Xi'an Jiaotong-Liverpool University
西交利物浦大学

University of Xi'an Jiaotong-Liverpool

School of Advanced Technology

Networking Project
Simple Transfer and Exchange Protocol (STEP)

Kaijie Lai 2034675

Zhen Ma 2034590

Ruochen Qi 2035507

November 20, 2022

Networking Project
Simple Transfer and Exchange Protocol (STEP)

Zhen Ma
School of advanced technology
Xi'an jiaotong-Liverpool University
Suzhou, China
Zhen.Ma20@student.xjtlu.edu.cn

Ruochen Qi
School of advanced technology
Xi'an jiaotong-Liverpool University
Suzhou, China
Ruochen.Qi20@student.xjtlu.edu.cn

Kaijie Lai
School of advanced technology
Xi'an jiaotong-Liverpool University
Suzhou, China
Kaijie.Lai20@student.xjtlu.edu.cn

***Abstract*—File transfer is one of the basic functions of computer networks and is of great importance in communication. In order to better investigate the file transfer process from client to server, we used a client/server protocol based on the Transmission Control Protocol. The result is the implementation of a basic file transfer between two machines. A project design that provides an efficient and stable file transfer.**

***Index Terms*—file transmission, TCP, STEP, uploading, Python**

I. Introduction

As the internet continues to grow in popularity, it continues to be an important part of people's lives. Uploading and downloading files is one of the most common ways of using the internet. These types of operations often require the use of communication protocols. The Transmission Control Protocol (TCP) is one of the most common communication protocols and is widely used for file transfers because of its connection-oriented, reliable transport, and congestion control characteristics. In this project, the group will use Python socket programming through the Simple Transfer and Exchange Protocol (STEP) based on TCP to achieve the following three goals [1] [2].

- Fixing problems with server code and running
- Obtaining authorization from the server-side
- Uploading a file to the server using the client

The project can be used in practice for stable file transfers between multiple devices, especially those based on the STEP protocol. As a contribution to the project, our group implemented a client-side login on the server first, then a save and upload operation based on the TOKEN feedback from the server. This resulted in the uploading of files from the client to the server in blocks. The report will begin with a description of the related work, followed by the design and implementation of the project, and finally the results of the testing.

II. Related work

The problem of processing and transferring files across the Internet has been one of the key research questions in the academic community for the past few decades. Many universities and companies are advancing their research and publishing results on the subject. This has provided a wealth of information for the project to draw on.

For the focus of the task, which is related to file transfers, Nossenson [3] suggests that the three main factors affecting transfer duration are file size, network conditions and server load. In contrast, the transfer duration distribution is very similar when the same client makes a transfer to the server, regardless of its size. In another paper, Yan [4] proposed a more mature TCP-based technique for transferring files within a LAN, with better transfer performance and a more accurate measure of the time required to transfer the entire file. In addition, in terms of multi-threading, Zhang's team has proposed ways to effectively improve the real-time performance of file transfers in their multi-threaded file push system [5].

The above related work provides an analysis of the factors influencing transmission rates and some of the code design techniques for the client side of the project. This has helped to guide the design, implementation and testing of our project.

III. Design

This section will describe the entire process of designing the solution including a C/S network architecture and a work flow in progressing.

A. Network Architecture of Design

Figure 1 illustrates the network architecture between a client and a server during a file upload.According to the STEP protocol, the server is opened based on the client's ip and given port 1379, after which the server orders Thread 1 to work and sets it in a listening state waiting for the client's request.

All operations mentioned in the STEP protocol require TOKEN verification before they can be performed, and the way to obtain TOKEN is to "log in to the server system" to obtain the TOKEN of the corresponding client. The server will hear the information and generate the corresponding $\langle token \rangle$ to return to the client. After the token is obtained, the client performs a SAVE operation and sends the $\langle key \rangle$ and $\langle size \rangle$ of the file. After receiving the SAVE request from the client, the server processes the request and sends back information including $\langle key \rangle$, $\langle block\_size \rangle$ and $\langle total\_size \rangle$ to the client for UPLOAD operation. After this, the client will

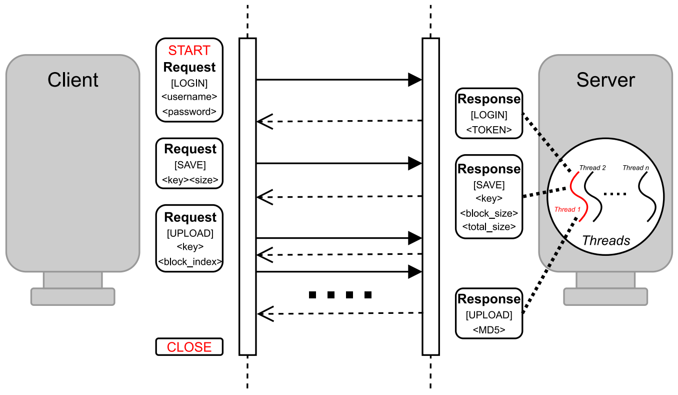

Fig. 1. C/S Network Architecture

make a new request and send message which includes $\langle key \rangle$ and $\langle block\_index \rangle$ in the form of a block for UPLOAD. After several transfers the server will return the $\langle MD5 \rangle$ of the file after the complete transfer of the file content for proofreading and closing the Thread 1 listening activity.

This is a complete Upload activity which performs in this architecture and more information of the specific steps will be explained in the next part.

B. Work-flow Solution of Design

In this section the design of our work-flow solution will be explained in detail. Figure 2 is a Machine Diagram which shows a more detailed explanation of how to Upload a file to the server from above. Here is a description of the key steps involved:

- The password is generated on the client side and sent to the server with the message before the first Request Login activity takes place.
- The token will be generated in the server in an encrypted form after receiving the user name and password from the client. Then it will be fetched by client with the Response Login activity and added in JSON data part. After that, all operations except Login are subject to a TOKEN check after the Request is sent to the server. Only messages which pass the Authorization will be judged to be reasonable and will proceed to the next step.
- A key will be determined by the transfer file in client and added to the JSON data part on the second Request, before the SAVE activity starts. Similarly, it will be transferred to the server with the block in the Request Upload.
- The MD5 will be generated after the server has received all blocks and is no longer receiving blocks from the client, and be sent back to the client for subsequent manual checks if required.

C. Algorithm

The algorithms of authorization and file uploading are as follows:

Algorithm 1 Algorithm of Authorization

```
Input:  FIELD_USERNAME ← __$id$
Client:
    $password$ ← hashlib.md5(__$id$).hexdigest().lower()
    $message$ ← make_request_packet()
    send_message($Request$)
Server:
    if $password$ is right then
        generate $token$
        $message$ ← make_response_packet()
        send_message($Response$)
        output“Login sucessefull”
    else
        $message$ ← make_response_packet()
        send_message($Response$)
        output “Password error for login.”

Output: Login Status
```

Algorithm 2 Algorithm of Uploading

```
Input:  FIELD_TOKEN ← $token$
        key ← __$file$
Client:
    $message$ ← make_request_packet()
    send_message($Request : Save$)
Server:
    save $file$ in tmp
    generate and send back $rval$
    $message$ ← make_response_packet()
    send_message($Response : Save$)
Client:
    get $rval$ packet
    while True do
        read $file\_data$
        if $file\_data$ = 0 then
            break
        $message$ ← make_request_packet()
        send_message($Request : Upload$)
        block_index ++
    Socket close
Server:
    $block\_index$ ← json_data[FIELD_BLOCK_INDEX]
    if $block\_index$ is legal then
        generate and send back new $rval$
    if all block is transferred then
        generate and send back $MD5$
    $message$ ← make_response_packet()
    send_message($Response : Upload$)
Output: Uploading Status
```

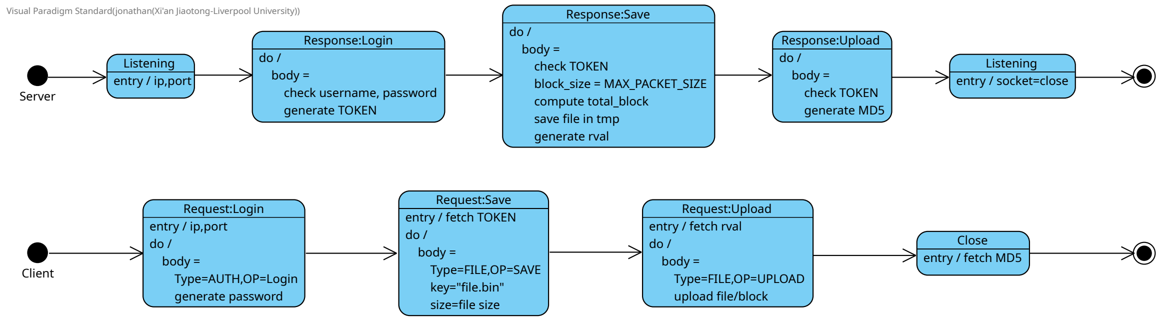

Fig. 2. State Machine Diagram

IV. Implementation

A. Host Environment

The following environment are used for the host implementation of this procedure:

TABLE I
Host Implementation Environment

CPU	AMD Ryzen 9 5900HX
OS	Arch Linux x86_64
Linux Kernel	6.0.8
Memory	16G
Python	3.10.8
IDE	Pycharm 2022.2

B. Programming Skills

In order to implement client-side functionality more efficiently and elegantly, Object-Oriented Programming, or OOP , was used in writing the program, shown here in the design of Class TCP_Client:

TABLE II
Class of TCP_Client

TCP_Client
- client_socket: *socket.socket* - id: *str* - port: *int* - ip: *str* - file: *str*
- __init_(self, parser) - __get_packet(self) - __make_packet(json_data, bin_data=None) - __make_request_packet(data_type, operation, json_data, bin_data=None) + comm(self)

- **__get_packet(self):**
  ⟨*Private*⟩ Return length of json_data and bin_data, entity of json_data and bin_data.
- **__make_packet(json_data, bin_data=None):**
  ⟨*@staticmethod*⟩ ⟨*Private*⟩ Return a packet including length of json_data and bin_data which is big-endian in order.
- **__make_request_packet(data_type, operation, json_data, bin_data=None):**
  ⟨*Private*⟩ Return a packet of request messages including json_data and bin_data.
- **comm(self):**
  ⟨*Private*⟩ Execute client operations on login, save and upload operations automatically and close the socket when file is uploaded.

C. Actual Implementation

```
self.__client_socket.send(
  self.__make_request_packet(TYPE_AUTH, OP_LOGIN,
  {
    FIELD_USERNAME: self.__id,
    FIELD_PASSWORD: hashlib.md5(self.__id.encode()
  ).hexdigest().lower()
  })
)
json_data_recv, bin_data = self.__get_packet()
token = json_data_recv[FIELD_TOKEN]
```

Listing 1. Actual Implementation of Authorization

```
self.__client_socket.send(
  self.__make_request_packet(TYPE_FILE, OP_SAVE, {
    FIELD_TOKEN: token,
    FIELD_KEY: self.__file,
    FIELD_SIZE: len(open(self.__file, 'rb').read()
  )
  })
)
json_data_recv, bin_data = self.__get_packet()

block_index = 0
with open(self.__file, 'rb') as f:
  while True:
    file_data = f.read(MAX_PACKET_SIZE)
    if not file_data:
    break

    self.__client_socket.send(
    self.__make_request_packet(TYPE_FILE,
  OP_UPLOAD, {
      FIELD_TOKEN: token,
      FIELD_KEY: self.__file,
      FIELD_SIZE: len(file_data),
      FIELD_BLOCK_INDEX: block_index
    }, file_data)
    )

    block_index += 1
    json_data_recv, bin_data = self.__get_packet()

self.__client_socket.close()
```

Listing 2. Actual Implementation of Uploading

TABLE III
TESTING RESULT

ID	Per-file Size	Number of Files	Transfer Integrity Rate (Checked by MD5)	Average Transfer Time (User Time + System Time) (MS)
1	1KB	1000	100%	23.898
2	64KB	1000	100%	24.739
3	512KB	1000	100%	33.972
4	1MB= 1024KB	1000	100%	44.620
5	64MB	10	100%	1165.348
6	512MB	10	100%	9380.012
7	1G= 1024MB	10	100%	11041.033

D. Issues and Problems

There are some issues during the initial designing phase, coding and debugging process, which includes issues regarding socket and threading.

1) Receive empty packet upon sock.connect()

TABLE IV
PROBLEM 1

Issue	Connect to a listening socket for the first time.
Cause	Unknown
Solution	Raise exceptions upon occurrence.

2) Address already in use

TABLE V
PROBLEM 2

Issue	Frequently start and stop the program.
Cause	The intended default settings of socket library.
Solution	Override the setting by using *sock.setsockopt().*

V. TESTING AND RESULTS

A series of tests are used to evaluate the performance of the program. The following shows the test environment, the design and implementation of the pipelined test script, and the test results.

A. Testing Environment

The test was conducted on two physical machines, and the software and hardware conditions used are shown in the table below. These two physical machines will be used as both servers and clients for the all tests.

TABLE VI
TEST EQUIPMENT PARAMETERS

ID	CPU	OS	Linux Kernel	Memory
1	AMD Ryzen 9 5900HX	Arch Linux x86_64	6.0.8	16G
2	Intel i5-7200U	Manjaro Linux x86_64	5.15.65	8G

B. Testing Pipeline Design

1) **Generate random binary files and MD5 values.**
To ensure the validity of the tests, the randomly generated binary files of fixed size are used to send and receive data between the client and the server before each test. The data in the file is randomly generated using /dev/urandom, a special file that serve as cryptographically secure pseudorandom number generators on Unix-like system.
After generating the file, the file MD5 value will be calculated using a shell tool md5sum.
2) **Timer and Time**
Runtime is an important metric for evaluating the performance of a program. In this test, GNU time is chosen as the timer to record the program runtime. This tool is usually built into the shell and can be easily used in the terminal. Pertimer returns three values:
   - real time: the wall clock time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked.
   - user time: the total number of CPU-seconds that the process spent in user mode.
   - system time: the total number of CPU-seconds that the process spent in kernel mode.

   For unified testing, CPU time (user time + system time) is used as the test metric. The experiential data is more convincing and more realistic.
3) **Shell Script**
A shell script was written to enable fast batch testing. The script is a pipeline of the above steps to achieve one-click testing. All tests are conducted in Z shell (zsh).

C. Testing Plan and Result

The following Table III shows the integrity and time taken to transfer binary files ranging in size from 1M to 1G in this C/S architecture. Each file of different sizes is averaged over multiple tests using random files to minimize errors. All tests were performed in the same stable wireless LAN.

The following figure 4 shows the relationship between file transfer time and file size. It can be seen that as the file size increases, the transfer time also increases, which verifies the validity of this test.

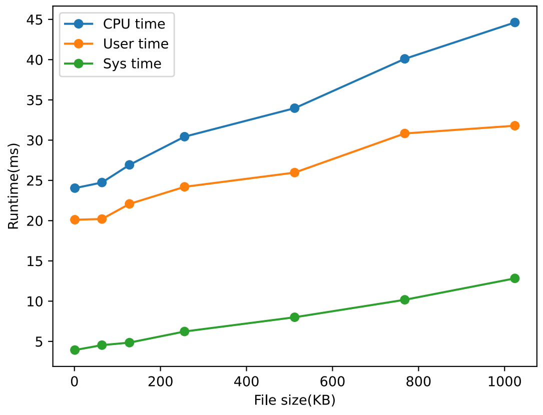

Fig. 3. Relationship between File Transfer Time and File Size

VI. Conclusion

In summary, after fixing the server code problem, the client logged in to the server and uploaded files in blocks through C/S interactions between the two Linux machines. Based on the test results of the code it can be concluded that its file upload speed is relatively stable and efficient.

For further work, considering that there are still more file/data processing and file/data transfer operations by STEP protocol that have not been used, we will do more research on download, deletion, etc.

References

[1] Nottingham, M. "HTTP Semantics" 2022.
[2] Fielding, Roy, et al. "RFC2616: Hypertext Transfer Protocol–HTTP/1.1" 1999.
[3] R. Nossenson, H. Attiya, "The Distribution of File Transmission Duration in the Web" International Journal of Communication Systems, vol. 17, no. 5, pp. 407–19, June 2004.
[4] P. Yan, "Research and Implementation on Network File Transmission Technology" Shanxi Electronic Technology, no. 4, pp. 92-94, November 2016.
[5] S. Zhang, T. Qiu, H. Zhang, et al. "Design and implementation of file transmission system based on multi-threading" Journal of Nanchang University(Engineering & Technology), vol. 35, no. 4, pp. 392-398, 2013.