The client side realization of Web disk system based on STEP

1st Fanyun Xu
ID: 2033366

2nd Xie Xiang
ID: 2035100

3rd Yanjie Xu
ID: 2034363

***Abstract*—This project plans to design a client-side program based on the STEP (Simple Transfer and Exchange Protocol), TCPTransmission Control Protocol) and given server-side program to do file uploading. Furthermore, after the realization of the primary function of file upload, the code has been optimized, such as using concurrent programming, thread pool and other technologies to improve the efficiency of the file transfer.**

***Index Terms*—threading, mutex, TCP, STEP**

I. INTRODUCTION

This task follows the TCP-based STEP to upload files from the client to the server. The problems that need to be solved are as follows. Firstly is to complete the authorization of the client during the login operation. Second, the operation of deleting the existing file has to be done. During the upload process, the file existing in the client is uploaded to the server following the requirements of it. This program can be applied to the web disk system based on C/S (Client/Server) network architecture. For this coursework, we corrected the existing bugs in the server, completed the design and implementation of the client, and successfully uploaded the files between the client and the server.

II. RELATED WORK

Related work: In order to solve the problems related to file processing and network transmission, files are packaged through the STEP of the application layer as JSON files for upload. JSON document is now very popular and has become a common way of data representation because it can process large amounts of data without explicit data schema [1]. It makes better use of thread synchronization, allowing two independent packages to be connected to each other without having to worry about the problem of sticking together. Similar to the most widely used HTTP (Hyper Text Transfer Protocol) on the Internet, STEP is also an application-layer protocol based on TCP [2], and they both specify that the server uses a specific TCP port to open a socket. HTTP is 80 and STEP is 1379. The reason for using TCP connection is that it is a secure and reliable transport protocol, which ensures the stability and security of the connection through three handshakes to establish the connection and four waves to close the connection. In addition, TCP congestion control reduces the round trip time (RTT) and provides better bandwidth sharing [3]. The final effect is similar to the network cloud disk, which optimizes its transmission through TCP/IP protocol and is stored in the server in a safe and efficient way to complete an online storage function [4].

III. DESIGN

A. C/S network architecture

The client and server architectures are shown in Figure 1. The client will send the user request to the server for processing. The server will give different responses according to the received data. For example, the server will return the request error information if the 'LOGIN' and 'SAVE' requests are not made in advance and the client directly uploads. The specific workflow is as follows: First, the user needs to provide the IP address of the server, the username, and the absolute path of the upload file when executing the client program. The client performs 'LOGIN,' 'DELETE,' 'SAVE,' and 'UPLOAD' operations according to the information provided by the user. It closes after four operations and waits for the next transmission.

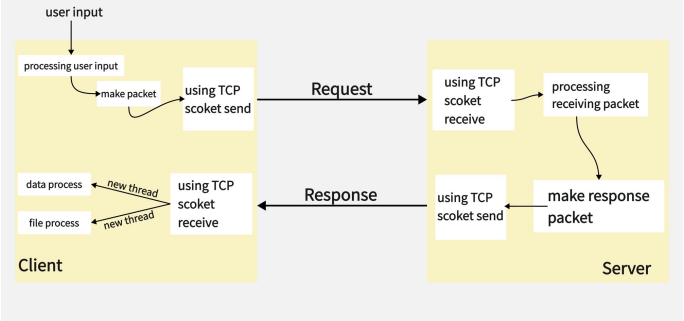

Fig. 1. C/S architecture

B. The workflow

The first step of the client side is to apply for the 'LOGIN' operation. The client processes the user input, packages it and sends it to the server. The server sends a response containing the token to indicate that the user has logged in successfully and can perform subsequent operations. After the user logs in successfully, the client will initiate the delete operation to delete the file with the same key from the server to ensure normal uploading. The client then informs the server of an 'UPLOAD' operation and the absolute path of the file to be uploaded. After the server returns the upload plan, execute the upload and add the absolute path of the file to be uploaded,

and display the upload progress. After the file is successfully uploaded, the server sends a response message to the client indicating the successful operation and the md5 value of the file. The client matches the md5 value of the uploaded file with the md5 value of the local file and prints the result on the terminal. The task that the client needs to complete is to process the user input, such as the user name, and change the user name to the password with the MD5 method. Then the client sends the request to the server, including the operation type, operation direction, username, password, etc., and converts them into JSON format for sending. The server receives the request, sends a response to the client, the client proceeds with the next operation, and then sends the request to the server. For example, the server receives a save request and returns "This is the upload plan" on the terminal. After receiving the response, the client input the operation name plus the absolute path of the object to be operated. According to the size and number of blocks stipulated by the server, the client-side send to the server through the TCP. The server will receive the file according to the block. The above is the preliminary operation of the client and server. Figure 2 clearly demonstrates C/S communication.

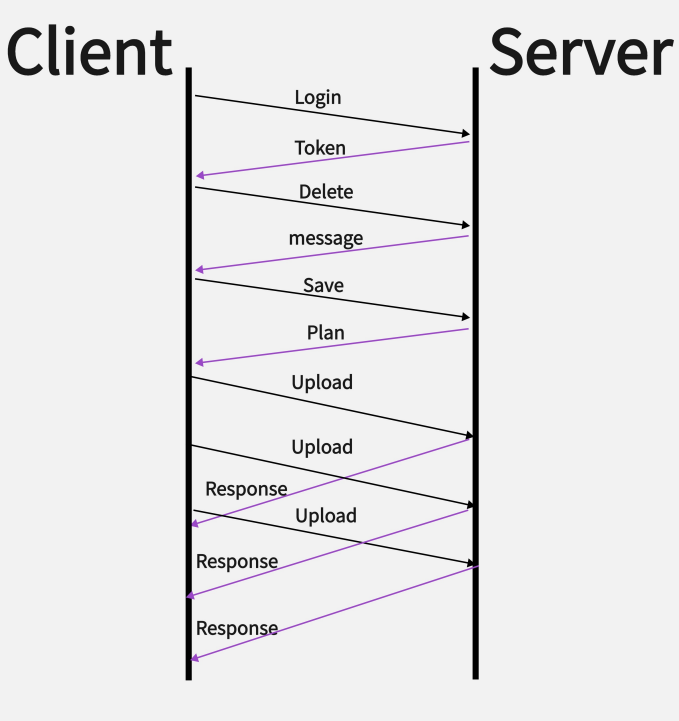

Fig. 2. C/S communication

C. Algorithm

The kernel pseudo codes of the authorization and file uploading as follows.

Algorithm 1 step_service(connection_socket,addr)

```
if request_operation ← OP_LOGIN then
    check the information in the request packet
    if all the information is correct then
        make_response_packet
    end if
end if
if request_operation ← OP_UPLOAD then
    check the information in the request packet
    if all the information is correct then
        write the block into the file
        and
        inform the client the block successfully being uploaded
    end if
end if
```

IV. Implementation

A. The host environment and the development tools

The development environment for this project is shown in Figure 3.

CPU: AMD Ryzen 9 5900X 12-Core Processor

Memory: G.skill 4000Mhz 16G x 2

Disk: Samsung SSD 980 1TB

Mother board: MSI B550i Gaming EDGE WIFI

operating system: Windows11 22H2

Fig. 3. Environment of project

The IDE used is PyCharm and the programming language is Python Version 3. The python libraries that client-side uses include socket library to set up network connections and transfer files, hashlib library to encrypt passwords and the threading library for multi-threading and thread synchronization. The struct library converts python values to strings, or streams of characters to python values. The time library is used to time. The os library does file input/output. The tqdm library displays a progress bar. The Pool in the Multiprocessing.dummy library is used to manage the thread. The json library serializes objects.

B. Steps of implementation

We design the client architecture according to the server side. The client_menu is first designed to read and split the execution commands, where vital information such as the server's IP and user name can be retrieved. Then create a socket in the client to establish a TCP connection with the server. Design a tcp_connector to accomplish this step,

connecting the IP address of the server and a specific port number. Then, prepare the parameters to be sent, encapsulating them in make_request_packet. Furthermore, to begin sending, we design the method step_client to do this step, using a loop to perform each of the four operations in turn and creating multiple threads to assist the sending. After sending the request, we need to receive the response from the server, so we designed a TCP_receive method to receive it, and designed a get_tcp_package method to split the obtained JSON data and binary data, and used file_process to process the parameters of the received JSON part. Finally, get_file_md5 is designed to compare and check the integrity of server files.

C. Programming skills

Object-oriented programming is applied to the client side in this task to make the code modular, with higher re-usability and more manageable system maintenance [5]. The global variable to the client is the object's attributes, and below the attributes are the client's methods. The schematic diagram is shown in Figure 4.

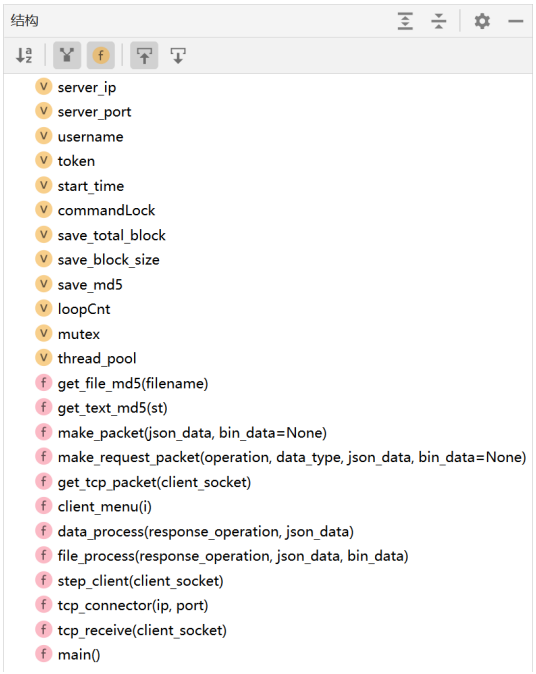

Fig. 4. Client's methods

The thread-based parallelism method is used in the file transfer steps. Two sub-threads, client_thread and receive_thread, are created when the main thread is executed for sending and receiving. This action will separate receiving and sending. So that the sending does not have to wait for receiving, and the execution efficiency of the client is improved. In the sending process, the mutex is set, and the four operations of 'LOGIN,' 'DELETE,' 'SAVE' and 'UPLOAD' are executed successively. The first three operations need to obtain the lock when sending the request and release the lock after receiving the response. This ensures that the necessary information is captured before each operation. The schematic diagram is shown in Figure 5.

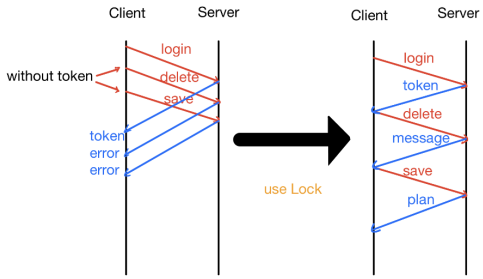

Fig. 5. Use Lock for thread synchronization

The 'UPLOAD' operation uses the packet switching method. Each time when the 'UPLOAD' operation is executed, a thread will be started separately. Furthermore, each block will be allocated a thread, so before uploading the next block, there is no need to wait for the previous block to finish uploading. Different blocks can be transmitted simultaneously, thus reducing the cost caused by waiting for network delay and improving transmission efficiency. The schematic diagram is shown in Figure 6.

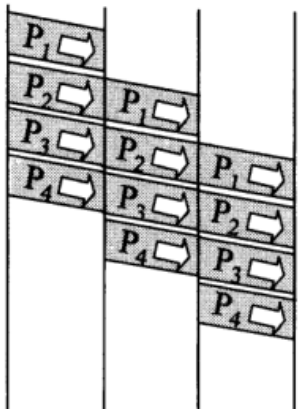

Fig. 6. Packet Switching

In addition, the client side sets up the thread pool to limit the maximum number of threads. Thread pools can separate threads from tasks and improve thread reuse. Avoid creating multiple threads which may consume system resources and reduce system stability.

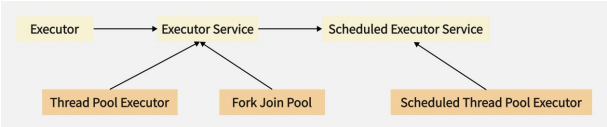

Fig. 7. Thread Pool

D. actual implementation

The detailed operations of authorization and file uploading are described below. The authorization function is implemented in the following steps:

(1) When operation is 'LOGIN', data_type is set to 'AUTH', token is 'NONE', and together with other JSON data are packaged and sent to the server.

(2) The server checks the data sent by the client and returns an error message if the information does not conform to the STEP format. For example, if the operation is 'LOGIN,' but data_type is not set to 'AUTH,' the server returns 'Type of LOGIN has to be AUTH.'

(3) After the check is correct, the server will generate a special token value for the current user to allow him to perform subsequent operations on the server.

(4) After receiving the response packet from the server, the token value returned is assigned to the global variable 'token' as part of the subsequent request packet.

To implement file upload, perform the following steps :

(1) The client sends a request packet to the server for the 'SAVE' operation.

(2) After receiving the request packet, the server checks whether the token value matches. If an error occurs, the server returns status code 403 and error information.

(3) If the token value is correct, enter file_process to process the request. Check the key value first. If the key value already exists, error message 402 is returned. If the request information does not contain information about the file size, the 402 error message is also returned.

(4) If the above check is correct, create a new file on the server to receive the file sent by the client, and use the log to record the upload. Further, the server sends the plan of the upload to the client, including the key, the file size, the total number of transferred blocks, and the size of the individual transferred blocks.

(5) After receiving the upload plan from the server, the client enters the 'Upload' operation. The client divides the file into multiple blocks for transmission according to the upload plan of the server.

(6) After receiving the request packet from the client, the server checks it. Each request package must declare key and block_index, which are checked by the server. After the check, the server writes the information in the block to the file and returns the information that the block has been successfully uploaded.

(7) After all blocks are uploaded, the server calculates and sends the md5 value of the file to the client. The client compares the md5 values of the uploaded file and the local file to check whether the md5 values are the same. And print the information to the console.

E. Difficulties

During the implementation of the project, the difficulty encountered in the implementation is that in the client process, the TCP receiving thread initially uses the dead loop of while(True), that is, the thread is always in the state of waiting for receiving. If the receiving is blocked, the thread cannot end result in the whole client process cannot end. So we later changed the code to get rid of the dead loop and stop the thread automatically after a finite number of executions. This is done by setting a counter at the receiving side. The value of the counter is 'loopCnt+total_block'. The receive_thread stops automatically when the counter is reduced to 0. It prevents the receive_thread from being blocked for waiting the response from the server and cannot be stopped externally.

Algorithm 2 tcp_client(client_socket)

```
loopCnt = 3
while loopCnt >0 do
  loopCnt-=1
  get_tcp_packet()
  if operation == "SAVE" then
    return loopCnt += total_block
  end if
end while
```

V. TESTING AND RESULT

A. Testing

The operation system of the computer is Windows10, with 16G memory and i5-10210U CPU. The test environment is two virtual machines equipped with Ubuntu operating system. The server uses 2 cores processors and the memory size is 2048MB. The client uses a 1 core processor and 1024MB of memory.

1) The first step is correct existing syntax errors on the server so that the server can run properly. print(" Server is ready ") is added to the code to inform the user that the server has run successfully. As shown in the Figure 8, after the server is running properly, the console displays "Server is ready" and is listening to port 1379.

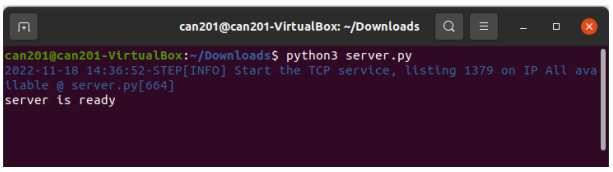

Fig. 8. Server is ready

2) The second step is run the following command to start the client: client.py -server_ip xxx.xxx.xxx -id xxxxxx -f ¡path to a file¿. The client side sends a 'LOGIN' request to the server. After receiving the response from the server, the client side prints the token on the console and performs subsequent operations. The schematic diagrams are as follows.

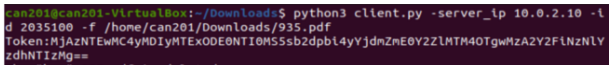

Fig. 9. Print token

3) The last step is as follows. Task3 requests the upload from the client, performs the upload according to the returned upload plan, and checks the MD5 value returned by the server to examine the integrity of the uploaded file. In the meantime, to test whether the above two tasks are complete, the following information will be displayed. First, if the user successfully logs in, the server will give the user a token value. The token value will be printed on the terminal. In order to ensure the success of subsequent uploads, the server will delete local files with the same key value. As shown in The following figure, "The" key" 935.pdf is deleted" indicates that the file has been successfully deleted. After the client requests the 'SAVE' operation, the server will return the corresponding upload plan according to the information of the file to be uploaded. "This is the upload plan" is displayed successfully on the terminal. Next, the upload operation will be performed, and the terminal will display the upload progress, including the percentage, the number of blocks uploaded. After the file is successfully uploaded, the client checks the md5 value transmitted by the server. If the file is correctly uploaded, "md5 is checked – OK" will be printed on the terminal. The time taken to complete the transmission is shown in milliseconds (ms). The schematic diagram is as follows.

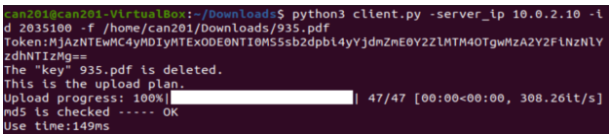

Fig. 10. Client running results

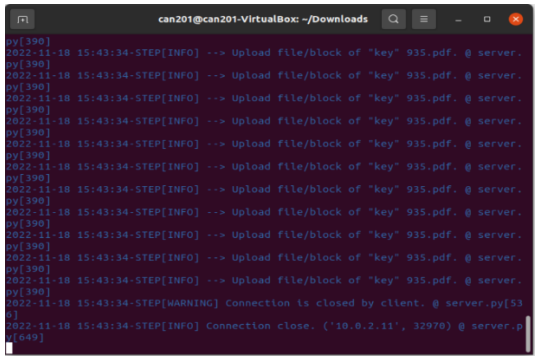

Fig. 11. Server running results

B. Results

In order to test the upload efficiency of the program, we uploaded 132KB, 223KB, 304KB, 458KB, 528KB, 629KB, 708KB, 849KB and 935KB files respectively. Since the operating system may store files that are frequently read and written temporarily in memory, We uploaded each file 11 times, and recorded the average upload time between the 2nd and 11th times of each file upload, so as to reduce data bias caused by disk IO and memory access. The corresponding line graph is shown below.

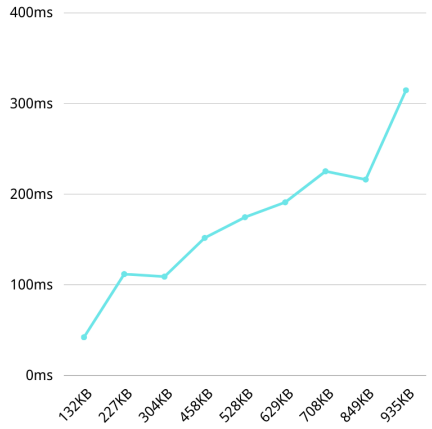

Fig. 12. Results

As can be seen from the Figure 12, when the size of the uploaded file is less than 1mb, the upload time and the file size show a linear relationship. Below is an analysis of the system's bottlenecks. The most important bottleneck is that the max_block_size set by the server is small, resulting in too many times of splitting. During each encapsulation, the block must carry corresponding JSON files, and the final total file size transferred is Total_file_size = total_block * json_file_size + file_size. Also, more splits will result in more disk input/output, resulting in slower program execution. In addition, CPU performance will affect the transfer speed, and network latency will also be a factor.

VI. CONCLUSION

In this project, we completed the design, implementation, and testing of the client according to the requirements. We use C/S architecture to design the communication between the client and server. The python programming language is used to implement the client code, and the use of concurrency and thread pool to improve the transmission efficiency. In the test phase, two virtual machines are used to simulate the communication between the server and the client to upload files, and the uploading efficiency is evaluated. In future improvements, we can address the transfer bottleneck of the system by having the server increase the fragment size appropriately according to the size of the transferred file.

REFERENCES

[1] Zhaogeng Li, Jun Bi, and Sen Wang. Http-ccn gateway: Adapting http protocol to content centric network. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–2, 2013.

[2] M. Chandrasekaran, M. Kalpana, and R.S.D. Wahida Banu. Interaction between polynomial congestion control algorithms mimd-poly and pipd-poly and other tcp variants in tcp/ip networks. In *2006 IFIP International Conference on Wireless and Optical Communications Networks*, pages 5 pp.–5, 2006.

[3] Bing Zhou, Jiangtao Wen, Jingyuan Wang, and Zixuan Zou. Http streaming over an improved tcp congestion control algorithm. In *2011 International Conference on Multimedia Technology*, pages 3093–3096, 2011.

[4] D.A. Rodríguez-Silva, L. Adkinson-Orellana, F.J. Gonz'lez-Castaño, I. Armiño-Franco, and D. Gonz'lez-Martínez. Video surveillance based on cloud storage. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 991–992, 2012.

[5] Hussam Hourani, Hiba Wasmi, and Thamer Alrawashdeh. A code complexity model of object oriented programming (oop). In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, pages 560–564, 2019.