CAN 201

Hengqi Liang Chengyang Song Enze Zhou Boyan Li
ID:2254814 *ID:2252824* *ID:2254411* *ID:2254711*

Yatao Ouyang
ID:2254842

***Abstract*—The main purpose of this coursework is to implement a file transfer system based on the Simple Transfer Exchange Protocol (STEP) using Python Socket programming. The project consists of several key components: introduction to the system architecture and challenges, review of related work in client-server systems, detailed design of the authentication and file transfer mechanisms, implementation details with testing results, and conclusions. Our system successfully achieves secure file transfer between client and server through user authentication, token-based session management, and block-based file transmission.**

***Index Terms*—Authentication System, Socket Programming, File Transfer Protocol, Client-Server Architecture, Block-based Transfer, TCP Communication**

I. Introduction

A. Background

The Simple Transfer Exchange Protocol (STEP) is a protocol for file transfer operations between client and server. It uses JSON messages for commands and binary data for file transfers. The protocol requires user authentication and token-based session management, with files being transferred in blocks for efficiency. This project implements both server and client applications using Python Socket programming.

B. Challenge

The main challenges in this project include:

1. Server Development - Debugging and optimizing server code - Implementing error handling - Setting up system logging

2. Authentication System - Implementing secure login mechanism - Managing user sessions with tokens - Testing security features

3. File Transfer System - Developing block-based file transfer - Implementing data verification - Optimizing transfer performance

C. Practice Relevance

This project has practical applications in various scenarios:

1. Data Backup and Recovery - Secure file backup and storage - Fast and reliable data transfer

2. Software Distribution - Efficient software updates delivery - Secure package distribution

3. Cloud Storage Solutions - Personal file storage - File sharing between devices

D. Contributions

There are three major components to this evaluation:

1. Created the fundamental framework diagram for client-server transmission

2. Fixed the syntax issue in the server code

that was published on the LMO ;wrote the client code, tested it, and made the necessary corrections

3. Received the outcome

II. Related Work

In network traffic management and optimization, various methodologies have been proposed to address performance issues and improve traffic redirection through techniques such as load balancing, traffic scheduling, and remote management. This section reviews key studies that provide foundational support for network traffic redirection.

A. Client/Server Remote Control System

Sallow et al. developed a client/server remote management system, allowing administrators to control and monitor remote computers efficiently [1]. This approach reduces the need for on-site visits by enabling direct control via remote servers, which can be seen as a form of network traffic redirection, optimizing resource allocation through centralized control.

B. Socket Programming and Network Communication

Kalita explored the use of socket programming for network communication, specifically how TCP and UDP sockets facilitate data transfer between clients and servers [2]. This technique supports multi-client connections and enables load balancing, providing a robust foundation for traffic redirection and optimizing communication paths across networks.

C. Rigorous Specifications of TCP and UDP

Bishop et al. presented a detailed mathematical specification and experimental validation of TCP and UDP behaviors, which aids developers in controlling and redirecting network traffic more accurately [3]. These specifications ensure that various network protocols handle traffic efficiently, promoting stability and reliability.

D. Host-Network Joint Traffic Management Platform (HONE)

Sun et al. proposed HONE, a platform that integrates host-level and network-level traffic management using software-defined networking (SDN) [4]. This system offloads part of the traffic management to local hosts, reducing overhead and enabling finer control over data transmission. By combining host and network management, HONE optimizes traffic redirection, particularly in data center environments.

E. Performance Analysis of Load Balancing Algorithms

Sharma et al. conducted an analysis of static and dynamic load balancing algorithms. Their study shows that static methods like Round Robin distribute traffic evenly, while dynamic methods, such as the Threshold Algorithm, adjust in real-time to optimize resource utilization and reduce network congestion [5]. These techniques are vital for efficient traffic redirection in distributed systems.

The reviewed studies provide a comprehensive set of methodologies for optimizing network traffic, from remote management systems and socket programming to SDN-based platforms and load balancing strategies. Together, these technologies form a robust framework for traffic redirection and network performance optimization. Future research could explore integrating machine learning and AI to enhance automation in traffic management, addressing the growing complexity of modern networks.

III. Design

As shown in Figure 1, the table shows the code that the team modified in the server part.

A. Description of C/S diagram

The client interface collects login information and displays the upload status, while the file com-

Modify location	revised code	description
import statement	import struct	add struct modules to handle packaging of data
getfile function	getfile_md5	change function name to unify naming
Tcp_listener function	add server_socket.listen(10)	Increase the number of listeners to allow up to ten client connections
Tcp_listener function	add connection_socket, addr = server_socket.accept()	Add the client connection part of the code
Tcp_listener function	logger.info(f'--> New connection from {addr[0]} on {addr[1]}')	Enhanced logging with IP and port information for new connections
tcp_listener(...)	Tcp_Listener(...)	Changing the case of function names for consistency
make_packet	struct.pack()	Added handling of j_len and bin_data
file_process	logger.info(f'<-- Return block {block_index}({len(bin_data)} bytes) of "key" {json_data[FIELD_KEY]}')	Added logging of each download block to track status
STEP_service		Added check for FIELD_DIRECTION to validate request format compliance
data processing		Verify that the uploaded file exists and that the file upload is complete

Fig. 1

ponent segments files according to the server's plan and transfers them securely via TCP. On the server, TCP monitors connections, conducts authentication, and authorizes file access. After receiving an upload request, the server generates an upload plan, processes file chunks, runs integrity checks, logs requests for tracking, and provides the results to the client.

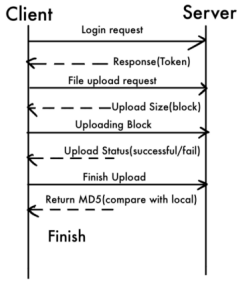

Fig. 2: CS diagram

B. Work flow

The system is based on the TCP STEP protocol and uses a request-response format. It is separated into four parts: user authorization and login, file upload preparation, file block upload, and final file verification.

During the user authorization and login phase, the client creates a TCP connection with the server using the `Login()` method and sends a request including the "LOGIN" operation, direction "REQUEST," type "AUTH," and the username with an MD5-encrypted password. The server uses the `STEP_service()` function to parse and verify the username and password, and if successful, it returns an authorization token for future requests.

The client then calls the `upload_file()` function to send an upload request that includes the "SAVE" operation, token, and file size. The server checks the token and creates an upload plan that includes a unique file identification ("key"), block size ("block_size"), and total number of blocks ("total_block"), and sends it to the client.

The client then breaks the file into chunks based on the upload plan. Before uploading each block, the client connects to the server over TCP and makes a request that includes the "UPLOAD" operation, direction "REQUEST," type "FILE," file "key," block index ("block_index"), token, and block data itself. Each block is verified and briefly stored by the server, and if an error occurs, the client may resend it until the server confirms it.Ultimately, the client provides a completion notice once every block has been uploaded. To make sure the transfer is error-free, the server then determines the combined file's MD5 value and compares it with the MD5 of the local file. This explains how the client and server interact together.

```
FUNCTION login(username, password)
    HASH password using MD5
    CREATE login request
    login request.operation = OP_LOGIN
    login request.username = username
    login request.password = hashed_password
    SEND login request to server using TCP socket
    RESPONSE = RECEIVE server response
    IF RESPONSE.status == 200
        RETURN RESPONSE.token
    ELSE
        PRINT "Fail to login: reason " + RESPONSE.status_msg
        RETURN NULL
END FUNCTION
```

```
FUNCTION upload_file(token, file_path)
    OPEN file in read mode
    CREATE save request with operation=OP_SAVE, token, file size and name
    SEND save request to server
    UPLOAD_PLAN = RECEIVE server response
    IF UPLOAD_PLAN.status != 200 THEN PRINT "Fail Uploading" AND RETURN

    FOR each BLOCK_INDEX in total_blocks from UPLOAD_PLAN
        READ block data from file
        CREATE upload request with operation=OP_UPLOAD, token, key, block_index
        SEND upload request with block data to server
        BLOCK_RESPONSE = RECEIVE server response
        IF BLOCK_RESPONSE.status != 200 THEN PRINT "Block upload failed" AND RETURN
    END FOR

    PRINT "File upload complete"
END FUNCTION
```

Fig. 3: Pseudocode

IV. Implementation Details

The implementation environment consists of an Intel Core i7 CPU, 16gb RAM, and an RTX 3060 GPU running on Windows 11. The development tools used include Pycharm as the IDE

and Python 3.11 as the programming language. Some of the key Python libraries used include sockets for networking, json for data serialization, struct for binary data processing, and hashlib for hash (MD5) To outline the implementation steps, I created a flowchart detailing the login and file upload process. As follows:

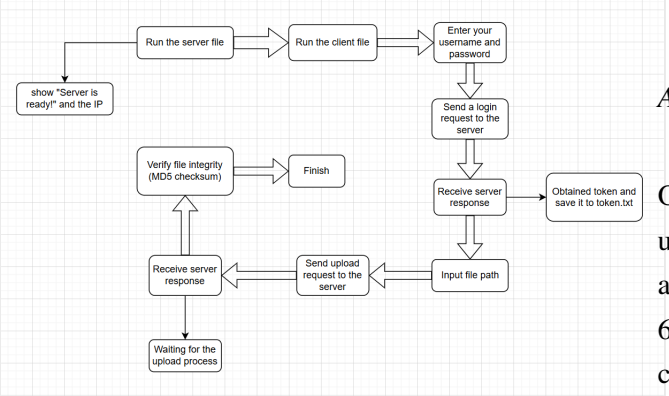

Fig. 4: Program flow charts

The entire program flow begins with the user entering their credentials. The client then sends a login request to the server and, if successful, saves the token and begins the file upload process. The client sends a request to save the file and then uploads the file as blocks while verifying that each block was successful. Finally, verify the MD5 checksum to ensure the integrity of the uploaded file. In terms of programming skills, we used object-oriented programming (OOP) to build applications in a modular manner and socket programming for client-server communication. It also implements good error handling to effectively manage connection problems and invalid inputs. For the authorization function, we use MD5 hash algorithm to ensure the security of the password and send the login request to the server. A valid token response from the server indicates successful login. The file upload function checks the existence and permissions of files, prepares files to be uploaded in blocks, and verifies whether the upload of each block is successful. After the upload is complete, the MD5 checksum of the uploaded file is compared with the server checksum to ensure file accuracy. Of course, we also encountered some difficulties, for example, in the upload process encountered a file not found error, and finally found that the absolute path to solve the problem, and can not be double quotes, that is, can not copy the address directly from the folder.

V. Testing and Results

A. Testing Environment

We ran and tested the code on both systems. On Windows, PyCharm (Python 3.12.0) was used for development and testing; It also runs on an Ubuntu virtual machine, as shown in Figure 6. Verify the compatibility and stability of the code under different operating systems.

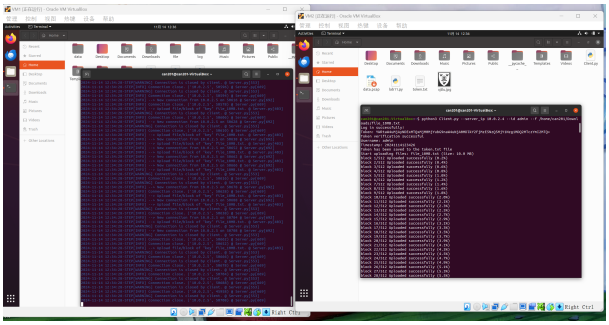

Fig. 5: Virtual machine running

B. Testing Steps

Task 1: Server Code Debug

We resolved the bug in the server-side file and ran the code. The message “Server is ready” appeared in the terminal, indicating that the server-side errors have been corrected. As shown in Figure 7, the server is listening on port 1379. The mandatory port number required by the STEP protocol is 1379. At this point, Task 1 has been completed.

Fig. 6

Task 2: Server Authorization

The next step is to write the client code. Once the code is complete, run the code, enter the username and password as required, and upon successful login, you can get a token in Figure 8, proving that Task 2 has been completed.

Fig. 7

Task 3: File Uploading

The last step is to send the file from the client to the server side, where we upload a file named `file_10MB_.txt` through the client to the server. The client is required to upload files block by block and verify whether each block has been successfully uploaded in real time. Once complete, the server authenticates and confirms the file status using MD5 and compares it to the client to ensure the integrity of the successful upload. figure 9, figure 10 and figure 11 show the upload progress and update the status block by block. This proves that Task 3 has been completed.

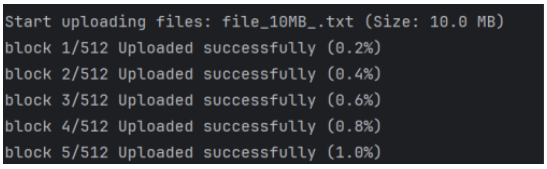

Fig. 8

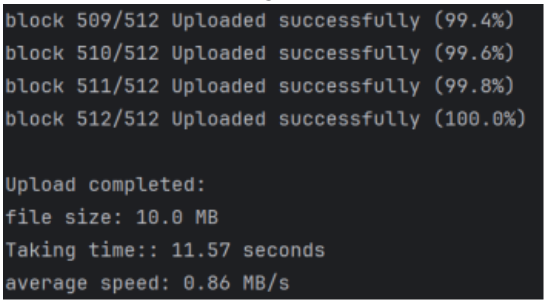

Fig. 9

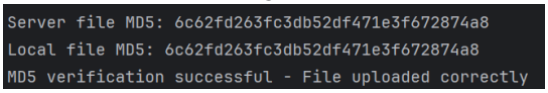

Fig. 10

C. Testing results

We uploaded five files of different sizes respectively, and made the following Figure 12 according to the upload time of different files, and analyzed their average performance.

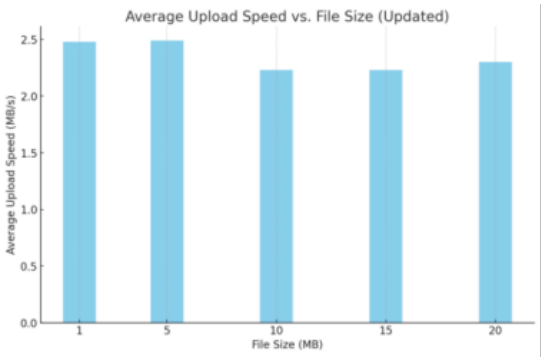

Fig. 11: Performance Analysis

1. Stability of Average Upload Speed: The bar chart shows that the average upload speed remains relatively stable across different file sizes, generally staying within the range of 2.2 to 2.5 MB/s. The minor height differences between the bars indicate consistent upload performance.

2. Slight Variation: The upload speed for smaller files (1 MB and 5 MB) is slightly higher, around 2.5 MB/s, while for larger files (10 MB, 15 MB, and 20 MB), it slightly drops to approximately 2.3 MB/s. This small decrease, visible in the bar heights, might result from network resource usage or the system's reduced efficiency when handling larger files.

VI. Conclusion

We collaborated to debug server-side Python code,build a client app that is reliable enough, designed the network architecture for the entire client-server (C/S) model, and developed the corresponding code. Key features include authentication, file segmentation, block-based transfer, and MD5-based integrity checks, providing stable file uploads across file sizes with real-time block upload monitoring. The proposed system, tested for both Windows and

Ubuntu, was confirmed as cross-platform stable. Future improvements include:

(1) Enhanced Security: Our focus is on end-to-end data transfer encryption, which relies on AES or RSA in lieu of simple MD5 hashing for storing passwords.

(2) Optimized File Transfer Speed: To reduce slowdowns with larger files,we are adopting adaptive block sizes that depend on network conditions.

REFERENCES

[1] A. B. Sallow, H. I. Dino, Z. S. Ageed, M. R. Mahmood, and M. B. Abdulrazaq, "Client/Server Remote Control Administration System: Design and Implementation," *International Journal of Multidisciplinary Research and Publications (IJMRAP)*, vol. 3, no. 2, pp. 5-11, 2020.

[2] L. Kalita, "Socket Programming," *International Journal of Computer Science and Information Technologies (IJCSIT)*, vol. 5, no. 3, pp. 4802-4807, 2014.

[3] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "TCP, UDP, and Sockets: Rigorous and Experimentally-Validated Behavioural Specification" *University of Cambridge, Computer Laboratory Technical Report Number 624*, Cambridge, UK, 2005.

[4] P. Sun, M. Yu, M. J. Freedman, J. Rexford, and D. Walker, "HONE: Joint Host-Network Traffic Management in Software-Defined Networks," *Journal of Network and Systems Management*, vol. 22, no. 3, pp. 331-356, 2014.

[5] S. Sharma, S. Singh, and M. Sharma, "Performance Analysis of Load Balancing Algorithms," *World Academy of Science, Engineering and Technology*, vol. 38, pp. 269-275, 2008.