Network File Transfer Project Based on Simple Transfer and Exchange Protocol (STEP) v1.1

1st Ruiyang.Wu
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
2257475

1st Junhao.Huang
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
2256792

1st Daiyan.Wu
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
2257387

1st Yuxiao.Tang
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
2252554

1st Weizhe.Sun
School of Advanced Technology
Xi'an Jiaotong - Liverpool University
Suzhou, China
1927150

***Abstract*—This project implements an efficient, secure, and stable file transfer system based on the STEP protocol within a client-server architecture. Key features include user authentication, multi-threaded uploads, a visual progress bar, and automatic file integrity verification and re-transmission. Once logged into the server and authenticated, the client uses a token to upload local files in chunks. Upon completion, the client compares the MD5 checksum of the remote file to verify its integrity, deciding if re-transmission is necessary. Through architectural and flow diagrams, we detailed the program's logic and structure, and testing confirmed successful file transfers under different environmental conditions. We also analyzed test results to propose scientific hypotheses regarding performance bottlenecks and potential future directions.**

***Keywords*—C/S, Socket, TCP, STEP, Different simulation scenarios**

I. Introduction

A. Background

From the File Transfer Protocol (FTP) proposed by A.K. Bhushan in 1971 in RFC 114 even before the advent of IP networks, to the widespread use of file-sharing systems like Google Drive and Baidu Cloud nowadays, file and information transfer has always been a crucial topic and fundamental function of the internet. In the process of studying CAN201 and inspired by the design approach in related project [1], our group decided to implement a Simple Transfer and Exchange Protocol (STEP), an application-layer protocol based on TCP/IP, to achieve fast, secure, and reliable network file transfer.

B. Task Specification

STEP is a protocol built on TCP, ensuring flow control and reliability of data transmission. The protocol establishes standards for data transfer to maintain data integrity. The server handles incoming requests, while the client initiates requests and performs file transfers. This project enables authorized clients to upload files to the server, allowing any number of uploads. Additionally, to improve transfer speed for large files and ensure data security, we implemented parallel uploading using multi-threading and incorporated automatic data integrity checks and re-transmission functions.

C. Practice Relevance

This project can be widely applied in online file management and content distribution for small businesses or individuals, providing users with efficient and secure file transfer and management services. By supporting user authorization, file storage, and downloading functionalities, this protocol enhances data transfer efficiency while ensuring the security and integrity of information.

D. Challenge and Contributions

During implementation, we encountered challenges related to increasing transfer speed, ensuring information security, and handling packet loss issues. Given the different working hours of team members, we used comprehensive documentation and comments, synchronized the code through private Git repository to improve efficiency, and applied agile development principles for both development and project management.

II. Related Work

To achieve secure and efficient file transfer, the initial design of this project's structure drew inspiration from the concept and programming principles of sockets in a client-server (C/S) model, as presented by Xue and Zhu [1], as well as an open-source local file transfer software ”LocalSend” [2]. Although the STEP protocol is TCP-based, we also researched UDP and found that in the context of this project. TCP offers advantages due to its capabilities for re-transmission of lost packets, flow control, and built-in error checking. Additionally, its connection-oriented nature enhances both the security and speed of data transmission [3]. Furthermore, to improve transfer efficiency, we referenced the work of Chase et al. [4] to optimize and analyze the multi-threading process.

III. Design

A. *Server Debugging*

The first task is to debug a server code that implements the STEP protocol that does not work as expected. The changes are listed as follows:

- Import `struct` package that is required in `make_package` function.
- Rename `get_file_md5(filename)` function, it can now be called correctly.
- Change from `Tcp_Listener(server_port, server_ip):` to `tcp _listener(server_ip, server_port):` The order of the arguments is reversed to improve readability. Function is renamed to comply with PEP 8 [5] – Style Guide for Python Code: Function names should be lowercase, with words separated by underscores as necessary to improve readability.
- Add `server_socket.listen()` to `tcp_listener` which makes the server ready to accept incoming connection requests.
- Create `connection_socket` as a socket object and `'addr'` as a tuple to receive the return values from the `accept()` function, which are then used to create TCP threads.
- Remove the comment block that disables the `tcp_listener()` function in the main function.

B. *C/S network Architecture Design*

This is a client-server application in Fig. 1.

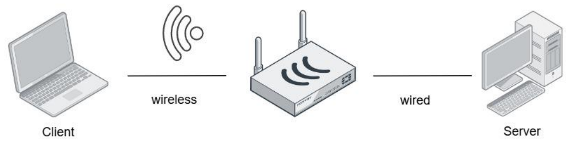


Fig. 1. Client-Server Application

Messages transferred between client and server are in json format and warped as byte, so two functions are needed to make message and parse message from server. Struct package is involved to build binary message data.

- Take the make_message function as an example. It is designed to warp operation, operator type, json message and binary data (when transferring file).
- Another function that parses raw data from the server is also needed. As described in STEP protocol definition, first 4Bytes (32 bits) contains json data length, followed by 4 Bytes binary data length at bias 4Byte. Then json data starts at bias 8 Byte with length defined earlier, finally binary data starts at bias 8+json_length.

C. *Client Sequence Diagram*

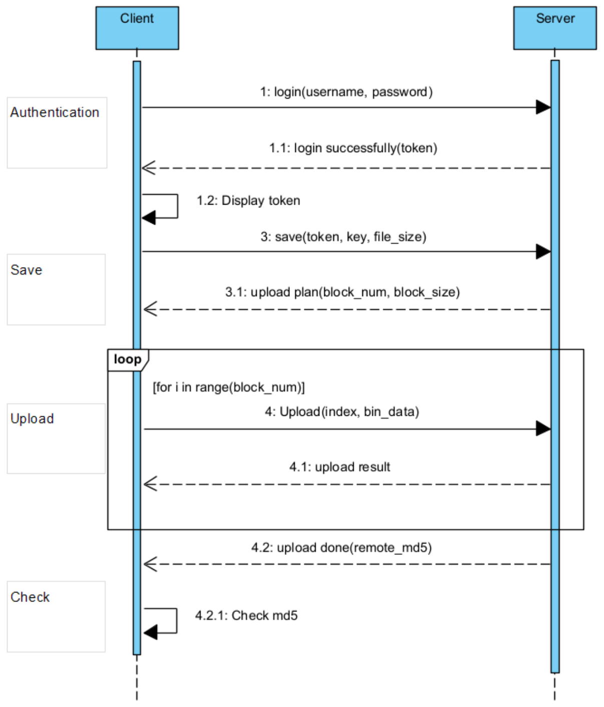

Client
Server
Authentication
1: login(username, password)
1.1: login successfully(token)
1.2: Display token
3: save(token, key, file_size)
Save
3.1: upload plan(block_num, block_size)
loop
[for i in range(block_num)]
Upload
4: Upload(index, bin_data)
4.1: upload result
4.2: upload done(remote_md5)
Check
4.2.1: Check md5

Fig. 2. Client Sequence Diagram

*1) **Authentication**:* In this part, users are required to pass the parameters `'username'` externally.

Then `'username'` is converted to MD5 which is the `'password'`.

To establish a connection with the server, the client generates and sends a message to the server with `'operation'`, `'operator type'`, `'username'` and generated `'password'`, excepting a response message from the server containing `'token'`.

As shown in the sequence diagram, part1 and 1.1 , if the token is returned, a successful login is performed, the token is displayed in the terminal window, if not, the client throws an error.

*2) **File Upload and Storage**:* This part involves uploading files to the server while ensuring data integrity. Users are required to pass the `'path to file'` parameter externally.

According to the STEP protocol definition, a save file message is sent to server, containing operation `'OP_SAVE'`, operator type `'TYPE_FILE'`, `'file_size'` and `'file_key'`, excepting a response message with upload plan.

The plan from server indicates whether the file is existed, if not our client will begin uploading, as shown in sequence diagram.

A multi-threaded upload algorithm is implemented by my group member as shown in the next part.

Finally, after all parts are uploaded, the server responds with the file's md5 value, the client compares the local MD5 with the remote MD5 for data integrity verification.

D. Program Flow Chart

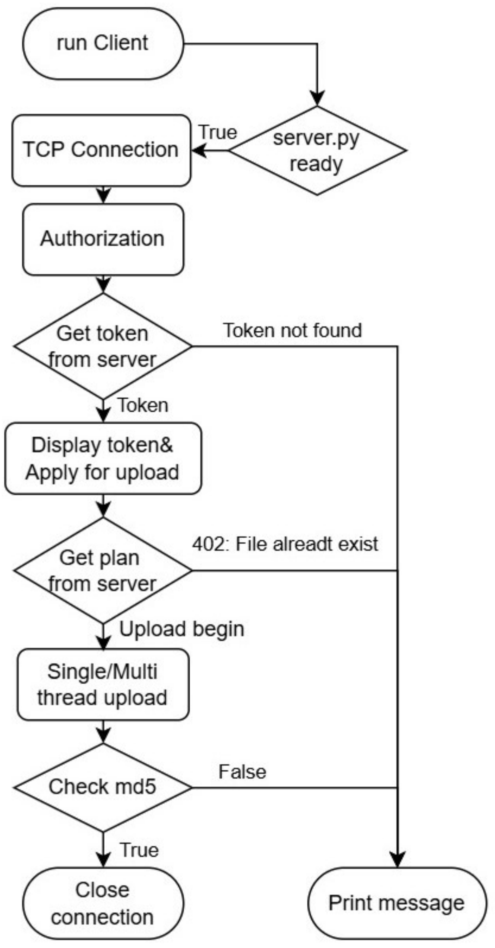


Fig. 3. Program Flow Chart

E. Algorithm

The content of the following code blocks are the kernel pseudo-code for uploading and authorization:

Algorithm 1: Authorization Algorithm

```
Input: username, socket
password ← md5(username)
login_json ← {username, password}
message ← make_message(login_request(OP_LOGIN,
  TYPE_AUTH, login_json)
Send message To socket
Receive json_data['TOKEN'] From server
if TOKEN is None then
    Print ( Token Not Found! )
else
    global token ← json_data['TOKEN']
    Print (token)
end
```

Algorithm 2: Upload Algorithm

```
Input: file_key, file_size, socket
save_file(file, file_size):
message ← make_message(OP_SAVE, TYPE_FILE,
  file_size, file_key)
Send message To socket
plan ← Receive json_data From server
if plan['status'] == 402 then
    Print (File already exist!)
    Return
end
key ← plan['key']
total_block ← plan['total_block']
UPLOAD_BLOCK ← 0
TOTAL_BLOCK ← total_block
for i in range(MAX_THREAD) do
    New thread (target = save_part, args = (file_key,
      socket)) # Multi thread
    thread.start()
end
if 'md5' in UPLOAD_RESULT[-1] then
    local_md5 = getfile_md5(file)
    remote_md5 = UPLOAD_RESULT[-1]['md5']
    if local_md5 = remote_md5 then
    print("Server recieved right file.") else
        re-upload()
    end
end

save_part(file_key, socket):
while UPLOAD_BLOCK <TOTAL_BLOCK do
    LOCK.acquire()
    index = UPLOAD_BLOCK
    UPLOAD_BLOCK += 1
    LOCK.release()
    json_data ← (file_key,index)
    f ← open(file_key) and read as binary
    f.seek(index * MAX_PACKET_SIZE)
    bin_data ← f.read(MAX_PACKET_SIZE)
    msg ← make_message(OP_UPLOAD,
      TYPE_FILE, json_data, bin_data)
    Send msg To socket
    result ← Receive result From Server
    UPLOAD_RESULT.append(result)
end
```

IV. Implementation

A. Develop Environment

During the project development, multiple team members are engaged with the code writing process. The majority of the development are happened in these two computer, as listed in the tables below.

TABLE I
Windows Dev Environment

	Specification
OS	Windows 10
CPU	AMD Ryzen7 5800x
RAM	64GB@3600 MHz
IDE	PyCharm 2024.2.4
Python	Python 3.8.8

TABLE II
macOS Dev Environment

	Specification
OS	macOS Sequoia 15
CPU	Apple M2
RAM	8GB@6400 MHz
IDE	PyCharm 2024.2.4
Python	Python 3.8.20

• **Used Python Modules**: argparse, hashlib, json, os, struct, threading, time, socket

B. Programming Skills

- **Abstraction**: Complex operations are extracted and encapsulated, such as making STEP messages and parsing STEP messages from the server.
- **Modular Design**: The code base is designed in a modular manner, most reusable codes are extracted into their own function, to keep the code clean and readable.
- **Multi-threading**: The client has the ability to use multiple threads to upload the file, in order to achieve better performance, while trying to keep the program thread safe.

C. Steps of Implementation

*1) **STEP Protocol Analysis**:* To implement a client according to the STEP protocol, the protocol must be thoroughly analyzed. STEP is TCP protocol like HTTP protocol, client sends request to the server, then server will response the correspond message. STEP message uses JSON to exchange data information, and use binary data to transfer files.

A STEP message is consists of four parts: length of JSON data part, length of binary data part, JSON data and binary data. First two parts are 4 bytes of unsigned integer, then followed by JSON data that is encoded to binary, finally is the optional binary data.

*2) **Development Process**:* According to the project specification, the broken server-side code needs to be fixed. There are several bugs in the server code, including missing import statement, misspelled function name, wrongly ordered function parameters and missing function call.

After understanding the STEP protocol and fixing the server-side code, attempts are made to write basic function to communicate to the server. During writing and testing these function, we gradually understand the STEP protocol and how the server works. Then we apply abstraction to the code for modular design and optimize the algorithm.

D. Program Implementation

*1) **Authorization Function**:* This function is responsible for login to the STEP server and get the token used for further operations. To make a LOGIN message, the JSON data need ”username” and ”password” field, password is MD5 encryption of the username. In addition, ”type” field is set to AUTH, ”operation” field is set to LOGIN, to tell the server this is a LOGIN message. Then the full LOGIN message is sent to the server through TCP socket, the server will respond the token corresponding to this user if this is a successful login.

*2) **File Uploading Function**:* To upload a file to the server, the upload plan is needed to split file into blocks of data. In this implementation, the file name is used to as the ”key” field, instead of let server generate a random UUID. The ”type” field is set to SAVE, ”operation” field is set to FILE, to request a upload plan from the server.

When server return the response, the status code is checked first to make sure this file is not yet exist on the server. If the file already exists, program will exit and inform the user about file existence. The ”total_block” and ”block_size” is extracted from the response, which is needed for the following upload operation.

Then the program will start uploading the file until all blocks are transferred to the server. When the last block is uploaded, the server will respond a message containing the MD5 of the file received, the server responded MD5 will be checked against local file’s MD5 to make sure the server received the right file.

E. Difficulties and Solutions

*1) **Teamwork**:* In a teamwork scenario, many problems might occur during the development process. Since our team members have different time schedule, we arranged online and offline meetings to discuss and exchange ideas with each other.

To avoid code conflicts and better synchronize working progress with each other, we set up a private Git server and use Git to sync code and resolve conflicts.

*2) **Coding**:* During writing and testing the multi-threaded upload function, we found that sometimes the file cannot be fully uploaded. After we examined the server-side code and error stack trace, we discovered that the server is not thread safe. When server receive a UPLOAD operation, race condition might will happen when the server is receiving data, causing some parts of the file missing or corrupted. The root cause is when server is writing data to disk, such as writing file and logs, there is no synchronization mechanism at all. When multiple threads could reach the file write part of the code simultaneously, race condition will happen, leads to inconsistent writes and exceptions.

There are two possible solutions to the problem, one is modifying the server-side code, add a lock to make the server thread safe. This solution cost less performance loss if implemented properly, but needs more effort in reading the server logic. The other solution is acquiring an atomic lock during file upload process in each thread, make sure that only one thread is sending data to the server at the same time. However, this will cost significant performance loss, making the transfer speed almost the same as single thread.

V. Testing and Results

A. *Testing Environment & Metrics*

The testing was conducted in two phases: the first phase ran on a local host on a MacBook, and the second phase took place on a real local area network (LAN), where the MacBook and a Windows laptop were connected to the same router(Xiaomi ax9000) via wireless (IEEE 802.11ax) and wired (IEEE 802.3bz) connections, respectively. In this setup, the MacBook acted as the client, while the Windows laptop served as the server.

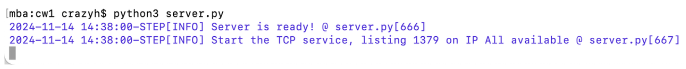
TABLE III
Testing Environment

	Client (MacBook)	**Server (Windows Laptop)**
CPU	Apple M2	Intel(R) Core(TM) i9-12950HX
RAM	8GB @6400MHz	64GB @4800MHz
OS	macOS Sequoia 15.1	Microsoft Windows 10 Pro
IDE	Pycharm 2024.2.4	PyCharm 2024.2 RC
Python Version	Python 3.8.20	Python 3.11.7
Connection	Wi-Fi 6	2.5G BASE-T

To investigate the relationship between the number of threads and transfer speed, we implemented a timer in the client. This timer records the time taken from the start of the transmission task to its completion, enabling us to calculate the average transfer speed. By fitting and analyzing the transfer speeds for various thread counts, we can infer the bottlenecks of the program under multi-threading conditions. Additionally, we tested the speed differences when transferring files of 10MB and 100MB in size. To minimize the impact of network fluctuations on the results, each set of test conditions was repeated three times, and the average values were taken.

B. *Localhost Testing*

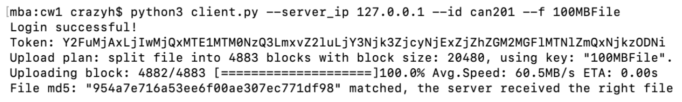
In the first phase of testing, we run the client and server on localhost to check if the program can run properly. As shown in Fig. 4, when running `python3 server.py`, the server correctly displays the message `'Server is ready'`.



```
mba:cw1 crazyh$ python3 server.py
 2024-11-14 14:38:00-STEP[INFO] Server is ready! @ server.py[666]
 2024-11-14 14:38:00-STEP[INFO] Start the TCP service, listing 1379 on IP All available @ server.py[667]
```

Fig. 4. Snapshot of Step 1

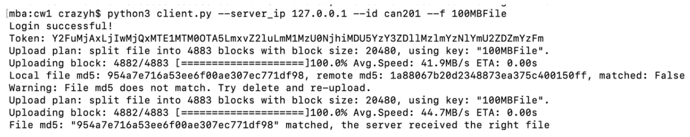
After running `python3 client.py --server_ip --id --f` on the client side, as shown in Fig. 5, the terminal correctly displays the information of the token returned by the server.



```
mba:cw1 crazyh$ python3 client.py --server_ip 127.0.0.1 --id can201 --f 100MBFile
 Login successful!
 Token: Y2FuMjAxLjIwMjQxMTE1MTM0NzQ3LmxvZ2luLjY3Njk3ZjcyNjExZjZhZGM2MGFlMTNlZmQxNjkzODNi
 Upload plan: split file into 4883 blocks with block size: 20480, using key: "100MBFile".
 Uploading block: 4882/4883 [====================]100.0% Avg.Speed: 60.5MB/s ETA: 0.00s
 File md5: "954a7e716a53ee6f00ae307ec771df98" matched, the server received the right file
```

Fig. 5. Snapshot of Step 2

Subsequently, the client automatically starts uploading files, as shown in Fig. 6. In the self-made progress bar, you can see the number of uploaded blocks, progress bar, real-time transmission speed, and Estimated Time of Arrival (ETA). After the transmission is completed, the client automatically verifies the MD5 code to ensure the integrity of remote data, automatically re-transmit files when the remote MD5 code is incorrect.



```
mba:cw1 crazyh$ python3 client.py --server_ip 127.0.0.1 --id can201 --f 100MBFile
 Login successful!
 Token: Y2FuMjAxLjIwMjQxMTE1MTM0OTA5LmxvZ2luLmM1MzU0NjhiMDU5YzY3ZDllMzlmYzNlYmU2ZDZmYzFm
 Upload plan: split file into 4883 blocks with block size: 20480, using key: "100MBFile".
 Uploading block: 4882/4883 [====================]100.0% Avg.Speed: 41.9MB/s ETA: 0.00s
 Local file md5: 954a7e716a53ee6f00ae307ec771df98, remote md5: 1a88067b20d2348873ea375c400150ff, matched: False
 Warning: File md5 does not match. Try delete and re-upload.
 Upload plan: split file into 4883 blocks with block size: 20480, using key: "100MBFile".
 Uploading block: 4882/4883 [====================]100.0% Avg.Speed: 44.7MB/s ETA: 0.00s
 File md5: "954a7e716a53ee6f00ae307ec771df98" matched, the server received the right file
```

Fig. 6. Snapshot of Step 3

C. *LAN Testing*

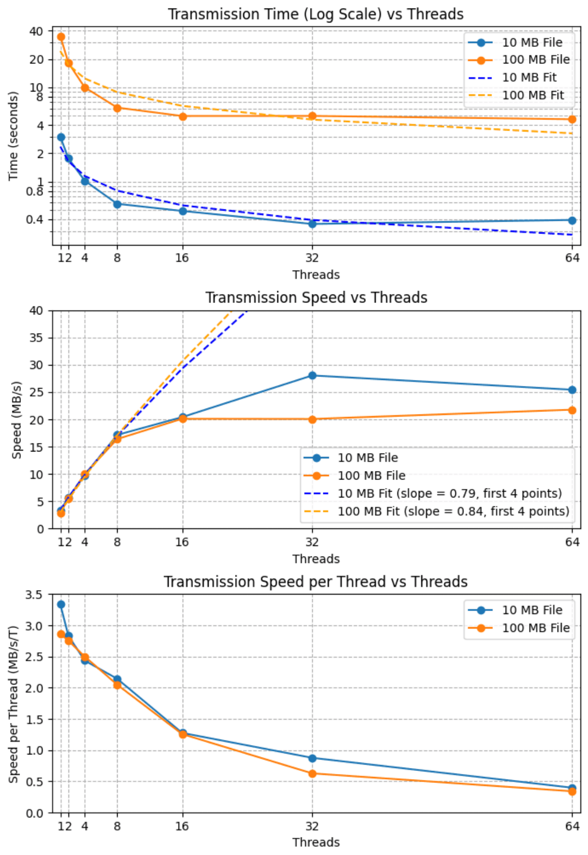


Fig. 7. Visual statistics of Network Transmission Performance

To study performance in realistic conditions, Phase 2 testing took place over a local network, as shown in Fig. 7. We tested transfers of 10MB and 100MB files using 1, 2, 4, 8, 16, 32, and 64 threads, measuring time, average transfer speed, and per-thread speed. We found that increasing threads up to 8 yielded almost linear speed improvements, but beyond 8 threads, performance bottlenecks appeared. Additional threads no longer improved speed and could even lead to a decrease due to increased overhead. Notably, file size did not significantly impact performance in this test.

Performance bottlenecks likely stem from multiple factors [4]. Excessive threads increase CPU load, and when thread count exceeds available CPU cores, context-switching consumes valuable CPU time. On the client side, locks were used for thread safety, but with more threads, lock contention increased, leading to resource access delays. Although this report could not fully measure or optimize each factor, we concluded that eight threads achieve satisfactory transfer results in our environment, balancing speed and overhead.

VI. Conclusion

In summary, our team successfully debugged the server program and developed a multi-threaded client suitable for efficient and secure file transfers. However, the current server program lacks thread safety, which may cause errors during multi-threaded transfers. Despite multi-threading, transfer speed does not yet approach device limits; while user authentication is in place, the TCP-based data transfer is non-encrypted, making the bit-stream vulnerable to interception by unauthorized users. Going forward, we plan to quantify factors causing bottlenecks and propose solutions, implement encrypted transmissions, and optimize both the client and server to enhance performance and security.

VII. Acknowledgment

During the collaboration to complete this project, each member engaged in thorough thinking and communication, and completed tasks within their capabilities and assignments. So we all agree that every individual team member contributes the same percentage to the project.

References

[1] M. Xue and C. Zhu, “The socket programming and software design for communication based on client/server,” in *2009 Pacific-Asia Conference on Circuits, Communications and Systems*, 2009, pp. 775–777.

[2] LocalSend, “Localsend: A self-hosted, cross-platform file sharing app using qr codes to share files across devices on the same network,” https://github.com/localsend/localsend, 2023, accessed: Nov. 11, 2024.

[3] F. T. AL-Dhief, N. Sabri, N. A. Latiff, M. Abbas, A. Albader, M. A. Mohammed, R. N. AL-Haddad, Y. D. Salman, M. Khanapi *et al.*, “Performance comparison between tcp and udp protocols in different simulation scenarios,” *International Journal of Engineering & Technology*, vol. 7, no. 4.36, pp. 172–176, 2018.

[4] J. Chase, A. Gallatin, and K. Yocum, “End system optimizations for high-speed tcp,” *IEEE Communications Magazine*, vol. 39, no. 4, pp. 68–74, 2001.

[5] P. S. Foundation, “Pep 8 – style guide for python code,” https://peps.python.org/pep-0008/, 2001, accessed: November 11, 2024. The function name was updated to comply with the guideline: function names should be lowercase with underscores for readability.