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

CAN201 Coursework Part1 Report

Yixuan Hou 2141978

Duan Wang 2144479

Yifan Wei 2142111

Taixu Lou 2142110

Mingyuan Li 2145618

November 15, 2023

Abstract

In CW 1, based on STEP protocol, we fixed 19 bugs in the given server-side code and completed the client-side code using TCP Socket programming. Rigorous testing yielded satisfactory results.

1 Introduction

In network systems, file transfer is pivotal. Our project's initial phase involves crafting a Python Socket program for client-side file upload and download. Tasks include fixing server-side bugs and devising a client-side application for server authorization and file upload. Challenges arose in grasping the STEP message format, which includes different types and operations in the reserved fields of the JSON data part, as well as the status codes. Managing Python Sockets, TCP, and related modules like JSON, struct, threading, and argparse posed additional hurdles.

Our proposal promises an efficient, secure, and reliable data exchange via STEP protocol's diverse operations and status codes. Initially, we tackled bugs in the server code and then built the client-side app. It accomplishes logging in, obtaining a server ”Token” retrieving upload plans, transferring files block by block, and verifying server-side file reception using MD5. This project sets the stage for streamlined, secure data transfers in network systems.

2 Related Work

When uploading or downloading large files, some methods related to network traffic redirection are frequently used to optimize the transmission process and enhance the overall transmission speed.

Several studies have explored strategies to facilitate network traffic redirection. [1] examined the impact of reduced DNS cache lifetimes on web access latency, revealed potential overheads of up to two orders of magnitude, and highlighted the often-violated assumption of client-nameserver proximity in DNS-based server selection. To solve these problems, they proposed protocol modifications to address these issues and improve the accuracy of DNS-based redirection schemes. [2] proposed an innovative approach using a Service-oriented Router (SoR) as a core router within a Content Delivery Network (CDN) to dynamically redirect client requests, leveraging deep packet inspection for more effective and efficient server selection, which ultimately reduces response time by 23.3% compared to the conventional DNS-based redirection.

In this coursework project, a method that transfers a large file with smaller blocks is utilized, which significantly improves the transfer efficiency.

3 Design

3.1 C/S Network Architecture

Based on the given server-side code and the STEP protocol file, we design the client-side for uploading a file to the server and optimize the server code. The server-side and client-side both are connected via a TCP network connection, which provides reliable data transmission.

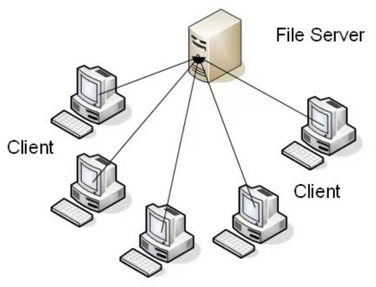


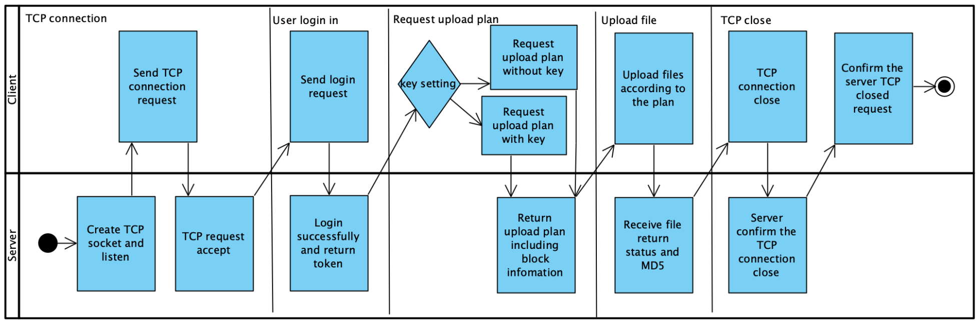
Figure 1: C/S network architecture diagram

In our client-side design, according to Figure 1, our C/S architecture uses the STEP protocol to facilitate distributed system tasks. The client initiates a request, and the server completes user verification, file upload, and returns the result. This allows for task division and cooperation among distributed systems. For the server side, since the server code is given, only the bugs in it need to be fixed. We fixed 19 bugs, please check the implementation section for details.

3.2 Work Flow of our solution

The following is the workflow diagram of our solution:

Login Workflow: The login operation is initiated by the client-side, which assembles a JSON file with



TCP connection
User login in
Request upload plan
Upload file
TCP close
Client
Server
Send TCP connection request
Send login request
key setting
Request upload plan without key
Request upload plan with key
Upload files according to the plan
TCP connection close
Confirm the server TCP closed request
Create TCP socket and listen
TCP request accept
Login successfully and return token
Return upload plan including block infomation
Receive file return status and MD5
Server confirm the TCP connection close

Figure 2: Workflow of our solution

the username and password in accordance with the STEP protocol and encapsulates it in a message to the server. The server then parses the message to see if it meets the requirements of the protocol and extracts the username and password by unpacking the message according to the protocol. If the password is available, the login is successful and the client can use the username to generate a token and return it to the client in a message.

File Uploading Plan Workflow: After the client gets the token, it needs to use it for all subsequent operations. To get the upload solution generated by the server, the client generates a message to the server with the file size, token, and an optional key in the same way as the message is generated. The key can be defined by the client, and if it is not defined, it will be automatically generated by the server in this module. Then the client receives the message and parses it in the same way, extracts or generates the key, and generates the upload plan according to the protocol, that is, the block allocation. After this, the client sends a message containing the above information in the same way.

File Uploading Workflow: Upon receiving the upload plan, the client needs to use a protocol-compliant method to split the file and meet the requirements of the upload plan. The client sends the files along with a message containing block information to the server, which extracts the block information from the message and uses it to reorganize the files after receiving all blocks. Finally, the server generates the MD5 of the received files and sends it back to the client in a message. The client compares the local MD5 with the MD5 of the message to determine the success of the file transfer.

3.3 Algorithm (Pseudo Codes)

Following are the details of the algorithm (pseudo-code).

```
Algorithm 1: Get authorization token
Data: connection_socket, username, password
Result: User token $FIELD\_TOKEN$, response response
json ← username, password;
send_packet(connection_socket, OP_LOGIN, TYPE_AUTH, json);
if response[FIELD_OPERATION] ← OP_LOGIN and response[FIELD_STATUS] ← 200 then
    return response[FIELD_TOKEN], response;
else
    return None;
end
```

(a) Get authorization token

```
Algorithm 2: Request uploading plan
Data: connection_socket, token, key
Result: Response plan with block size and block number
if key is None then
    json_data ← {size, token};
else
    json_data ← {key, size, token};
end
send_packet(connection_socket, OP_SAVE, TYPE_FILE, json_data);
response, _ ← get_tcp_packet(connection_socket);
if response[FIELD_OPERATION] = OP_SAVE and response[FIELD_STATUS] = 200 then
    return response;
else
    return None;
end
```

(b) Get uploading plan

```
Algorithm 3: Uploading File
Data: connection_socket, token, key, response
Result: Response $MD5$ or None
total_blocks ← response[FIELD_TOTAL_BLOCK];
block_size ← response[FIELD_BLOCK_SIZE];
key ← response[FIELD_KEY];
json_data ← {key, block_index, total_block, token};
send_packet(connection_socket, OP_UPLOAD, TYPE_FILE, json_data, data);
response, _ ← get_tcp_packet(connection_socket);
if response[FIELD_OPERATION] = OP_UPLOAD and response[FIELD_STATUS] = 200 then
    return response;
else
    return None;
end
```

(c) File Uploading

Figure 3: Pseudo Codes

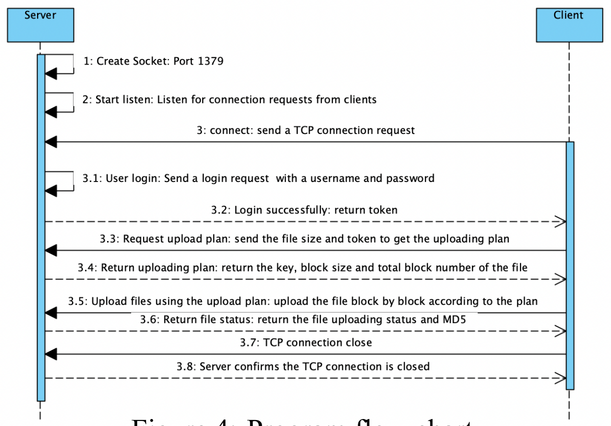
4 Implementation

4.1 Host environment and Development software

The implementation, running on Ubuntu Server 20.04 with an Intel(R) processor and 16GB of RAM, is optimized for stability and compatibility. PyCharm, with its robust Python support, was the chosen IDE, utilizing key libraries like hashlib and argparse for secure hashing and command-line parsing, ensuring efficient and successful development.

4.2 Steps of implementation

The server initiates by creating a socket and listens on port 1379 for client connections. When a client connects, the server authenticates using a username and password. After successful authentication, it issues a token to the client. The client requests an upload plan from the server, providing file size and the authentication token. The server responds with an upload plan detailing file key, block size, and block count. The client uploads file blocks following the plan. Once completed, the server responds to the client with the upload status and MD5 hash value. Upon successful upload, the client closes the TCP connection, and the server acknowledges the closure.Figure 4 is our program flow chart.


Server
Client
1: Create Socket: Port 1379
2: Start listen: Listen for connection requests from clients
3: connect: send a TCP connection request
3.1: User login: Send a login request with a username and password
3.2: Login successfully: return token
3.3: Request upload plan: send the file size and token to get the uploading plan
3.4: Return uploading plan: return the key, block size and total block number of the file
3.5: Upload files using the upload plan: upload the file block by block according to the plan
3.6: Return file status: return the file uploading status and MD5
3.7: TCP connection close
3.8: Server confirms the TCP connection is closed
Figure 4: Program flow chart

4.3 Programming skills

The implementation showcased strong Object-Oriented Programming (OOP) skills, featuring well-structured classes and encapsulation for improved code organization and maintainability. Advanced parallel programming techniques, including parallel processing and multi-threading, enhanced efficiency and responsiveness by handling concurrent tasks effectively. The adept use of error handling and exception management highlighted skilled problem-solving abilities, resulting in a robust and scalable network application capable of managing multiple operations and user interactions concurrently.

4.4 Actual implementation

Authorization function: The authorization function in the client-server interaction follows STEP protocol steps. The client sends login details to the server, which validates the information. Upon successful validation, the server generates and returns a token to the client. This token facilitates subsequent tasks like data uploading and file operations. The function involves validating login details, token generation, and its return to the client.

File uploading function: The file uploading function divides the file into blocks, uploads them sequentially following the server's plan (including permission key, block size, total block number), and displays progress on the terminal. After uploading all blocks, the client checks the file status on the server using the protocol. The implementation includes block division, sequential upload, and server status check.

4.5 Difficulties

Difficulty1: Addressed server.py bugs by fixing module inclusion, correcting function names, ensuring complete function calls, and resolving method call mismatches. Adjustments were made for TCP communication, including changing "SOCK_DGRAM" to "SOCK_STREAM" and adding double quotes to a message. Running server.py with client.py revealed and resolved issues in status codes, logger levels, thread initiation, and key modification.

Difficulty2: To address connection establishment problems, ensured that the client code specifies the correct IP address and port number (1379) for the server.

Difficulty3: For data/file integrity in transmission, implemented solutions using checksum algorithms like MD5 for data integrity verification at the receiving end. For file block integrity, details such as file size and block index are used for verification.

5 Testing and Results

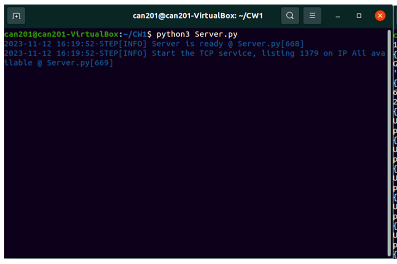
5.1 Testing Environment

Tinycore linux in Virtual box in Windows 10.

Linux version:

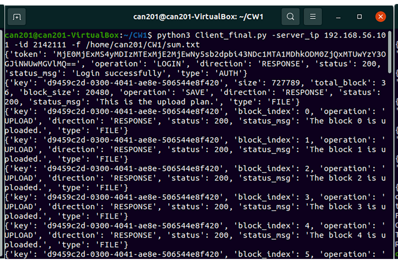
Distributor ID: Ubuntu Description:Ubuntu 20.04.4 LTS Release: 20.04 Codename: focal

5.2 Testing Steps



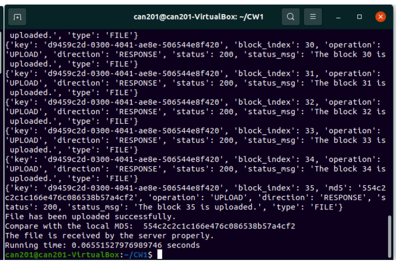
```
can201@can201-VirtualBox:~/CW1$ python3 Server.py
2023-11-12 16:19:52-STEP[INFO] Server is ready @ Server.py[668]
2023-11-12 16:19:52-STEP[INFO] Start the TCP service, listing 1379 on IP All ava
ilable @ Server.py[669]
```

(a) Debug for server code



```
can201@can201-VirtualBox:~/CW1$ python3 Client_final.py -server_ip 192.168.56.10
1 -id 2142111 -f /home/can201/CW1/sum.txt
{'token': 'MjE0MjExMS4yMDIzMTExMjE2MjEwNy5sb2dpbi43NDc1MTA1MDhkODM0ZjQxMTUwYzY3O
GJiNWUwMGVlMQ==', 'operation': 'LOGIN', 'direction': 'RESPONSE', 'status': 200,
'status_msg': 'Login successfully', 'type': 'AUTH'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'size': 727789, 'total_block': 3
6, 'block_size': 20480, 'operation': 'SAVE', 'direction': 'RESPONSE', 'status':
200, 'status_msg': 'This is the upload plan.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 0, 'operation': '
UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 0 is u
ploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 1, 'operation': '
UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 1 is u
ploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 2, 'operation': '
UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 2 is u
ploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 3, 'operation': '
UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 3 is u
ploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 4, 'operation': '
UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 4 is u
ploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 5, 'operation': '
```

(b) Authorization



```
 uploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 30, 'operation':
'UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 30 is
 uploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 31, 'operation':
'UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 31 is
 uploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 32, 'operation':
'UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 32 is
 uploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 33, 'operation':
'UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 33 is
 uploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 34, 'operation':
'UPLOAD', 'direction': 'RESPONSE', 'status': 200, 'status_msg': 'The block 34 is
 uploaded.', 'type': 'FILE'}
{'key': 'd9459c2d-0300-4041-ae8e-506544e8f420', 'block_index': 35, 'md5': '554c2
c2c1c166e476c086538b57a4cf2', 'operation': 'UPLOAD', 'direction': 'RESPONSE', 's
tatus': 200, 'status_msg': 'The block 35 is uploaded.', 'type': 'FILE'}
File has been uploaded successfully.
Compare with the local MD5:  554c2c2c1c166e476c086538b57a4cf2
The file is received by the server properly.
Running time: 0.06551527976989746 seconds
can201@can201-VirtualBox:~/CW1$
```

(c) File Uploading

Figure 5: Testing Steps

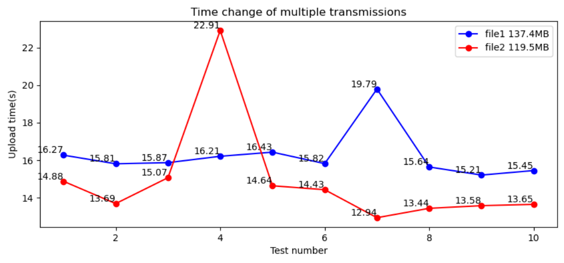
Debug for server code: Initiate the Server's code on VM1's virtual machine. Verify the smooth execution by monitoring the terminal for the prompt, confirming that the server is ready.

Authorization: After server readiness, execute the Client's code on VM2 following a specific order, utilizing the server's IP address (IP). Key information, including the token, operation, direction, and status, is displayed in the VM2 terminal. Additionally, a successful connection message is observed in the VM1 terminal.

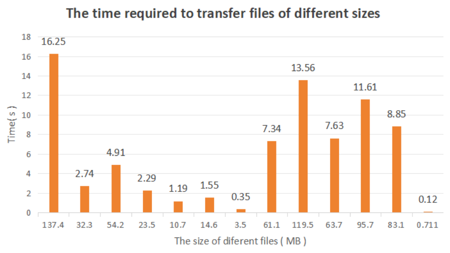
File uploading: Enter instructions into the VM2 terminal while the client code is active. The file transfer duration varies based on file size, and the process is detailed in the terminal to keep users informed. Subsequently, the server calculates the MD5, displayed in the terminal, allowing for a comparison to confirm file integrity.

5.3 Testing Results

It appears that the test results indicate varying transfer times for files, with a generally stable overall trend, but occasional spikes in transfer times for specific files. These anomalies, such as the higher transfer times for File 1 in Result 1 (19.79s) and File 2 in Result 1 (22.91s), are believed to be attributed to fluctuations in network transmission speeds. Additionally, the analysis in result 2 suggests that the file size plays a role in the transfer time variation. Larger files, as demonstrated by a file with a maximum size of 137.4MB, took longer to transfer (16.25s), while smaller files, such as a minimum size of 0.711MB, had quicker transfer times (0.12s). This variation in transfer times based on file size implies that the network's capacity or efficiency may be influenced by the volume of data being transmitted.



(a) Result 1



(b) Result 2

Figure 6: Testing Results

6 Conclusion

Throughout the completion of coursework part 1, we undertook three essential tasks. Initially, we focused on debugging the server-side code. Subsequently, we constructed client-side code, aiming at acquiring authorization and facilitating the function of uploading files to the server. There remains potential for further enhancements, particularly in optimizing the upload speed.

References

[1] A. Shaikh, R. Tewari, and M. Agrawal, "On the effectiveness of dns-based server selection," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, vol. 3. IEEE, 2001, pp. 1801–1810.

[2] E. Harahap, J. Wijekoon, R. Tennekoon, F. Yamaguchi, S. Ishida, and H. Nishi, "Modeling of router-based request redirection for content distribution network," *International Journal of Computer Applications*, vol. 76, no. 13, 2013.