Lightweight File Transfer Client Based on the Defined Protocol Stack(STEP): Implementation and Validation via Python Socket Programming

Jingyao Zhang
CST XJTLU
Student ID: 2363985
Suzhou, China
Jingyao.Zhang23@student.xjtlu.edu.cn

Zixi Chen
ICS XJTLU
Student ID: 2362352
Suzhou, China
Zixi.Chen23@student.xjtlu.edu.cn

Kaiming Liu
CST XJTLU
Student ID: 2362259
Suzhou, China
Kaiming.Liu23@student.xjtlu.edu.cn

***Abstract*—This project presents a Python-based client–server file transfer system using the Simple Transfer and Exchange Protocol (STEP). It integrates block-wise transmission and MD5 checksum verification to ensure end-to-end reliability and data integrity. The work proceeds in two phases: (i) refactoring and debugging the provided server code to enable stable protocol-compliant communication, and (ii) developing the client-side modules for authentication, upload planning, and chunked file transmission. The system is evaluated under file sizes (5–25 MB) and concurrent client loads (1–5 users). Results show a near-linear transfer latency growth with file size and up to approximately 40% performance degradation under multi-client contention. The code is publicly available at https://github.com/czx6365/CAN201_assessment1.git**

***Index Terms*—C/S Architecture, Simple Transfer Exchange Protocol (STEP), TCP, file uploading, Computer Network**

I. Introduction

A defining hallmark of contemporary society lies in the exponential advancement of digitalization, networking, and information technologies [1]. With the ubiquity of both personal cloud storage and enterprise-level data management, file uploading and downloading have become indispensable fundamental network services in daily life, serving as the cornerstone of information transmission [2]. The Simple Transfer and Exchange Protocol (STEP) provides a TCP-based lightweight request/response mechanism [3], supporting data storage, file uploads, downloads, and integrity verification through JSON-formatted messages [4]. This project implements a client/server file transfer system compliant with STEP specifications using Python [5], primarily realizing secure login authentication, chunk-based file uploading, and MD5 verification [6]. The research challenges encompass server code debugging, STEP protocol parsing and implementation, and the development of transmission mechanisms that balance efficiency and security.

The practical significance of this work is manifested in its direct deployability in real-world network environments, providing support for cloud storage, content distribution, and backup services. During the testing phase, a dual-virtual-machine architecture was employed to validate upload performance. By varying file sizes and concurrent client scales, the system quantitatively evaluated transmission rate discrepancies and analyzed their underlying causes.

II. Related Work

Reliable large-file transfer in client-server systems has been broadly explored, with emphasis on chunking and integrity verification. Comparative studies of bulk-data protocols such as GridFTP and FDT show that segmented transmission and buffer tuning are essential for sustaining throughput under diverse network conditions [8]. Concurrent transfer frameworks similarly adopt multithreaded upload and end-to-end checksum validation to ensure correctness during unstable or high-latency transmissions [9]. Recent designs like MDTP further optimize performance by dynamically adjusting chunk size and distributing tasks to reduce stragglers [7]. Our system follows these principles through fixed-size chunk transfer, MD5-based integrity checking, and token-based authentication, offering a streamlined yet reliable approach to secure large-file transmission.

III. Design

A. The Architecture Description

As shown in Fig.1. This system employs a client-server (C/S) network architecture to achieve reliable file transmission through a custom STEP protocol .

The **client-side architecture** comprises five fundamental modules: **the User Interface, Connection Management, Authentication, File Processing, and Progress Display**. The **User Interface** enables users to select files for upload and input login credentials, which the system utilizes for subsequent file transfer operations and authorization verification. The **Connection Management module**, encapsulated by the STEPClient class, oversees all network operations and manages the TCP connection with the server. The **Authentication module** is tasked with handling the user login procedure and managing the authentication token. The **File Processing module** reads the designated file in blocks and computes its MD5 checksum for subsequent integrity validation. The

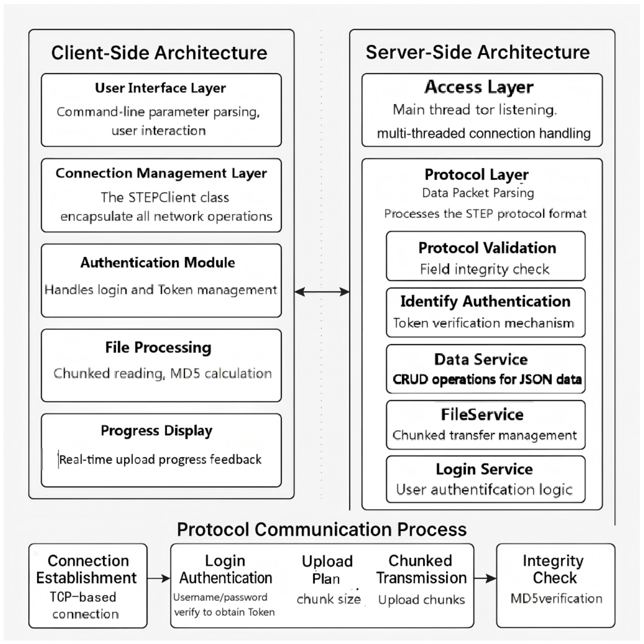

Client-Side Architecture
User Interface Layer
Command-line parameter parsing, user interaction
Connection Management Layer
The STEPClient class encapsulate all network operations
Authentication Module
Handles login and Token management
File Processing
Chunked reading, MD5 calculation
Progress Display
Real-time upload progress feedback
Server-Side Architecture
Access Layer
Main thread tor listening.
multi-threaded connection handling
Protocol Layer
Data Packet Parsing
Processes the STEP protocol format
Protocol Validation
Field integrity check
Identify Authentication
Token verification mechanism
Data Service
CRUD operations for JSON data
FileService
Chunked transfer management
Login Service
User authentifcation logic
Protocol Communication Process
Connection Establishment
TCP-based connection
Login Authentication
Username/password verify to obtain Token
Upload Plan
chunk size
Chunked Transmission
Upload chunks
Integrity Check
MD5verification

Fig. 1: C/S Network Architecture Diagram

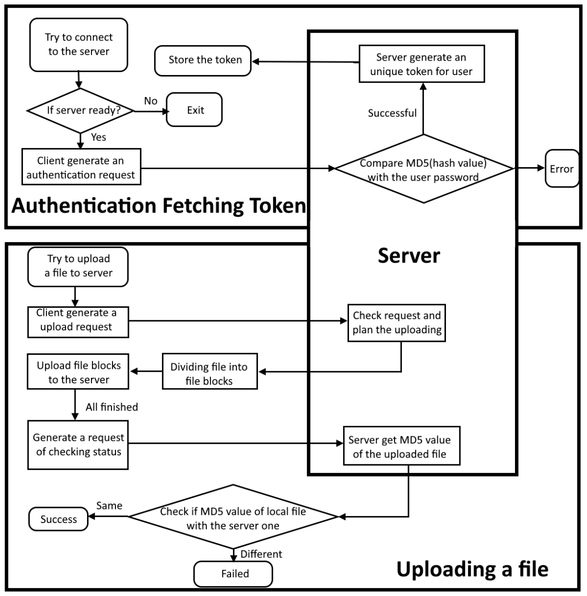

Try to connect to the server
Store the token
Server generate an unique token for user
If server ready?
No
Exit
Successful
Yes
Client generate an authentication request
Compare MD5(hash value) with the user password
Error
Authentication Fetching Token
Server
Try to upload a file to server
Client generate a upload request
Check request and plan the uploading
Upload file blocks to the server
Dividing file into file blocks
All finished
Generate a request of checking status
Server get MD5 value of the uploaded file
Same
Success
Check if MD5 value of local file with the server one
Different
Failed
Uploading a file

Fig. 2: Workflow of key steps

Progress Display module offers real-time feedback on the upload progress .

On the server side, a layered design is adopted to support multi-process concurrent processing. The outermost layer is the **Access Layer**, where a main thread listens on a designated port and accepts incoming connections. Upon receiving a request, the **Protocol Layer** commences operation: it initially parses the incoming data packets according to the STEP protocol format, then validates the protocol by checking the integrity of all packet fields, and finally performs identity authentication by verifying the validity of the token presented in the request. Following successful verification, the request is dispatched to three core service modules: the Data Service handles CRUD (Create, Read, Update, Delete) operations on JSON data; the File Service administers the chunked file transfer process; and the Login Service contains the core logic for user authentication .

The system adheres to a meticulous communication sequence: initial establishment of a TCP connection, followed by successful login authentication and token acquisition. Subsequently, an upload plan is negotiated, leading to the execution of chunked data transmission. The process culminates in an MD5 checksum verification to ensure end-to-end file integrity. This architecture, through its clear hierarchical design and modular division of labor, achieves secure, reliable, and efficient large-file transfer capabilities .

B. The Workflows

As shown in Fig.2.Firstly, the client attempts to establish a connection with the server. Once the server is ready, the client generates **an authentication request**. The server then generates **a unique token** and sends it to the client, who stores the token for subsequent operations.

Next, the client attempts to upload a file to the server. The client creates **an upload request** that includes the file location and the token. The server verifies the request and plans the upload process, returning a plan that includes the size of the transfer blocks, the access key, and the total number of blocks required for the upload. The client divides the file into blocks according to the block size determined by the server. The client transmits each block of data to the server, which uses an MD5 checksum to confirm the integrity and acknowledge the receipt of each block. After each block is uploaded, the client requests a status check of the upload. The server calculates the MD5 checksum of the uploaded file and compares it with the original file. If the MD5 values match, the client receives a success message; if they do not match, the client receives an error message indicating a failed upload.

C. Kernel Pseudo Codes

Algorithm 1 Authorization Pseudo Code

```
Input:user_id, password
Output:token or error
password_md5 ← MD5(password)
request ← create_request(”LOGIN”, ”AUTH”, user_id, password_md5)
send_to_server(request)
response ← wait_for_response()
if response.status == 200 then
    token ← response.token
    return token
else
    error_msg ← response.error_message
    throw Error(”Authorization failed: ” + error_msg)
end
```

Algorithm 2 File Uploading Pseudo Code

```
Input:token, file_path
Output:success or error
file_size ← get_file_size(file_path)
request ← create_request("SAVE", "FILE", token, file_size, key)
send_to_server(request)
upload_plan ← wait_for_response()
access_key ← upload_plan.key
block_size ← upload_plan.block_size
total_blocks ← upload_plan.total_blocks
for block_index == 0 do
    current_size ← min(block_size, file_size - block_index*block_size)
    data_block ← read_file_block(file_path, block_index, current_size)
    block_req ← create_request("UPLOAD","FILE",token, access_key, block_index, data_block)
    send_to_server(block_req)
    response ← wait_for_response()
    if response.status ≠ 200 then
        throw Error("Block upload failed: " + response.error_message)
    end
end
md5_req ← create_request("GET", "FILE", token, access_key)
send_to_server(md5_req)
server_md5 ← wait_for_response().md5
if server_md5 == compute_md5(file_path then
    return success
else
    throw Error("File integrity check failed: MD5 mismatch")
end
```

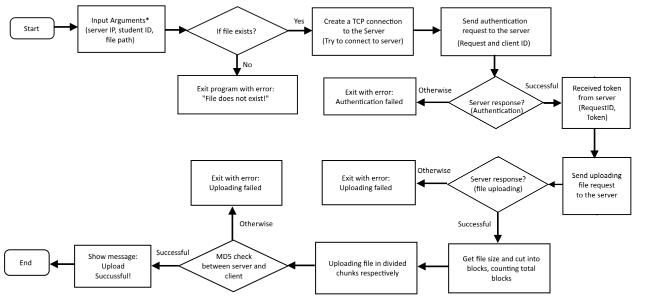

Start
Input Arguments* (server IP, student ID, file path)
If file exists?
Yes
No
Create a TCP connection to the Server (Try to connect to server)
Send authentication request to the server (Request and client ID)
Exit program with error: "File does not exist!"
Exit with error: Authentication failed
Otherwise
Server response? (Authentication)
Successful
Received token from server (RequestID, Token)
Send uploading file request to the server
Server response? (file uploading)
Otherwise
Exit with error: Uploading failed
Successful
Get file size and cut into blocks, counting total blocks
Uploading file in divided chunks respectively
MD5 check between server and client
Otherwise
Exit with error: Uploading failed
Successful
Show message: Upload Succussful!
End

Fig. 3: Program Flow Chart

IV. Implementation

A. Implementation Environment

We implemented the program with a laptop, the host environment is shown as followed:

TABLE I: Development Environment Configuration

Field	Description
Operating System	Microsoft Windows 11
CPU	12th Gen Intel(R) Core(TM) i7-12700H
Memory	16.0 GB
IDE	PyCharm 2023.3
Python Version	3.8

B. Step of Implementation

a) Authorization Function: Within the system, the client firstly establishes a TCP connection with the server. After connection establishment, the client sends a login request containing type 'AUTH', operation 'LOGIN', direction 'REQUEST', username, and the MD5 hash value of the username. The server verifies the integrity of all fields in the login data packet. If any field is missing or incorrect, an error response is returned. Subsequently, the server calculates the MD5 value of the username and compares it with the original MD5 value sent by the client to authenticate credentials. Upon successful authentication, the server generates a unique token containing the username, timestamp, and MD5 hash of a fixed salt value. This token is Base64-encoded and returned to the client. All subsequent requests must carry this token, and the server first validates the token's authenticity to ensure request legitimacy while creating isolated data storage directories for different users.

b) File Upload Function: After obtaining the server token, the client can initiate file processing requests. The server first validates the token's authenticity, after which the client sends a save request containing the token, file identifier, and total file size. The server checks whether the file identifier exists. If it doesn't exist, the server provides an upload plan to the client. The client then divides the file into blocks according to the block size and total number of blocks specified in the plan, transmitting them sequentially. During this process, each block's information is verified to ensure correct transmission. The server employs a two-phase commit mechanism: block data is temporarily stored in a temporary directory with transmission logs recorded. After all blocks are received, a comprehensive MD5 verification is performed. Upon completion of all block uploads, the client calculates the file's MD5 value and compares it with the server's returned value. A successful match indicates that the file upload has completed successfully. This mechanism ensures reliable large file transmission through block-based transfer and integrity verification.

C. Programming Skills

- **Object-Oriented Programming (OOP)** The STEPClient class was created to encapsulate the complexity of network communication. It abstracts operations such as connection establishment, user authentication, and file uploads into discrete methods, providing a clean and simplified interface to external components.
- **Parallel** Independent processing threads are created for each incoming connection via module Threading.Thread, enabling parallel request handling without mutual blocking.

- **Chunked Transfer** This algorithm segments large files into multiple data blocks based on constant MAX_PACKET_SIZE, with each block managed sequentially using precise index information.

D. *Difficulties*

In implementing large file transfer functionality, the most critical challenge lies in ensuring reliability while preventing memory overflow. This challenge is addressed through the design of a chunked transfer mechanism, where large files are segmented into multiple data blocks of a fixed size (e.g., 20 KB) for incremental transmission. Each block is accompanied by precise indexing information to guarantee correct sequencing. On the server side, a two-phase commit strategy is employed: data blocks are initially stored in a temporary directory with transfer logs recorded, and upon reception of all blocks, a comprehensive MD5 checksum verification is performed before the data is moved to permanent storage.

V. Testing and Results

A. *Testing Environment*

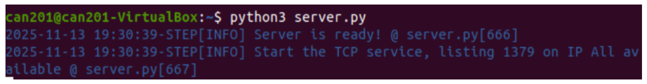
TABLE II: Virtual Machine Configuration

Field	Description
Operating System	Ubuntu (64-bit)
Memory	4096 MB
Software	Virtualbox
Python Version	3.8
Network	NAT Network

During the testing process, we used two virtual machines with the same parameters (configuration as shown in the TABLE II), one as the client and the other as the server.

B. *Testing Steps*

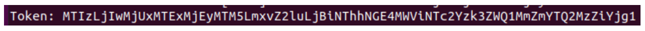
- **Step1:** Enter the location of the server.py file and run the code `python3 server.py` on the Terminal of the virtual machine (VM1), and the result `Server is ready!` is shown as in Fig.4. (The code repair for the server-side application has been completed.)



```
can201@can201-VirtualBox:~$ python3 server.py
2025-11-13 19:30:39-STEP[INFO] Server is ready! @ server.py[666]
2025-11-13 19:30:39-STEP[INFO] Start the TCP service, listing 1379 on IP All av
ailable @ server.py[667]
```

Fig. 4: Result of task 1

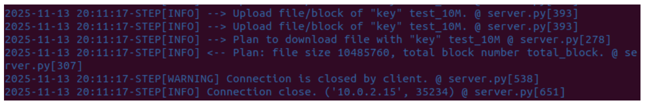
- **Step2:** Enter the command that includes the server IP, user ID and file path and run the client code client.py on VM2 (Command: python3 client.py–server_ip xxx.xxx.xxx–`id xxx`–f`<path to a file>`). After successfully logging into the server, the system will return the token, as shown in Fig.5. And, `id xxx` can be any number you input, and `<path to a file>` should be replaced with the absolute address of the file to be sent on the current virtual machine.



```
Token: MTIzLjIwMjUxMTExMjEyMTM5LmxvZ2luLjBiNThhNGE4MWViNTc2Yzk3ZWQ1MmZmYTQ2MzZiYjg1
```

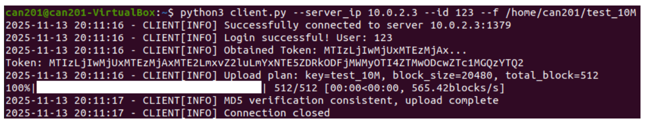
Fig. 5: Get the token

- **Step3:** After that, the machine start to upload the file block by block. When all blocks are uploaded successfully, the machine check the MD5 value to ensure the file is not missing any part during transmission. The result is shown in Fig.6 and Fig.7.



```
2025-11-13 20:11:17-STEP[INFO] --> Upload file/block of "key" test_10M. @ server.py[393]
2025-11-13 20:11:17-STEP[INFO] --> Upload file/block of "key" test_10M. @ server.py[393]
2025-11-13 20:11:17-STEP[INFO] --> Plan to download file with "key" test_10M @ server.py[278]
2025-11-13 20:11:17-STEP[INFO] <-- Plan: file size 10485760, total block number total_block. @ se
rver.py[307]
2025-11-13 20:11:17-STEP[WARNING] Connection is closed by client. @ server.py[538]
2025-11-13 20:11:17-STEP[INFO] Connection close. ('10.0.2.15', 35234) @ server.py[651]
```

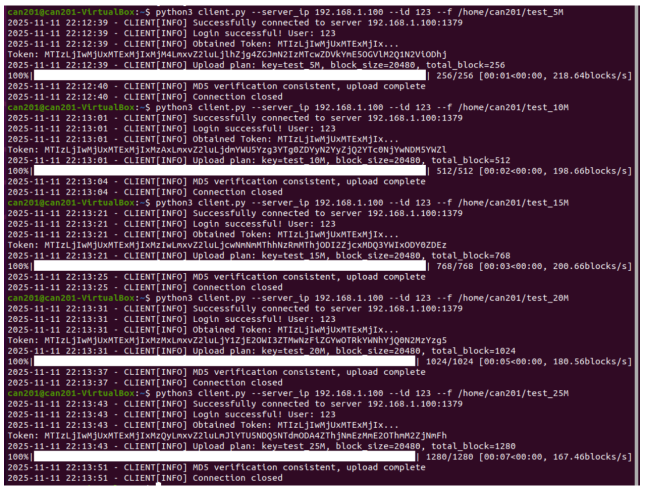
Fig. 6: Server result



```
can201@can201-VirtualBox:~$ python3 client.py --server_ip 10.0.2.3 --id 123 --f /home/can201/test_10M
2025-11-13 20:11:16 - CLIENT[INFO] Successfully connected to server 10.0.2.3:1379
2025-11-13 20:11:16 - CLIENT[INFO] Login successful! User: 123
2025-11-13 20:11:16 - CLIENT[INFO] Obtained Token: MTIzLjIwMjUxMTEzMjAx...
Token: MTIzLjIwMjUxMTEzMjAxMTE2LmxvZ2luLmYxNTE5ZDRkODFjMWMyOTI4ZTMwODcwZTc1MGQzYTQ2
2025-11-13 20:11:16 - CLIENT[INFO] Upload plan: key=test_10M, block_size=20480, total_block=512
100%|                                    | 512/512 [00:00<00:00, 565.42blocks/s]
2025-11-13 20:11:17 - CLIENT[INFO] MD5 verification consistent, upload complete
2025-11-13 20:11:17 - CLIENT[INFO] Connection closed
```

Fig. 7: Client result

- **Step4:** Finally, multiple upload tests were conducted using files of different sizes to compare the differences with the original files and to test the reliability of the system in uploading large files and its stability in transmission over unstable networks. The result is shown in Fig.8.



```
can201@can201-VirtualBox:~$ python3 client.py --server_ip 192.168.1.100 --id 123 --f /home/can201/test_5M
2025-11-11 22:12:39 - CLIENT[INFO] Successfully connected to server 192.168.1.100:1379
2025-11-11 22:12:39 - CLIENT[INFO] Login successful! User: 123
2025-11-11 22:12:39 - CLIENT[INFO] Obtained Token: MTIzLjIwMjUxMTExMjIx...
Token: MTIzLjIwMjUxMTExMjIxMjM4LmxvZ2luLjlhZjg4ZGJmN2IzMTcwZDVkYmE5OGVlM2Q1N2ViODhj
2025-11-11 22:12:39 - CLIENT[INFO] Upload plan: key=test_5M, block_size=20480, total_block=256
100%|                                                            | 256/256 [00:01<00:00, 218.64blocks/s]
2025-11-11 22:12:40 - CLIENT[INFO] MD5 verification consistent, upload complete
2025-11-11 22:12:40 - CLIENT[INFO] Connection closed
can201@can201-VirtualBox:~$ python3 client.py --server_ip 192.168.1.100 --id 123 --f /home/can201/test_10M
2025-11-11 22:13:01 - CLIENT[INFO] Successfully connected to server 192.168.1.100:1379
2025-11-11 22:13:01 - CLIENT[INFO] Login successful! User: 123
2025-11-11 22:13:01 - CLIENT[INFO] Obtained Token: MTIzLjIwMjUxMTExMjIx...
Token: MTIzLjIwMjUxMTExMjIxMzAxLmxvZ2luLjdmYWU5Yzg3YTg0ZDYyN2YyZjQ2YTc0NjYwNDM5YWZl
2025-11-11 22:13:01 - CLIENT[INFO] Upload plan: key=test_10M, block_size=20480, total_block=512
100%|                                                            | 512/512 [00:02<00:00, 198.66blocks/s]
2025-11-11 22:13:04 - CLIENT[INFO] MD5 verification consistent, upload complete
2025-11-11 22:13:04 - CLIENT[INFO] Connection closed
can201@can201-VirtualBox:~$ python3 client.py --server_ip 192.168.1.100 --id 123 --f /home/can201/test_15M
2025-11-11 22:13:21 - CLIENT[INFO] Successfully connected to server 192.168.1.100:1379
2025-11-11 22:13:21 - CLIENT[INFO] Login successful! User: 123
2025-11-11 22:13:21 - CLIENT[INFO] Obtained Token: MTIzLjIwMjUxMTExMjIx...
Token: MTIzLjIwMjUxMTExMjIxMzIwLmxvZ2luLjcwNmNmMThhNzRmMThjODI2ZjcxMDQ3YWIxODY0ZDEz
2025-11-11 22:13:21 - CLIENT[INFO] Upload plan: key=test_15M, block_size=20480, total_block=768
100%|                                                            | 768/768 [00:03<00:00, 200.66blocks/s]
2025-11-11 22:13:25 - CLIENT[INFO] MD5 verification consistent, upload complete
2025-11-11 22:13:25 - CLIENT[INFO] Connection closed
can201@can201-VirtualBox:~$ python3 client.py --server_ip 192.168.1.100 --id 123 --f /home/can201/test_20M
2025-11-11 22:13:31 - CLIENT[INFO] Successfully connected to server 192.168.1.100:1379
2025-11-11 22:13:31 - CLIENT[INFO] Login successful! User: 123
2025-11-11 22:13:31 - CLIENT[INFO] Obtained Token: MTIzLjIwMjUxMTExMjIx...
Token: MTIzLjIwMjUxMTExMjIxMzMxLmxvZ2luLjY1ZjE2OWI3ZTMwNzFiZGYwOTRkYWNhYjQ0N2MzYzg5
2025-11-11 22:13:31 - CLIENT[INFO] Upload plan: key=test_20M, block_size=20480, total_block=1024
100%|                                                          | 1024/1024 [00:05<00:00, 180.56blocks/s]
2025-11-11 22:13:37 - CLIENT[INFO] MD5 verification consistent, upload complete
2025-11-11 22:13:37 - CLIENT[INFO] Connection closed
can201@can201-VirtualBox:~$ python3 client.py --server_ip 192.168.1.100 --id 123 --f /home/can201/test_25M
2025-11-11 22:13:43 - CLIENT[INFO] Successfully connected to server 192.168.1.100:1379
2025-11-11 22:13:43 - CLIENT[INFO] Login successful! User: 123
2025-11-11 22:13:43 - CLIENT[INFO] Obtained Token: MTIzLjIwMjUxMTExMjIx...
Token: MTIzLjIwMjUxMTExMjIxMzQyLmxvZ2luLmJlYTU5NDQ5NTdmODA4ZThjNmEzMmE2OThmM2ZjNmFh
2025-11-11 22:13:43 - CLIENT[INFO] Upload plan: key=test_25M, block_size=20480, total_block=1280
100%|                                                          | 1280/1280 [00:07<00:00, 167.46blocks/s]
2025-11-11 22:13:51 - CLIENT[INFO] MD5 verification consistent, upload complete
2025-11-11 22:13:51 - CLIENT[INFO] Connection closed
```

Fig. 8: Uploading files of different sizes

C. *Testing Result*

Our experiments first evaluated the relationship between file size and upload latency. Five files of sizes 5 MB, 10 MB, 15 MB, 20 MB, and 25 MB were uploaded under identical network conditions. Each test was repeated five times to reduce random fluctuations, and the average latency was recorded. As illustrated in Fig. 9, upload time increases in a near-linear fashion with file size, indicating that the chunked transmission mechanism scales predictably for moderate file sizes.

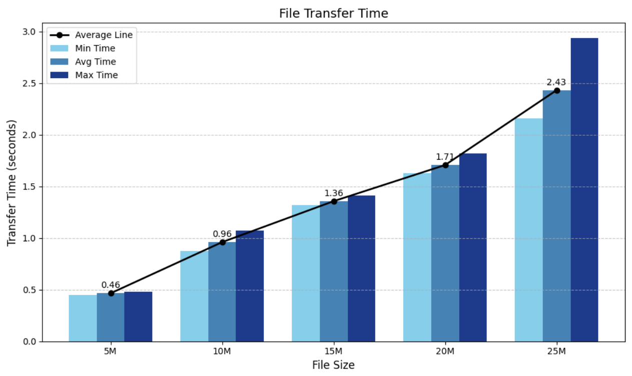


Fig. 9: File transfer time

To further assess concurrency performance, we conducted multi-threaded tests involving 1 to 5 clients uploading the same file simultaneously. The average upload latency per client was measured across five repeated trials. As shown in Fig. 10, latency rises steadily with increasing concurrency, reaching approximately a 40% increase at five clients compared to the single-client baseline. This degradation is likely caused by CPU and I/O contention on the server and the inherent serialization limitations of the C/S architecture. These results highlight that, while the system handles moderate concurrency, performance optimization would be necessary for highly parallel workloads.

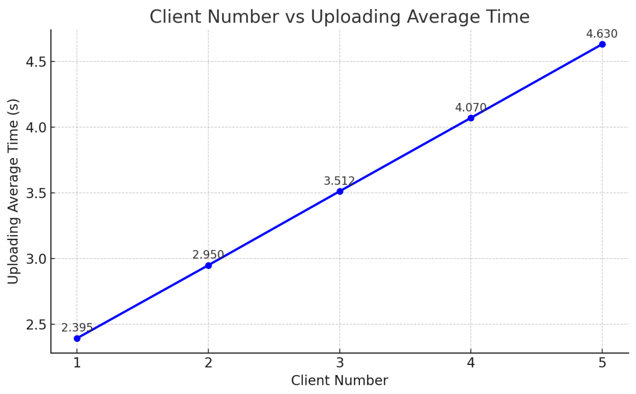


Fig. 10: Multi-client upload time

VI. Conclusion

This paper presents a reliable file transfer system built on the STEP protocol, integrating secure authentication, chunked data transmission, and end-to-end MD5 verification. The modular client–server architecture, together with the two-phase commit mechanism, ensures stable performance during large-file uploads while preventing data loss and guaranteeing transmission integrity. Experimental evaluations conducted under different network environments demonstrate a clear linear relationship between file size and transfer latency.

Furthermore, the results highlight distinct performance differences between hotspot and dormitory networks, with the former exhibiting noticeably higher latency. Despite such variations, the system consistently maintains strong reliability and correctness across all experimental conditions. Overall, the proposed design provides an efficient, lightweight, and practical solution suitable for instructional use as well as small-scale distributed applications requiring dependable file transfer. Future work includes optimizing concurrency handling, adopting adaptive chunk sizing, and enhancing error recovery to further improve efficiency and robustness.

Acknowledgment

In this project, although there were only three of us, we had a clear division of labor and each member fulfilled their duties. Therefore, each of us contributed 33.3%.

References

[1] A. Falade, C. Ayo, K. Akindeji, and A. Adebiyi, "Evolution of Computer Network from Inception to the Internet of Everything: An Overview," in *Proc. 2023 Int. Conf. Sci., Eng. Bus. Sustain. Develop. Goals (SEB-SDG)*, Apr. 2023.

[2] K. Karthik, M. Kaviya, K. Keerthana, and M. Bhavadharani, "A Survey on Sharing Cloud Data Securely with Encrypted Indexing and User Identity Verification," in *Proc. 2023 2nd Int. Conf. Autom., Comput. Renew. Energy*, Dec. 2023.

[3] M. R. Palattella et al., "Standardized Protocol Stack for the Internet of (Important) Things," in IEEE Communications Surveys Tutorials, vol. 15, no. 3, pp. 1389-1406, Third Quarter 2013, doi: 10.1109/SURV.2012.111412.00158.

[4] A. A. Abd El-Aziz and A. Kannan, "JSON encryption," 2014 International Conference on Computer Communication and Informatics, Coimbatore, India, 2014, pp. 1-6, doi: 10.1109/ICCCI.2014.6921719.

[5] *Simple Transfer and Exchange Protocol (STEP) v1.2*. CAN 201 Course Document, 2025.

[6] Z. Yong-Xia and Z. Ge, "MD5 Research," 2010 Second International Conference on Multimedia and Information Technology, Kaifeng, China, 2010, pp. 271-273, doi: 10.1109/MMIT.2010.186.

[7] S. Abdollah, C. Partridge, and S. Shannigrahi, "MDTP – An Adaptive Multi-Source Data Transfer Protocol," in *Proc. Int. Conf. Comput. Commun. Netw. (ICCCN)*, May 2025.

[8] P. Zhang, C. Hu, P. Zhang, J. Chen, and M. Li, "A Novel Cooperative Caching Scheme for Content Delivery Networks," in *Proc. IEEE 34th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2015.

[9] L. Strakova and M. Labaš, "The impact of information technology on the sustainable development," *Int. J. Inf. Technol. Intercult. Stabil.*, vol. 2, no. 1, pp. 48-53, 2019.