Exploration of Python Socket Programming and Concurrency

Ruoxuan Cao 2141986 Zhihui Wang 2144366 Bowen Fang 2144218
Zeyu Chen 2143415 Yiran Peng 2142136

***Abstract*—The function of file upload and file download is important. In this project, bugs are fixed so that the client and server can run and transfer files. Python socket function to implement the file transfer function. Other approaches such as multithreading and multiprocessing are also applied to find the best solution. This report describes the process of file transfer function implementation, including code designing, testing, discovery, and challenge.**

I. INTRODUCTION

With the advancement of Internet technology, file transfer emerges as a fundamental function. The project aims to facilitate file transfer between client and server through program creation. Two tasks include debugging the server file and crafting the client code for client-side operations. This function finds application in cross-platform or cross-device file transmission, enabling efficient work transmission or personal file backup. During the project, the goal of improving file transfer rates through multithreading in the programming project proved unsuccessful, as continuous exploration and testing revealed that it did not significantly enhance efficiency and introduced additional challenges.

II. RELATED WORK

Redirection is when a user's web page request is redirected to another page by the server. This section reviews research on network traffic redirection and cutting-edge applications of related technologies. Chand et al. (2015) proposed TRCCTP for wireless sensor networks, offering effective congestion control [1]. Shih et al. (2017) established a mobile edge computing platform in a container-based environment, focusing on mobile terminal design, traffic redirection workflow, and eNodeB application awareness [2].

III. DESIGN

A. *C/S network architecture*

Fig. 1. is a two-tier structure in the form of a client/server. The client receives the user's request and sends it over the network to the server to run the database. The client and server are connected via a local area network (LAN) using the TCP/IP and STEP protocols. The SOCKS compatible proxy receives redirected connections intercepted by the redirector on the client node. The server agrees to the client's request and sends the data to the client, which then computes the data and displays the results to the user.

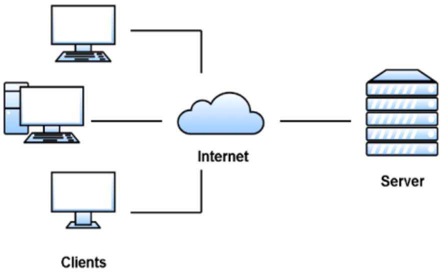


Fig. 1. C/S architecture

B. *Workflow*

Client establishes a link with the server by parsing command line parameters and sending a login request. Upon receiving the login request, the server authenticates the provided credentials and returns a login token for file upload functionality. For file upload, the client sends an upload request with details to the server, which processes and returns results for handling by the client.

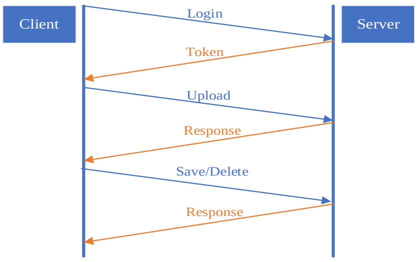
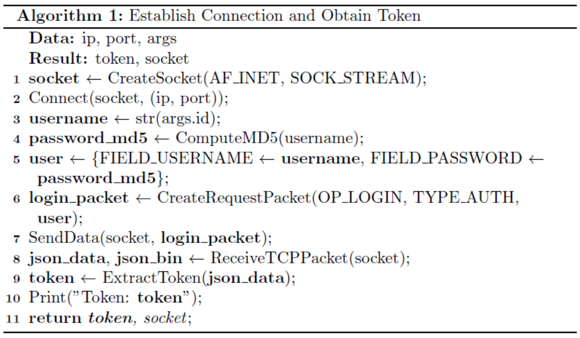


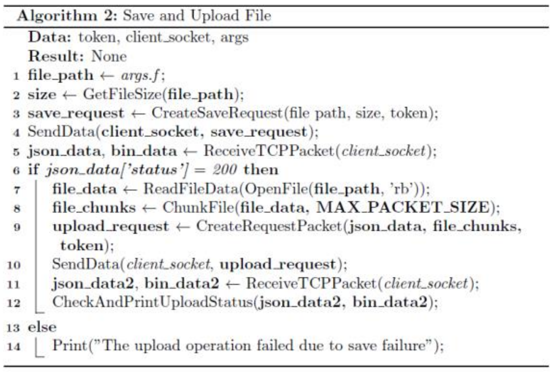
Fig. 2. C/S communication

The client-server interaction persists until disconnection, with clients using functions like *make_packet* and *make_request_packet* to package data into TCP packets and servers using functions like *get_tcp_ packet* to receive and parse data, employing *make_response_packet* to package results into TCP packets before returning them to clients. The C/S communication is shown in Fig. 2.

C. Algorithm



```
Algorithm 1: Establish Connection and Obtain Token
Data: ip, port, args
Result: token, socket
socket ← CreateSocket(AF_INET, SOCK_STREAM);
Connect(socket, (ip, port));
username ← str(args.id);
password_md5 ← ComputeMD5(username);
user ← {FIELD_USERNAME ← username, FIELD_PASSWORD ← password_md5};
login_packet ← CreateRequestPacket(OP_LOGIN, TYPE_AUTH, user);
SendData(socket, login_packet);
json_data, json_bin ← ReceiveTCPPacket(socket);
token ← ExtractToken(json_data);
Print("Token: token");
return token, socket;
```



```
Algorithm 2: Save and Upload File
Data: token, client_socket, args
Result: None
file_path ← args.f;
size ← GetFileSize(file_path);
save_request ← CreateSaveRequest(file path, size, token);
SendData(client_socket, save_request);
json_data, bin_data ← ReceiveTCPPacket(client_socket);
if json_data['status'] = 200 then
    file_data ← ReadFileData(OpenFile(file_path, 'rb'));
    file_chunks ← ChunkFile(file_data, MAX_PACKET_SIZE);
    upload_request ← CreateRequestPacket(json_data, file_chunks, token);
    SendData(client_socket, upload_request);
    json_data2, bin_data2 ← ReceiveTCPPacket(client_socket);
    CheckAndPrintUploadStatus(json_data2, bin_data2);
else
    Print("The upload operation failed due to save failure");
```

IV. IMPLEMENTATION

A. The host environment

CPU: AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz; Memory: 32.0 GB; Operating system: Windows 10.

B. The development tools

The IDE used is PyCharm and the programming language is Python Version 3.6. The Python libraries that the client-side uses include the *hashlib* library which provides a common interface to various secure hash and message digest algorithms. The *JSON* library for encoding and decoding JSON data. The *os* library provides a way to interact with the operating system. The *struct* library is used to pack and unpack binary data. The *time* library for working with time-related functions. The *argparse* library parses command-line arguments. The *socket* library is used to network communication and transfer files.

C. Steps of implementation

This project designs the client architecture according to the server side. The program flow chart is shown in Fig. 3. (1) Import necessary libraries & constants for packet generation. Define max packet size. (2) Create *_argparse* function for *cmd* line argument parsing. (3) Define *make_packet* function to generate packets based on JSON & binary data. (4) Create request packet using *make_request_packet* function. (5) Maintain segment boundaries with *get_tcp_packet* function. (6) Establish TCP connection and obtain token by calling *connection_and_token* function. (7) Save file by calling *save_file* function & upload it to the server. (8) Upload file in fixed-size blocks, handling errors & printing success/failure messages.

D. Programming skills

In this task, OOP is used on the client side for improved code modularity, reusability, and maintenance. Object attributes act as global vars, and methods are below them. Fig. 4. shows the schematic diagram. On the client side, code is modularized with functions for different tasks, enhancing readability & maintainability. *argparse* library is used for parsing *cmd* line *args*, providing a user-friendly interface. JSON processing is employed, encoding & decoding with 'JSON' library. Try-except blocks handle JSON parsing exceptions. For TCP-based network comms, data is packaged & unpacked by struct for accurate transmission.

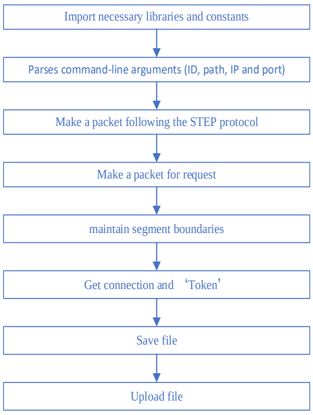
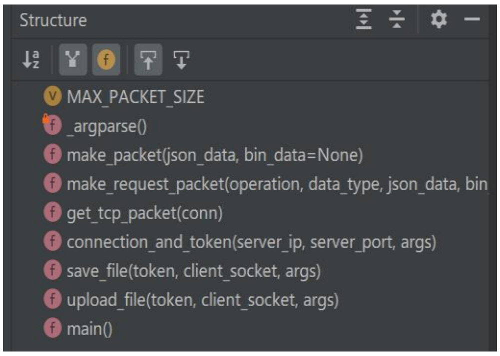


Fig. 3. Flowchart of implementation



```
MAX_PACKET_SIZE
_argparse()
make_packet(json_data, bin_data=None)
make_request_packet(operation, data_type, json_data, bin
get_tcp_packet(conn)
connection_and_token(server_ip, server_port, args)
save_file(token, client_socket, args)
upload_file(token, client_socket, args)
main()
```

Fig. 3. Client's methods

E. *Actual implementation*

The detailed operations of authorization and file uploading are described below.

a. The authorization function:

(1) *client_socket* to create a client socket and connection server side. (2) Get username and calculate the MD5 (*password_md5*). (3) *login_packet* to create a request packet. (4) Receive TCP packet. (5) Extract the token and print ("Token: token"). (6) Return *token, client_socket*.

b. File uploading function:

(1) Obtain file info (*size, file_path*), and build *save-request* dict. (2) Create request packet (OP_SAVE, TYPE_FILE), send to server via *client_socket*. (3) Receive server response, extract *json_data* & *bin_data*. (4) Check success/failure, and output the corresponding message. (5) Call *save_file* to prepare request. (6) Read file data into *image_data*. (7) Determine the number of chunks, construct an upload request packet (OP_UPLOAD, TYPE_FILE), and send via *client_socket*. (8) Receive server response for each chunk, extract *json_data* & *bin_data*. (9) Check success/failure, and output the corresponding message.

F. *Difficulties*

During the project, we are committed to improving file transfer rates. Several methods are taken, single threading, multithreading, and multiprocessing. However, for multithreading and multiprocessing, not all situations are applicable. Some new discoveries about this situation and further details are in part VI.

V. TESTING AND RESULTS

A. *Testing Environment*

Computer OS: Windows 10, 16GB RAM, AMD Ryzen 75800H with Radeon Graphics. Test environment: Two Ubuntu (64-bit) VMs. Server: 8-core CPU, 4096MB RAM. Client: 8-core CPU, 4096MB RAM.

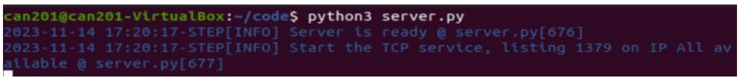
B. *Testing Steps*

The following test steps and screenshots are based on transferring the image otter.jpg.

1. Server code debug: Fixed six syntax bugs in 'server_with_bugs.py'. Enter "python3 server.py" on the server VM terminal. Fig 5. shows the successful display: "Server is ready".
2. Authorization: Check if the token is obtained through authorization after logging in with the

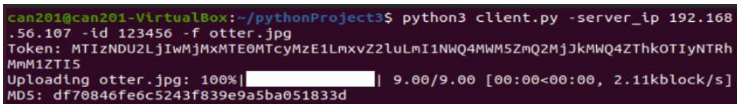
username. Enter "python3 client.py –server_ip 192.168.56.107 –id 123456 –f otter.jpg" on the client VM terminal and the ‘token’ will display on the terminal window. Fig 6. shows the successful effect.

3. File uploading: Check if the specified command line file uploads successfully from client to server, displaying progress in the terminal window. Fig 6. depicts the progress in the form of a progress bar, and MD5 information used to verify if the server has received the file correctly. Fig 7. displays the server-side page post-successful operation.



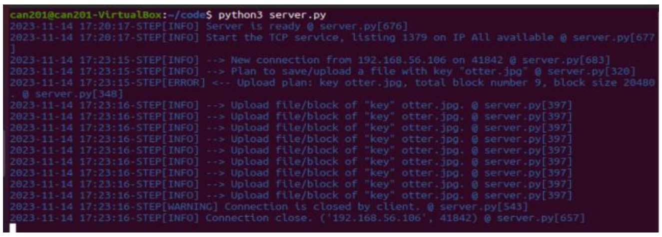
```
can201@can201-VirtualBox:~/code$ python3 server.py
2023-11-14 17:20:17-STEP[INFO] Server is ready @ server.py[676]
2023-11-14 17:20:17-STEP[INFO] Start the TCP service, listing 1379 on IP All av
ailable @ server.py[677]
```

Fig. 5. Server is ready



```
can201@can201-VirtualBox:~/pythonProject3$ python3 client.py -server_ip 192.168
.56.107 -id 123456 -f otter.jpg
Token: MTIzNDU2LjIwMjMxMTE0MTcyMzE1LmxvZ2luLmI1NWQ4MWM5ZmQ2MjJkMWQ4ZThkOTIyNTRh
MmM1ZTI5
Uploading otter.jpg: 100%|████████████| 9.00/9.00 [00:00<00:00, 2.11kblock/s]
MD5: df70846fe6c5243f839e9a5ba051833d
```

Fig. 6. Client running results



```
can201@can201-VirtualBox:~/code$ python3 server.py
2023-11-14 17:20:17-STEP[INFO] Server is ready @ server.py[676]
2023-11-14 17:20:17-STEP[INFO] Start the TCP service, listing 1379 on IP All available @ server.py[677
]
2023-11-14 17:23:15-STEP[INFO] --> New connection from 192.168.56.106 on 41842 @ server.py[683]
2023-11-14 17:23:15-STEP[INFO] --> Plan to save/upload a file with key "otter.jpg" @ server.py[320]
2023-11-14 17:23:15-STEP[ERROR] <-- Upload plan: key otter.jpg, total block number 9, block size 20480
. @ server.py[348]
2023-11-14 17:23:16-STEP[INFO] --> Upload file/block of "key" otter.jpg. @ server.py[397]
2023-11-14 17:23:16-STEP[INFO] --> Upload file/block of "key" otter.jpg. @ server.py[397]
2023-11-14 17:23:16-STEP[INFO] --> Upload file/block of "key" otter.jpg. @ server.py[397]
2023-11-14 17:23:16-STEP[INFO] --> Upload file/block of "key" otter.jpg. @ server.py[397]
2023-11-14 17:23:16-STEP[INFO] --> Upload file/block of "key" otter.jpg. @ server.py[397]
2023-11-14 17:23:16-STEP[INFO] --> Upload file/block of "key" otter.jpg. @ server.py[397]
2023-11-14 17:23:16-STEP[INFO] --> Upload file/block of "key" otter.jpg. @ server.py[397]
2023-11-14 17:23:16-STEP[INFO] --> Upload file/block of "key" otter.jpg. @ server.py[397]
2023-11-14 17:23:16-STEP[INFO] --> Upload file/block of "key" otter.jpg. @ server.py[397]
2023-11-14 17:23:16-STEP[WARNING] Connection is closed by client. @ server.py[543]
2023-11-14 17:23:16-STEP[INFO] Connection close. ('192.168.56.106', 41842) @ server.py[657]
```

Fig. 7. Server running results

C. *Testing Results*

Performance Testing: Emphasized file transfer speed, uploading 10 files of various sizes (9.5KB to 976.6KB). The file upload time fluctuates within a certain small range, therefore, each file has been tested 10 times and the average value is taken. The method of testing time is to use SSH to connect the virtual machine to Pycharm and then use <profile 'client'>. The line diagram drawn is as Fig. 8.

Test results indicate a linear positive correlation between upload time and file size for file transfers below 1MB. In the analysis of factors affecting speed, MAX_ PACKET_ SIZE is one of the influencing factors, as it is a fixed and small value that affects the number of blocks. The file size is linearly positively correlated with the number of blocks, while the number of blocks is positively correlated with the program runtime, thereby affecting the upload speed. In addition, the display in the profile is <method’recv’ of ‘_socket.socket’ objects> takes more time, thus the hardware and network protocols used have a certain impact on the speed.

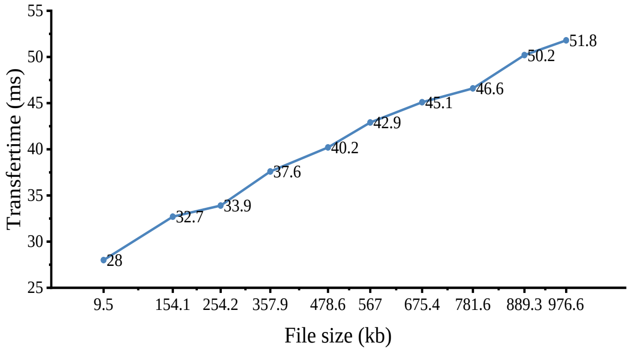


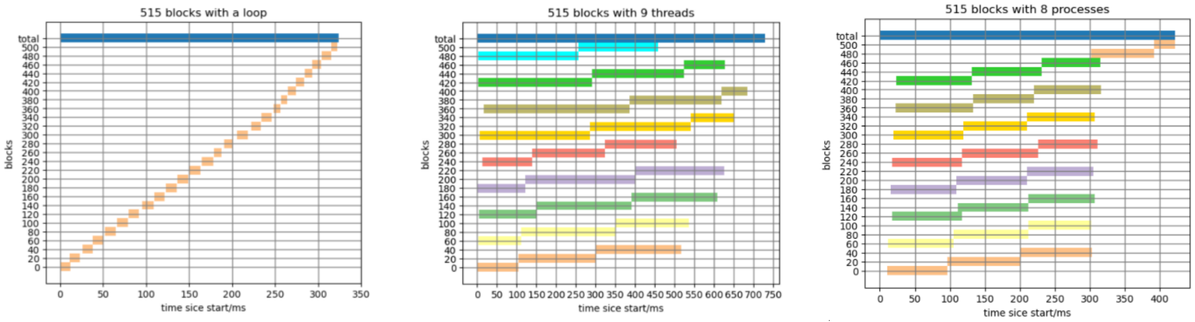
Fig. 8. Results

VI. DISCOVER

To boost file transfer rates, we use six approaches, including "one block per thread," "multiple blocks per thread," thread pool creation, multiprocessing, streaming file transfer, and creating multiple TCP connections. However, after extensive exploration, we found did not yield the anticipated results. The time comparison of multithreading, multiprocessing and a loop for transporting 515 blocks are shown in Fig. 9.

Multithreading

After multiple attempt, we allocating a fixed number of file blocks to each thread and find the suitable number instead of assigning each file block to a separate thread. A thread pool is also created to control concurrent threads. This prompted consideration of limitations imposed by Python's Global Interpreter Lock (GIL) on CPU core utilization or potential server-side constraints.


515 blocks with a loop

515 blocks with 9 threads

515 blocks with 8 processes

Fig. 9. The time comparison of multithreading, multiprocessing and a loop for transporting 515 blocks

Multiprocessing

To overcome the GIL lock dilemma, multiprocessing is applied with a process pool to limit concurrent processes to eight on an eight-core computer, allowing each process to run independently on different CPU cores simultaneously. However, it did not perform faster. This could because the complexity introduced by managing multiple processes, which may be unnecessary for simpler tasks without substantial computation.

Other approaches

Other alternative approaches involved writing file blocks to memory and using threads to transmit them to the server as they were read, or utilizing multiple TCP connections simultaneously to transmit the same file Both were effective for large files but did not significantly improve transfer rates for small files.

CONCLUSION

Our project focused on implementing a file transfer client using Python Socket programming. After thorough testing, the single-threaded approach emerged as the fastest within server limitations, leading to its selection as the final client. Future improvements should address the slowdown in multithreading and multiprocessing by refining server-side modifications and transmission strategies. In conclusion, multithreading did not consistently enhance file transfer rates, prompting us to continue with the more efficient single-threaded approach. The effectiveness of multithreading depends on the context, with limited benefits for CPU-intensive tasks. Choosing concurrency strategies should be context-driven in networking projects.

ACKNOWLEDGMENT

Ruoxuan Cao (2141986) 20.45%; Zhihui Wang (2144366) 20.45%; Bowen Fang (2144218) 19.7%; Zeyu Chen (2143415) 19.7%; Yiran Peng (2142136) 19.7%.

REFERENCES

[1] T. Chand, B. Sharma, and M. Kour, "TRCCTP: A traffic redirection based congestion control transport protocol for wireless sensor networks," 2015 IEEE SENSORS, Busan, Korea (South), 2015, pp. 1-4, doi: 10.1109/ICSENS.2015.7370452.

[2] S. -C. Huang, Y. -C. Luo, B. -L. Chen, Y. -C. Chung and J. Chou, "Application-Aware Traffic Redirection: A Mobile Edge Computing Implementation Toward Future 5G Networks," 2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2), Kanazawa, Japan, 2017, pp. 17-23, doi: 10.1109/SC2.2017.11.