File Transfer System in LAN Based on Simple Transfer and Exchange Protocol (STEP)

Shuchen Ji
2034172

Xinxin Fan
2039125

Shiqi Huang
2034174

***Abstract*—This project aims to complete a data and file exchange application based on a TCP protocol called Simple Transfer and Exchange Protocol (STEP). This application provides users with data and file uploading, storage, downloading and deleting functions. While developing the project, syntax bugs in the provided server-side code were fixed, and a corresponding client-side code was completed. The application was then tested on two virtual machines and the host with a designated server respectively, by uploading a binary file from the client to the server. Results have shown that the file could be uploaded successfully. Then the file upload performance was tested through repeated operations, and some conclusions were drawn.**

I. Introduction

The file transfer function is fundamental and widely used in many aspects of people's life, including social networks, virtual games and cloud services. The TCP protocol is often used to achieve this function. It is a reliable transmission protocol that enables data to arrive completely and sequentially through a connection. In its application layer, File Transfer Protocol (FTP) is often used as a standard protocol to exchange files between servers and clients [1] [2].

In this research, we would design and complete a project to upload and download files in a client-server structure based on a given TCP protocol, Simple Transfer and Exchange Protocol (STEP). We will first need to address all the syntax errors in the provided server-side code. Then we need to offer and test a client-side code using Python Socket Programming to get authorization and upload a file to the server. The potential implementation of this project is transferring images from computers to cloud services and posting videos and posts to social media etc. In order to solve these challenges, we searched for relevant work to further understand the file transfer between the client and the server. Then, through the design of the C/S network architecture diagram, workflow and pseudo code, we implemented the code on the client using Python Socket and completed the performance test of the file transfer function. Finally we do some future outlook according to our present work.

II. Related Work

Numerous papers and reports have designed different approaches to uploading and downloading files via the network. Hybrid Parallelized File Transport Protocol (HPFTP) and Parallelized File Transfer Protocol (P-FTP) were designed to decrease the time taken to download large files [3] [4]. This was achieved by downloading files from multiple file servers simultaneously, and the client would manage the file distribution by detecting slow servers. The result shows that the download time by HPFTP takes half as long as the conventional file transfer method.

Xiaojin combined the HTML5 interface of file operations with Ajax technology to offer a better service to users when uploading large files [5].

In addition, a research paper designed a similar application to the project proposed in this report, whereas a different protocol is applied. The designed File Transfer in Wireless Networks (FTWN) by Das, Purkayastha and Debnath used File Transfer Protocol and allowed people to transfer files, including large files, in everyday life [6].

III. Design

A. The C/S network architecture diagram

The C/S network architecture diagram was drawn and shown below:

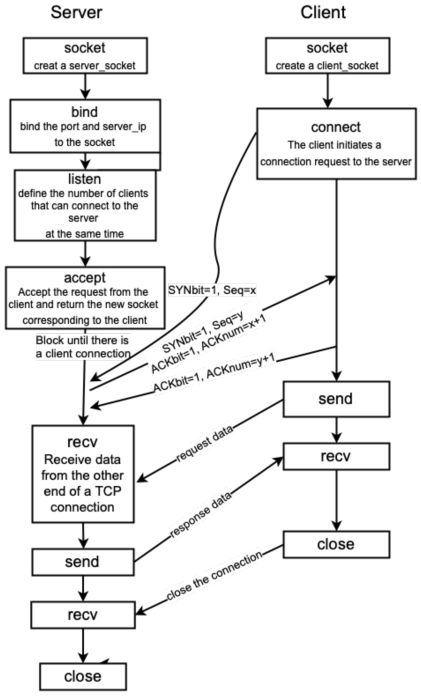


Fig. 1. The C/S network architecture diagram

The server creates a server socket and binds its port and IP to the socket. The server defines the maximum of clients that can connect to the server at the same time by listen method and then keep blocking until a client connection accepts the server's request. After the step, the server will return a new socket corresponding to the client.

The client also creates a client socket and initiates a connection request to the server. Then, to establish a TCP connection between a client and a server process, we need to start a three-way handshake. In the first handshake, the client sends a request message segment to the server. In the second handshake, the server allocates the cache and variables for the TCP connection, returns the acknowledgement message segment to the client, and allows the connection. In the third handshake, the client allocates the cache and variables for the TCP connection and returns the confirmation to the server, which means that the data can be carried. After the three steps, the client and the server can start data transmission.

B. Workflow

The workflow chart of our solution has been drawn below:

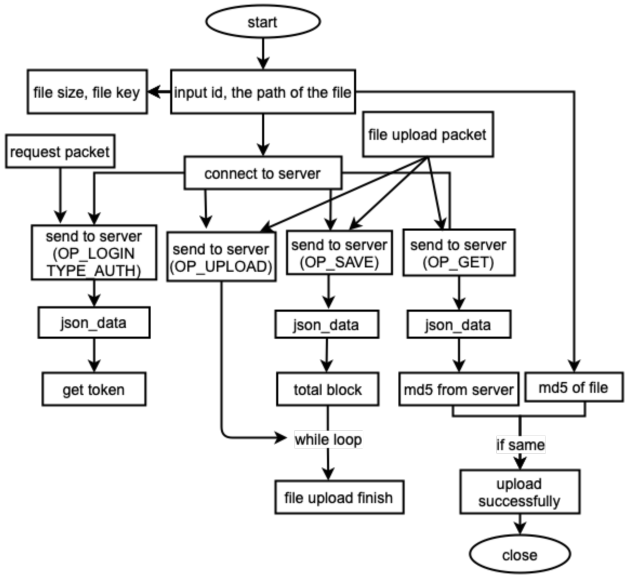


Fig. 2. The workflow chart of our solution on the client

According to the STEP, a request packet will be made firstly on the client to be the login request data that contains OP_LOGIN operation, TYPE_AUTH data_type and json format of the given id with its password which is the MD5 value of id. After inputting the given username, which is id, the login request package will be sent to the server through the send function. Then the server will check the value of the operation, data type, user name and password. If they all match the actual data, authorization will be returned to the client, which means the client login is successful. Therefore, the client can output user information in JSON format, which is defined as json_data. Then, a token can be obtained by json_data["token"] from json_data.

Next, the file transfer operation can be started. Since the server provides corresponding services for file transfer based on the request type, a file upload package was created on the client, which includes operation, data type, and json_data, bin_data. According to the different definitions of the request operations on the server, we defined the different operation types in the file upload package on the client in different steps. The file upload package with OP_SAVE operation was sent to the server, so the client could get the total block of an upload file from the output json_data. Then a while loop was used to send the block, and the file upload packet with OP_UPLOAD operation was sent one by one from the client to the server until all the blocks are uploaded completely. After that, the file upload packet with OP_GET operation was sent to the server, and the MD5 value of the uploaded file would be obtained from the server. Finally, if the MD5 value of the uploaded file from the server matches that calculated from the file path, the file upload successfully.

The following are the details of the client sending file data block by block through the while loop:

1. After the client sends a file upload package and the server returns a response, the server will create an empty file to accept the data sent by the client.
2. The client sends the file size sent to the server in binary format through OP_SAVE operation according to the number of blocks uploaded and the size of each block sent.
3. After receiving the request, the server will return a response and write the received block data into an empty file according to OP_SAVE.
4. Then the client will continue to repeat step 2 and step 3 above for the next block until all blocks are sent.

C. The algorithm

The following is the kernel pseudo codes of the authorization and file uploading:

The kernel pseudo codes of the authorization:

Algorithm 1 The kernel pseudo codes of the authorization

Input: id, server_ip, server_port, get_tcp_packet(), make_request_packet()
Output: token

```
client_socket=socket(AF_INET,SOCK_STREAM)
client_socket.connect((server_ip,server_port))
password=hashlib.md5(id.encode()).hexdigest()
client_socket.send(make_request_packet(OP_LOGIN,
  TYPE_AUTH ,{id,password}))
json_data, bin_data=get_tcp_packet(client_socket)
token=json_data["token"]
return token
```

The kernel pseudo codes of file uploading:

Algorithm 2 The kernel pseudo codes of file uploading

Input: id, server_ip, server_port, filePath, get_tcp_packet(), fileUploadPacket(), client_socket, token, block_size, get_file_md5()

Output: upload successfully

```
client_socket.send(fileUploadPacket(OP_SAVE,...))
json_data, bin_data=get_tcp_packet(client_socket)
total_block = json_data["total_block"]
block_index = 0
while block_index < total_block do
    f = open(filePath, 'rb')
    f.seek(block_size*block_index)
    bin_data = f.read(block_size)
    f.close()
    client_socket.send(fileUploadPacket(OP_UPLOAD, ...))
    json_data, bin_data = get_tcp_packet(client_socket)
    block_index = block_index+1
get_file_md5(filePath)
client_socket.send(fileUploadPacket(OP_GET, ...))
json_data, bin_data = get_tcp_packet(client_socket)
md5=json_data["md5"]
if md5=get_file_md5(filePath) then
    print("upload successfully")
```

IV. Implementation

Working environment: The project is developed in the following host environment:

- Mac OS [host CPU: Apple M1 Pro, physical memory: 16GB, operation system: arm64]
- Ubuntu [CPU: Intel(R) Core(TM) i7-11800H CPU @ 2.30GHz, physical memory: 3.8 Gi, operation system: 64-bit)]

The development software used is PyCharm 2022.2.3 (Professional Edition), and python libraries implemented for this project are socket, JSON, argparse, hashlib etc.

Suppose the TCP connection is established between the Server and Client, then the project will perform as follow (see Fig. 3):

1. The client will first send a request packet with the username to log in and a file. The server will check the username and password.
2. The server will send a response packet with a field Token if the login data is correct.
3. The client will then call the Upload() function to upload the file. The Upload() function will first get the file size and the file key of the uploading file and ask for an uploading plan by sending an OP_SAVE request to the server. The server will return the file's upload plan, including the file key, total block number and size.
4. After the upload plan is received, the client will send the upload file request to the server. The server will read and write the uploaded file to a new file location. It will inform the client once the file is uploaded successfully.
5. The client will check the completeness of the uploaded file by sending a get file request, and the server will return the MD5 value of the uploaded file.
6. If the MD5 value of the uploaded file is the same as the original file, then the client will print 'upload successfully' and close the socket.

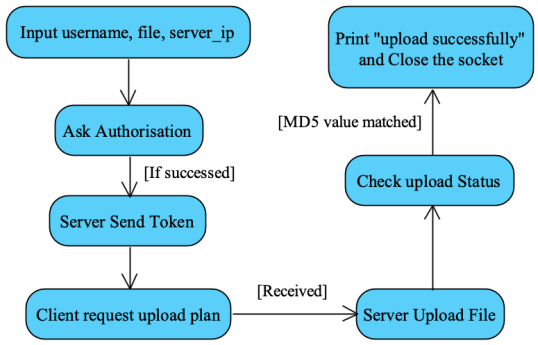


Fig. 3. The flow chart represents the general process of uploading file

While doing this project, some obstacles were met. For instance, 'KeyError: 'token'' appears when the client tries to transfer a file to the server, which means the token cannot be found. It was discovered that the request direction was not added to the file upload package sent to the server, so the request could not be sent successfully. Besides, when executing this program for testing the performance, the file with the same name cannot be uploaded repeatedly since the program is not allowed. Therefore, 'str(uuid.uuid4())' was added in the variable 'file_key' to assign a unique name every time the file is transferred when this program is under test.

V. Testing and Results

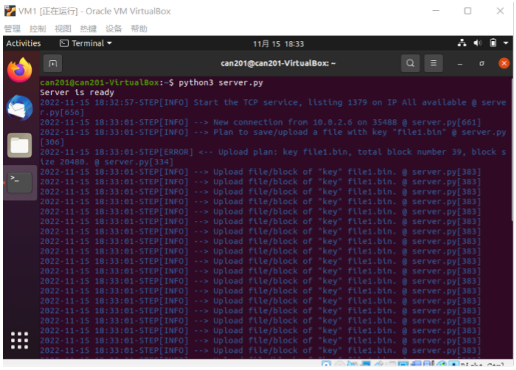
A. Running on the virtual machines

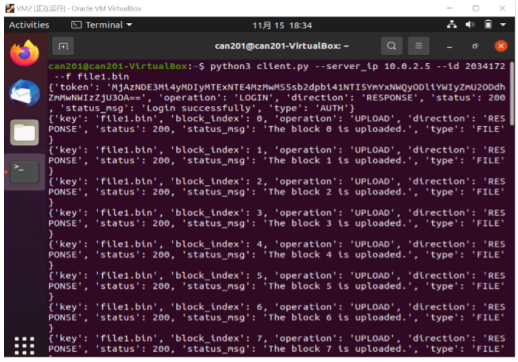
We used two virtual machines of the ubuntu system for testing, named VM1 and VM2, respectively. We used VM1 as the server, VM2 as the client, a binary file named file1.bin whose size is 787KB, as the test file and the student ID as the id.

The test steps:

Running server.py on the VM1 which is the server and outputting 'Star the TCP service, listing 1379 on IP All available' indicates that the server was started successfully. Then run client.py on VM2 in the required command format.

The following are the screenshots of the results:

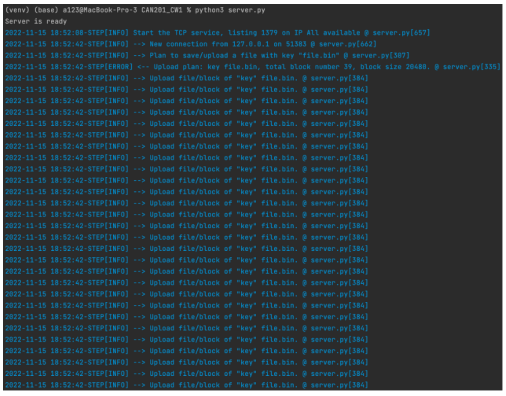


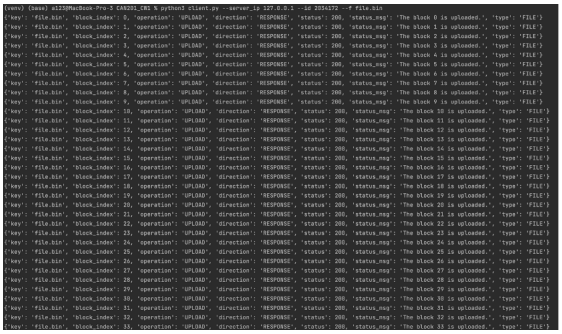


B. Running on the host

We use a binary file named file.bin, whose size is 787KB, as the test file, 127.0.0.1 as the server address and the student ID as the id.

The following are the screenshots of the results:





C. Test upload performance

We test the upload performance in the host environment.

We print the following code into the main function of client.py to calculate the time required to upload files.

```
import time
start=time.time()
#.......
end=time.time()
print("upload  time{}".format(end-start))
```

And the terminal will get the following output:

```
upload successfully
upload time: 0.02666187286376953
```

Since files with the same name cannot be uploaded repeatedly, we modified the code in UploadFile function.

```
#change
file_key=os.path.basename(filePath)
#into
file_key=str(uuid.uuid4())+os.path.basename(filePath)
```

To ensure that the keys of uploaded files are different each time, we have repeated the upload operation of the same file 10 times, and the upload time (s) are respectively 0.028222084045410156, 0.02976202964782715, 0.026664257049560547, 0.025966882705688477, 0.0268402099609375, 0.026638031005859375, 0.024698972702026367, 0.025756120681762695, 0.025079011917114258, 0.023501873016357422.

The obtained line chart of File upload performance is as follows:

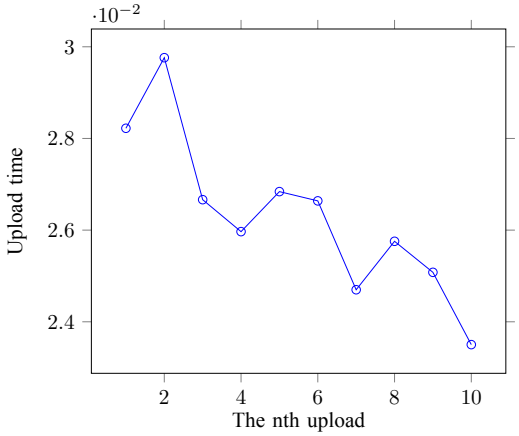


Fig. 4. File upload performance

The average performance of file upload is 0.0123505116 seconds.

VI. Conclusion

This project solves all the bugs in server-side code, and a client-side code is designed to upload the file. A client that

is an authorized user could send a file to upload, and the server will return the uploading plan, including general file information and upload the file. This report examined and analysed the general performance of this uploading process.

Future work will focus on allowing multiple clients to connect to the server and uploading multiple files to the server simultaneously. In addition, since the server cannot control the rate of pushing data to the client [7], the file transmission time is prone to unstable fluctuations, as shown in the file transmission performance tested in this report. Therefore, it is also necessary to solve the problem of excessive changes in the instantaneous throughput during file transmission.

Acknowledgment

Shuchen Ji (2034172) contributes 35% to the project, Xinxin Fan (2039125) contributes 35% to the project, and Shiqi Huang (2034174) contributes 30% to the project.

References

[1] 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.

[2] J. Postel and J. K. Reynolds, "File transfer protocol," 1980, available: https://tools.ietf.org/html/rfc765. [Online]. Available: https://tools.ietf.org/html/rfc765

[3] R. Abhijit and D. B. Kulkarni, "File download delay reduction through parallelization," in *2009 Second International Conference on Emerging Trends in Engineering Technology*, 2009, pp. 653–658.

[4] S. Sohail, S. Jha, and H. ElGindy, "Parallelized file transfer protocol (p-ftp)," in *28th Annual IEEE International Conference on Local Computer Networks, 2003. LCN '03. Proceedings.*, 2003, pp. 624–631.

[5] C. Xiaojin, "Research on file upload based on html5," in *2014 11th International Conference on Service Systems and Service Management (ICSSSM)*, 2014, pp. 1–3.

[6] P. Das, B. S. Purkayastha, and A. Debnath, "Large size file transfer over wireless environment," in *2010 Second International Conference on Computer Engineering and Applications*, vol. 1, 2010, pp. 408–411.

[7] E. Steinbach, Y. Liang, and B. Girod, "A simulation study of packet path diversity for tcp file transfer and media transport on the internet," 09 2002.