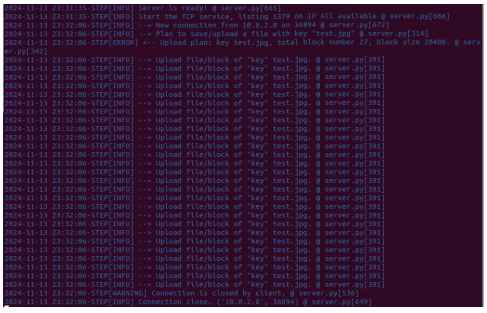

Fig. 5. Server running results

C. Explanation of Architecture

Performance Testing: In order to test the impact on transmission performance under different conditions, the following tests were performed (all tests strictly adhered to the control variable method). In addition, to determine the exact upload

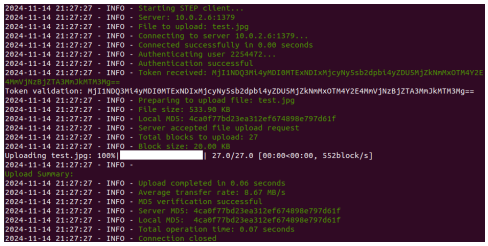

Fig. 6. Client running results

time for a file, we performed 5 identical upload operations of the same file under identical environmental conditions, and used the average value as the upload time. And to obtain more intuitive results, we saved the client-side recorded data as a txt file, and then used the matplotlib library in Python to visualize the recorded data:

1) To demonstrate the application's performance, in this set of experiments, we tested by uploading txt files of six different sizes, ranging from 1MB to 20MB. This was done to study the relationship between file size (MB) and upload time (s), and we used a line graph to represent this relationship. Additionally, we tested file transmission across hosts in different network environments, including Wi-Fi and iPhone 13 Pro Max mobile hotspot, both based on the 802.11ax protocol, for comparison.(Fig.7 and Fig.8)

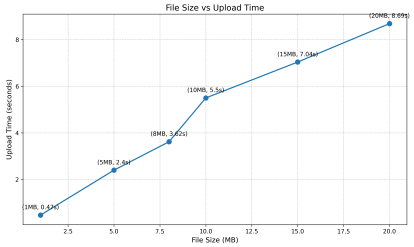

Fig. 7. File size relation

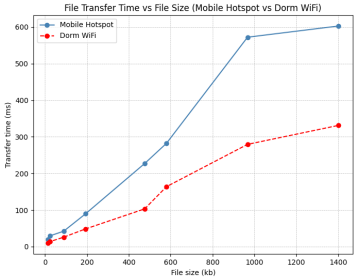

Fig. 8. Network environment relation

2) Then, to test the authenticity and robustness of the results, we examined how different types of files, with the same file size, would affect the upload time. We selected files of 10MB in size from the following formats: txt, jpg, png, pdf, and mp4. Similarly, each file was tested at least five times to calculate the average upload time. Additionally, we used matplotlib to visualize the results in a bar chart.(Fig.9)

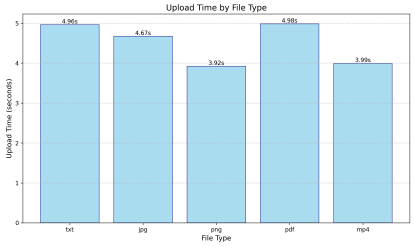

Fig. 9. File type relation

3) Additionally, to study the stability of the program, we tested the impact of file size on the average upload speed. The average upload speed here refers to the mean speed recorded by the client after testing at least five times, with the unit in MB/s. The results are presented using a line chart.(Fig.10)

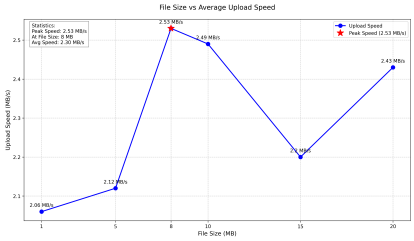

Fig. 10. Mean velocity comparison

4) To consider the program's performance in a real-world environment, we tested the impact on average upload time when multiple clients upload files simultaneously. The average upload time here refers to the total time taken for all clients to complete their uploads, divided by the number of clients. This comparison provides a more intuitive demonstration of how multiple clients affect performance. Similarly, we used matplotlib to visualize the results as both a line chart and a bar chart.(Fig.11)

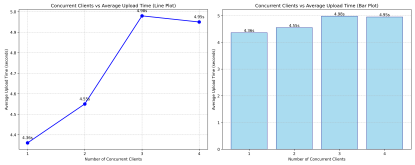

Fig. 11. Multi-client

5) To test the impact of multi-threaded file uploads on overall performance, we implemented multi-threading on the client side. We used a thread pool to manage the multi-threaded uploads, and the number of threads was specified in advance. Finally, we used a line chart

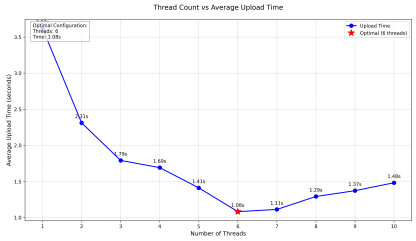

Fig. 12. Multithreading

to show the relationship between the number of threads and the average upload time.(Fig.12)

6) Moreover, to explore the deeper impact of multi-threaded uploads on performance, we compared the percentage difference in upload time when using the same number of threads to upload files of different sizes, versus uploading without multi-threading. The percentage time difference is presented using a bar chart.(Fig.13)

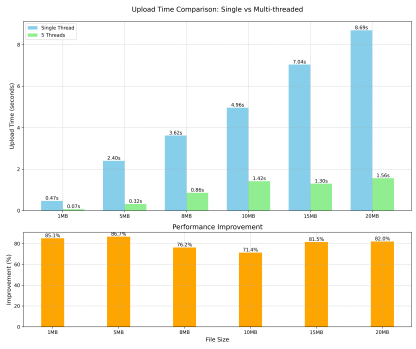

Fig. 13. Multithread comparison

D. Testing results

First, the test results show that there is an approximately linear positive correlation between upload time and file size. When analyzing the factors affecting upload speed, MAX PACKET_SIZE is one of the key factors, as it is a fixed small value that affects the number of blocks. File size is linearly correlated with the number of blocks, and the number of blocks is positively correlated with the program's runtime, which in turn affects the upload speed. Additionally, the performance may vary depending on the network environment.

Secondly, although different file types do affect upload time, the impact is minimal, and the fluctuation in upload time is within an acceptable range. This may also be influenced by the hardware system of the transmitting device and other incidental factors. As for the impact of file size on average upload speed, we can observe from the graph that there is no significant direct relationship between the two. The average upload speed fluctuates, but remains acceptable, indicating that the application is relatively stable.

Furthermore, when multiple clients upload files simultaneously, we observe a trend of increasing upload time as the number of clients increases. This is likely due to competition for network bandwidth and server resources, causing the overall upload time to extend as the number of clients grows.

Apart from this, when comparing multi-threaded file uploads, we see that performance improves with an increase in the number of threads, as the overall upload speed increases. However, we also observe a slight increase in upload time with more threads, which may be due to race conditions. When multiple threads attempt to acquire locks and access shared resources simultaneously, race conditions can occur, reducing overall transmission speed as threads may have to wait for the lock to be released. As for the impact of multi-threaded uploads with different file sizes, we find that the overall efficiency improvement percentage is around 80% for all file sizes, with no significant difference.

VI. CONCLUSION

In summary, This report presented the design and implementation of a high-performance STEP protocol-based client-server file upload system. We developed a robust three-phase workflow encompassing authorization, upload preparation, and block-based transfer with progress tracking. Performance testing under various conditions revealed a linear correlation between file size and upload time, minimal impact of file type, increased upload times with concurrent clients, and improved efficiency through multithreading.

However, there are some limitations in the test. Firstly, during testing, we were unable to account for files of all sizes, and we did not test what would happen with very large files. Secondly, we found that the testing results are significantly influenced by the hardware configuration of the computer. In the future, we plan to make improvements in these areas.

Our work contributes an efficient and secure file upload solution, with future work potentially exploring further optimizations for enhanced performance and reliability.

ACKNOWLEDGMENT

All group members contribute the same percentage .

REFERENCES

[1] T. Zheng, S. Yunxuan, W. X. An, and L. Ruifeng, "Design and implementation of secure file transfer system based on java," in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing: Proceedings of the 15th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2020) 15*. Springer, 2021, pp. 366–375.

[2] M. Lim, "C2cftp: direct and indirect file transfer protocols between clients in client-server architecture," *IEEE Access*, vol. 8, pp. 102 833–102 845, 2020.

[3] N. Kayastha, D. Niyato, P. Wang, and E. Hossain, "Applications, architectures, and protocol design issues for mobile social networks: A survey," *Proceedings of the IEEE*, vol. 99, no. 12, pp. 2130–2158, 2011.

[4] C. Zhao, J. Yang, and T. Ma, "Research on cloud storage technology based on fpga," in *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*. IEEE, 2017, pp. 50–54.

[5] T. Chand and B. Sharma, "Hrcctp: a hybrid reliable and congestion control transport protocol for wireless sensor networks," in *2015 IEEE sensors*. IEEE, 2015, pp. 1–4.

[6] C. Liu, S. Liu, N. Xing, S. Jin, Y. Ji, N. Zhang, and J. Tang, "A load balancing-based real-time traffic redirection method for edge networks," in *International Conference on Simulation Tools and Techniques*. Springer, 2020, pp. 34–43.