Optimizing TCP File Transfer: A Multi-Process Approach for Efficient Large-File Uploads

Pengkai Chen 2251486
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
SuZhou, China
Pengkai.Chen22@student.xjtlu.edu.cn

Yilin Li 2255705
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
SuZhou, China
Yilin.Li2202@student.xjtlu.edu.cn

Peiling Tu 2251487
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
SuZhou, China
Peiling.Tu22@student.xjtlu.edu.cn

Jinhong Jiang 2251601
School of Advanced Technology
Xi'an Jiaotong-Liverpool University
SuZhou, China
Jinhong.Jiang22@student.xjtlu.edu.cn

***Abstract*—This project focuses on the development and optimization of the client application based on the Simple Transfer and Exchange Protocol (STEP). A multi-process file uploading algorithm is developed and embedded into the client application. After the development, the client application is successfully implemented and tested to verify its feasibility. Based on the testing result, the multi-process algorithm demonstrates higher efficiency in the uploading of files with larger sizes compared to the single-process algorithm.**

Keywords—STEP, client, algorithm, multi-process, file upload

I. INTRODUCTION

With the rapid evolution of the Internet, more applications and functions related to the computer network are being implemented. The file transmission is one of the most pivotal components of the modern network communication. With the increasing demand for daily file transmission, a stable and well-constructed client-server architecture is required.

This project aims to develop a client application through Python Socket Programming based on the STEP, realizing functions such as authorization, uploading, and other useful operations. STEP is a TCP-based protocol that could guarantee a safe and reliable connection between server and client by utilizing the token and JSON format data transmission mechanism.

To achieve this goal, this project is separated into the following steps:

1. Debug the server-side application to eliminate remaining syntax errors.

2. Complete the authorization function, retrieve the correct token from the client, and use the token for verification.

3. Retrieve the file uploading plan, upload the file to the server segmentally, and retrieve the file md5 for verifying the completion of the uploading process.

In this project, an optimized client application is successfully developed. To reduce the file upload time, a specific concurrent file reading algorithm was designed and embedded into the client application by utilizing the multi-process code execution strategy. Subsequently, the client application was successfully implemented and tested for the overall file uploading performance. According to the testing result, the multi-process approach is proven to have a higher uploading efficiency for large files than the single-process approach.

II. RELATED WORK

A. Socket communication

This technology was developed by Jon Postel in 1970s, used for the data communication between two processes running on different devices. The socket is the interface between the application and transport layers of the network communication, allowing developers to implement various message exchange mechanism across networks through programming [1].

B. Data packing and unpacking

The struct library supports the functionality of packing and unpacking binary data, following the concept of structures in C for data serialization. This method facilitates the accurate data exchange between the client and server, ensuring a reliable transmission while addressing disk space efficiency issues [2].

C. The ***md5*** hash algorithm

Introduced by Ron Rivest in 1991, the md5 hash algorithm was designed to secure sensitive information during network transmission by preventing direct data interception and exposure. However, due to vulnerabilities that make it susceptible to certain attacks, md5 is no longer recommended for cryptographic purposes. Instead, it remains popular in non-cryptographic applications, such as data integrity verification in file transfer, where security risks are lower [3].

D. JavaScript Object Notation (***JSON***)

JSON is a lightweight data-interchange format introduced by Douglas Crockford in 2001. Its simplicity, human-readable structure, and standardized format have made it a widely adopted choice for data exchange across various platforms and applications. By representing data compactly and efficiently, the utilization of JSON minimizes the bandwidth consumption during network transmission, thus enhancing the speed and overall efficiency of the data transfer, making it an ideal solution for network communication, data storage, and integration in modern web development [4, 5].

III. DESIGN

A. The C/S Network architecture & Code Workflow

The architecture of the C/S network communication mechanism is exhibited by Figure 1.

1) Layer Communication

The packet created by the client/server application will travel down through each layer of the computer network stack, from the top to the bottom.

2) TCP connection Setup

Firstly, the TCP connection will be set up between the two applications.

3) Login operation

Then, the client sends the login request packet to the server for verification. The token will be sent by the server if the authorization is successful.

4) File save operation

After the login process, the client will send a file save request packet to the server for retrieving the file upload plan.

5) File uploading operation

A multi-processing approach is designed to read the file blocks concurrently, reducing the total uploading time. Along with the reading processes, the main uploading process will concurrently send all the file blocks to the server. After all blocks are uploaded, the server will send the md5 value to the client to notify the completion of the uploading operation. Once the md5 is detected by the client, the application will exit automatically.

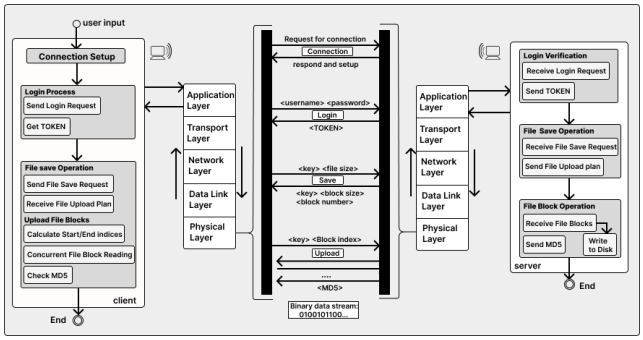

Fig. 1. The C/S Architecture & Code Workflow

B. The Optimization of the file uploading strategy

To maximumly reduce the total file uploading time, the most efficient strategy will be the parallel TCP uploading. However, based on the current server code architecture, this strategy will not be achievable. Thus, reducing the time cost of the file block reading process will be the only choice.

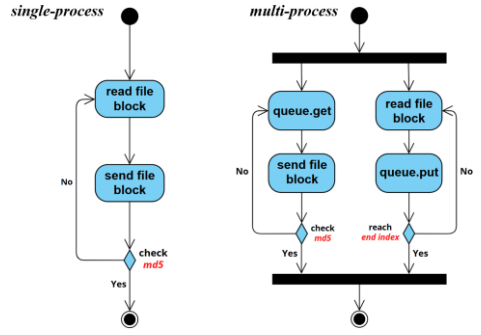

Fig. 2. The single & multiple process uploading mechanism diagram

By analyzing the code execution workflow manifested by Figure 2, the uploading time of the single-process and multi-process can be approximately expressed as equation (1) and equation (2) as follows. (Let $\boldsymbol{n}$ denotes the total block num; Let $\boldsymbol{m}$ denotes the number of blocks allocated to each child process; Let $\boldsymbol{CMO}$ denotes the time of the concurrency management overhead; Let $\boldsymbol{u}$ and $\boldsymbol{r}$ denote words "upload" and "read")

$$Time = \sum_1^n (r + u) \tag{1}$$

$$Time = \begin{cases} \sum_1^n u + CMO\,, & \sum_1^n u > \sum_1^m r \\ \sum_1^m r + CMO\,, & \sum_1^n u < \sum_1^m r \end{cases} \tag{2}$$

Thus, according to equations, the time saved by using the concurrent file reading strategy is approximately equal to $\sum_1^{n-m} r$. However, the single-process uploading strategy will be superior in time if $CMO > \sum_1^{n-m} r$ if the file size is relatively small. Therefore, an adaptive file reading algorithm based on the user's computer hardware configuration will be the optimal solution. Nevertheless, due to the limited software developing time, this technical issue remains unsolved.

C. The Pseudo Code

The pseudo code of the client application is shown by Figure 3 as follow.

Algorithm 1 Main Function Algorithm

```
function MAIN
    IP, id, file_path ← argparse()
    socket ← socket_set_up()
    token ← login(socket, id)
    file upload plan ← upload_plan_retrieve(socket, file_path)
    Evoke → file_upload(socket, file_path)
    Wait For File Upload Process To End.....
    Close client socket, disconnect TCP connection
    Exit application
end function
```

Algorithm 2 Login Function Algorithm

```
function LOGIN(socket, id)
    password ← hashlib.md5(id.encode()).hexdigest().lower()
    socket.send(id, password)
    receive and return token
end function
```

Algorithm 3 File Upload Function Algorithm

```
function FILE_UPLOAD(socket, file_path)
    queue ← Queue()
    process_num = set total child process num: 3
    index ← index_calculation(process_num)
    for i : 1 to (process_num + 1) do
        concurrent evoke ⇒ file_read(queue, index, file_path)
    end for
    evoke ⇒ file_block_upload(socket, queue)
    Wait For All The Processes To End.....
end function
```

Algorithm 4 File Read Function Algorithm

```
function FILE_READ(queue, index, file_path)
    Open and map the file into RAM based on file_path
    while not exceed end index do
        bin_data ← read()
        queue.put(bin_data)
    end while
    Close the file
end function
```

Algorithm 5 File Block Upload Function Algorithm

```
function FILE_BLOCK_UPLOAD(socket, queue)
    while md5 not detected do
        if queue is not empty then
            bin_data ← queue.get()
            socket.send(bin_data)
            ack received or retransmission
        end if
    end while
end function
```

Fig. 3. The pesudo code of the client application

IV. IMPLEMENTATION

A. The Host Environment

One laptop device is used for developing the client-side software, the environment of which is demonstrated as follows.

TABLE I. THE HOST ENVIRONMENT CONFIGURATION

Laptop	Huawei MateBook 14
Operation system	Win 11 Pro version
CPU	Intel i5-1340P

B. Developing tools

The Python programming language (version: ***3.12.6***) and PyCharm IDE are used for the software development. Several Python modules are imported from the Python standard library and utilized within the development to achieve the target functional requirements of the client application, which are listed in Figure 4 as follows. (The utilization of Python ***mmap*** module has been **permitted** through E-mail consultation.)

```
import argparse
import hashlib
import json
import mmap
import os
import struct
import sys
import time
from multiprocessing import Process, Queue
from os.path import getsize
import socket
```

Fig. 4. Imported Python modules

C. Steps of implementation

The program flow of the client application accords with the following flowchart shown in Figure 5. The details of the program execution as follows.

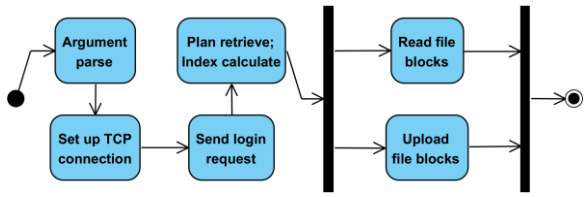

Fig. 5. The client application code execution flowchart

1) Argument parse:

a) The command line arguments will be read and transmitted to the application along with the code execution.

2) Set up TCP connection:

a) The TCP connection between the server application and the client application will be set up initially. The corresponding socket object will be returned to main function and utilized by the client.

3) Send login request:

a) The login will be started along with the completion of the TCP connection. The value of the password will be calculated by performing a md5 hash operation on the given username, then converting the result into a lowercase 32-character hexadecimal string.

b) Then, A dictionary will be created to store the value of the username and password, subsequently passed to the message packing function for the JSON conversion and encoding, after which will be sent to the server.

c) After receiving the response message from the server, the client will check the received status code to validate the login status. If successful, the token will be retrieved from the dictionary and displayed for visualization.

4) Plan retrieve; Index calculate:

a) After the login operation, the client will retrieve file information by utilizing the file path, pack it, and send it to the server for retrieving the file uploading plan.

b) The uploading plan information will be displayed on the Python console after the validation of the server response. Then, the client will calculate the average number of file blocks need to be tackled by each process and the corresponding start/end index for each child process based on the designed algorithm.

c) The data of indices will be stored and further transmitted to every child process for file partitioning.

5) Read file blocks:

a) The child process will be started one after another, evoking the target file reading function.

b) The target file will be opened and mapped into the system memory based on the file path in each child process for quicker file data retrieval. Then, each child process will start to read file block data iteratively from its own specific start index till reaching beyond the end index and put the block data into queue at the end of every iteration.

c) The file will be close ultimately along with execution of the break command.

6) Upload file blocks:

a) The uploading process will be started after the initiation of all the child processes, which will iteratively check whether the queue is empty or not and proceed if file block data is detected.

b) A dictionary will be created to store the file name and block index value, which will be passed to the message packing function together with the file block data for the JSON conversion and encoding and sent to the server ultimately.

c) Then, the client will continuously wait for the response from the server and retransmit the message if nothing is received within 20 seconds.

d) The server response message and status code will be displayed after every block uploading for monitoring, while the client will iteratively check the existence of the file md5 value. Once detected, the value of md5 will be displayed. The TCP connection will subsequently be closed and the application will be automatically terminated.

D. Programming skills

- Procedural programming

  The code execution logic of the client application is designed by following the principle of procedural programming concept, decomposing the program logic into a series of reusable functions that are executed in a certain order.

- Abstraction

Encapsulating complex operations such as message creating, message packing, and message unpacking into specific functions, which could highly facilitate the interaction between users and the system and offer great convenience to the code tracing task

- Program modular design

  The whole client application is evenly divided into several specific Python functions based on the functionality of each one, which enhances the readability and maintainability of the code.

- Multi-process

  To maximumly increase file uploading speed, a multi-process strategy is utilized.

E. Development difficulties

*1) The Python global interpreter lock (**GIL**)*

The GIL is a mechanism in the Python interpreter to ensure that only one thread in an interpreter process can execute the Python bytecode at one single time. Therefore, it will be generally impossible to achieve the real sense of parallel code execution by utilizing a multi-thread strategy. To bypass the restriction of the GIL, the multi-process and queue is used. Multiple processes will be started, each with its own GIL and own memory space, to utilize the multi-core CPU as much as possible to achieve the true parallel code execution.

V. TESTING AND RESULTS

A. Testing environment

Two laptops are used for the code testing, acting as the server and client respectively. To ensure the stability of the network test environment, two laptop devices are connected to a wi-fi hotspot provided a mobile phone, in which case the inter-device TCP connection will be achievable in the same subnet environment.

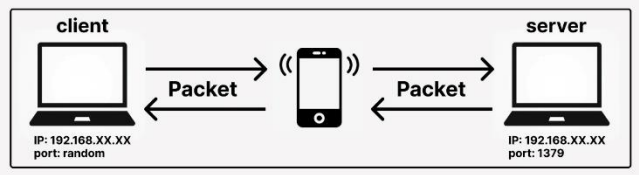

Fig. 6. The server application executing result

B. Testing steps

Python ***time*** module is utilized for recording and calculating the file uploading time.

1) Step 1: Executing the server application

Executing the server code after all remaining syntax errors is fixed. “Server is ready” is correctly displayed on the Python console demonstrated by Figure 7.

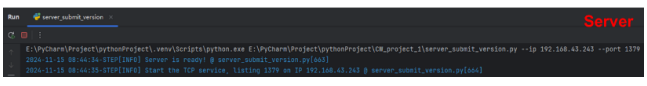

Fig. 7. The server application executing result

2) Step 2: The client login operation

Executing the client code, the login status and correct token are successfully retrieved and displayed on the Python console referring to Figure 8. Then the client application will automatically proceed to the next operation.

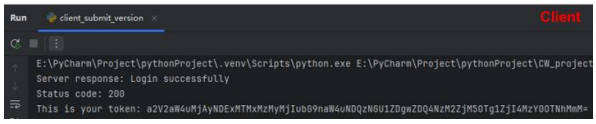

Fig. 8. The client-side application login operation executing result

3) Step3: The client file save & uploading operation

After the login operation, the file save operation request is subsequently sent to the server and the file uploading plan is successfully retrieved and displayed on the Python console shown by Figure 9. Then, the file uploading operation will be launched automatically. The file md5 is successfully retrieved and displayed ultimately and total uploading time is calculated as well.

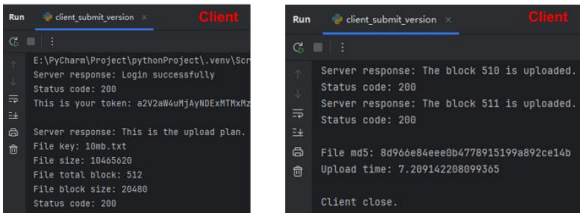

Fig. 9. The file save & uploading operations executing results

C. Testing results

Uploading operations of files with different sizes, ranging from 100 megabytes to 2000 megabytes, is conducted to test the performance variance of two different file uploading algorithms. Based on the hardware configuration of the client device, the maximum number of usable processes is set to be 4. To enhance the generalizability of the testing results, the file uploading algorithm was tested sequentially with the number of utilized processes set from 1 to 4 respectively.

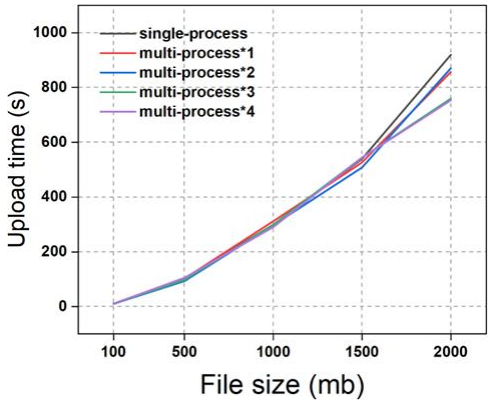

Fig. 10. The uploading time of different upload strategies

Based on the data demonstrated by Figure 10, it can be observed that the variance in uploading time between single-process and multi-process is not significant when the file size is less than 1000 megabytes. As the file size reaching over 1500 megabytes, the multi-process algorithm gradually manifests higher efficiency in file uploading operation. A conspicuous time variance can be noted from the diagram when the file size reaches 2000 megabytes, in which case the 3 and 4 multi-processes algorithms demonstrate a much higher uploading efficiency compared to the others. It can therefore be predicted that the time advantage of a multi-process uploading algorithm will likely be further amplified when handling files with larger sizes, which accords with the aforementioned equations in ***Design*** section.

VI. CONCLUSION

In summary, a client application based on the STEP is developed, optimized, and implemented successfully. A multi-process file uploading algorithm is designed and embedded, whose performance are tested by performing multiple file uploading operations with different file sizes. Testing results demonstrated that the multi-process algorithm shows a higher efficiency in uploading larger files compared to the single-process algorithm.

For future work, developers should further optimize the file reading algorithm to make it highly adaptive to any hardware configuration to maximize the file uploading efficiency.

ACKNOWLEDGMENT

Pengkai Chen (2251486) contributes 25% to the project, Yilin Li (2255705) contributes 25% to the project, Peiling Tu (2251487) contributes 25% to the project and Jinhong Jiang (2251601) contributes 25% to the project.

REFERENCES

[1] B. Ciubotaru and G.-M. Muntean, "Socket-Based Client–Server Communication," 2013.

[2] L. Jie, "Realizing the File Packing and Unpacking by Means of C," Science Technology and Engineer, 2005.

[3] A. Mohammed Ali and A. Kadhim Farhan, "A Novel Improvement With an Effective Expansion to Enhance the MD5 Hash Function for Verification of a Secure E-Document," *IEEE Access*, vol. 8, pp. 80290-80304, 2020.

[4] F. Pezoa, J. L. Reutter, F. Suárez, M. Ugarte, and D. Vrgoč, "Foundations of JSON Schema," Proceedings of the 25th International Conference on World Wide Web, 2016.

[5] D. Crockford, "JavaScript: The Good Parts," 2008

APPENDIX

Figure 11 and Figure 12 is the image of conducting the file upload test using two laptop devices and the snapshot of the wi-fi hotspot connection.

Fig. 11. The file uploading test

Fig. 12. The wi-fi hotspot connection for the file uploading test