



Xi'an Jiaotong-Liverpool University

西交利物浦大學

CAN201 Introduction to Networking

Networking Project

Large Efficient Flexible and Trusty

(LEFT) Files Sharing

Project Report

Student Name: Wenbo Yu

Student ID: 1929603

Catalog

1. Abstract	2
2. Introduction	2
2.1 Project requirement	2
2.2 Background	2
3. Methodology	3
3.2 Design_v1.0	3
3.3 Design_v2.0	4
4. Implementation	4
4.1 Implementation layout	4
4.2 Difficulties and solutions (programming skills)	6
5. Testing and results	7
5.1 Testing environment and plan	7
5.2 Results and discussion	8
6. Conclusion	9
7. Reference	9

1. Abstract

This project is aimed to develop a Large Efficient Flexible and Trusty (LEFT) Files Sharing program based on TCP protocol via Python. It is a file synchronization protocol that ensures file integrity through breakpoint transfer and modification retransmission features while providing transfer efficiency.

2. Introduction

2.1 Project requirement

1. Design TCP-based transport protocols via Python instead of using existing protocols, such as HTTP and FTP.
2. The files and folders in the `./share` directory will be automatically synchronized and transferred without error and loss after entering the ip parameter of the other PC.
3. After this program has been forcibly closed and restarted, it is possible to continue the transfer that was not completed.
4. Modified files can be automatically detected and synchronized.

2.2 Background

In recent years, file synchronization services are widely considered as the key to collaborate with colleagues across devices, platforms, and to keep files secure, such as Groove, Dropbox, and Google Drive, and the TCP protocol is precisely one of the most important protocols for file synchronization applications [1], [2].

3. Methodology

This section will describe the protocol according to the two main versions in the development process and analyze the vulnerabilities and improvements..

3.2 Design_v1.0

In version 1.0, the protocol was split into file list, file transfer and download queue, containing a total of four threads.

- ❖ **File_list_Client**: Send requests, receive file list information, generate download tasks.
- ❖ **File_list_Server**: Waiting for new connections and sending local file list.
- ❖ **File_Client**: Send a file download request and receive the file.
- ❖ **File_Server**: Wait for new download requests and transfer files.

Drawbacks:

Unnecessary waste of resources due to redundant threads. For example, the file transfer thread is still running after a file synchronization has finished, which results in wasted resources.

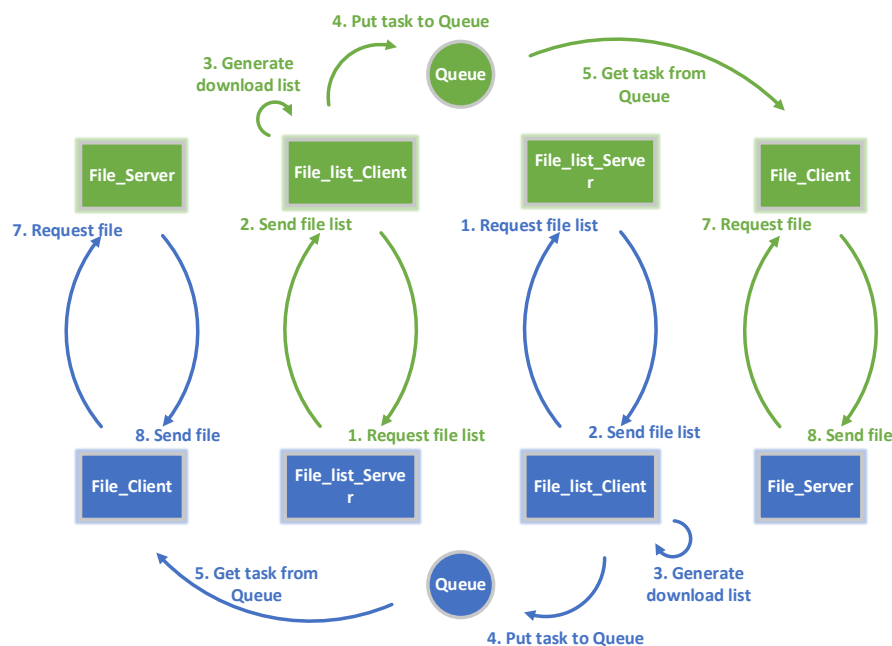


Figure 1: Flow chart of Design_v1.0

3.3 Design_v2.0

In version 2.0, the file receiving side was merged into a sub-thread of the file list client, which means that the file receiver threads will only be enabled when there are files that need to be synchronized.

Core design:

Merge the file receiver (File_Client) into a sub-thread of the File_list_Client, which improves efficiency and stability.

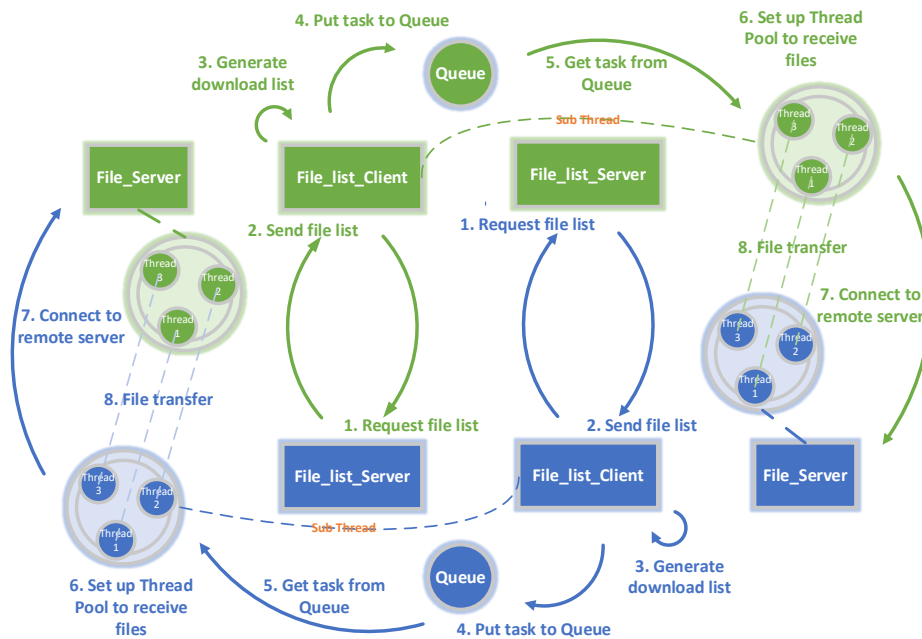


Figure 2: Flow chart of Design_v2.0

4. Implementation

The implementation the program will be analyzed according to the design of version 2.0, where Figure 3 and 4 show the logical framework of the program in terms of the Activity diagram and Sequence diagram.

4.1 Implementation layout

This program consists of the following files:

- ❖ **main.py**: Contains the body of the file and an interface for receiving command line arguments.
- ❖ **filelist.py**: Contains methods for handling file list.
- ❖ **filetransfer.py**: Contains the main functions for file transfer.
- ❖ **function.py**: Contains functions that need to be reused such as traversing folders, generating lists, etc.

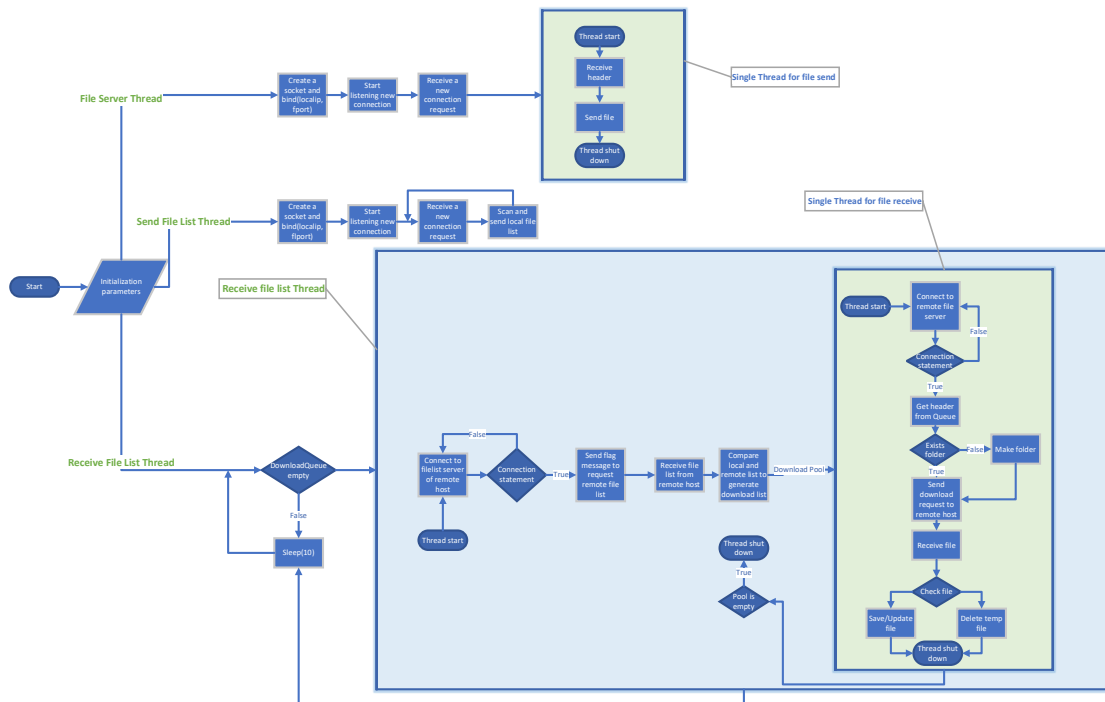


Figure 3: Activity diagram based on Design_v2.0

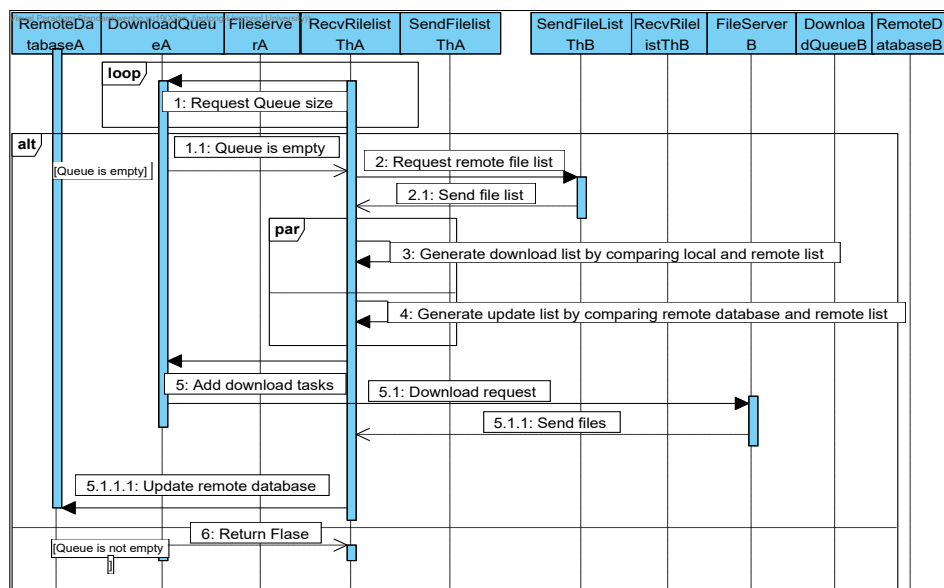


Figure 4: Sequence diagram based on Design_v2.0

4.2 Difficulties and solutions (programming skills)

This section will analyze several key obstacles and discuss solutions based on the following charts.

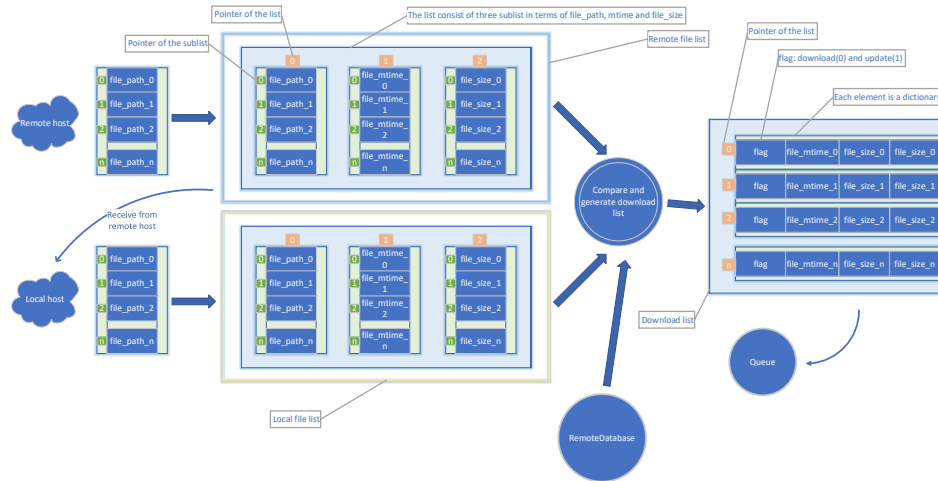


Figure 5: Data structure of file list and file header

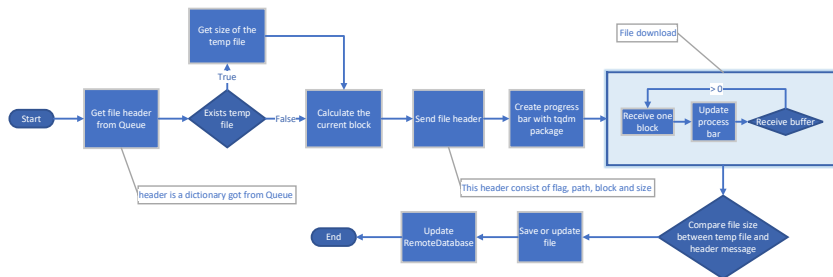


Figure 6: Detailed process of file receiving

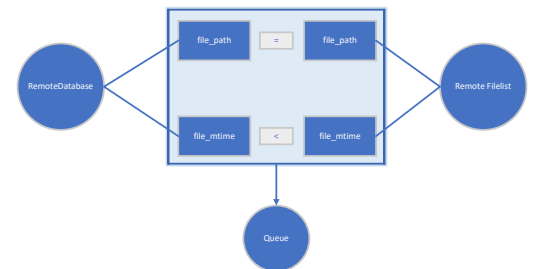


Figure 7: RemoteDatabase Update

❖ File list and header

The data structure of the file list and the file header is indicated in Figure 6.

❖ Breakpoint retransmission




Transferring in chunks and creating temporary files while downloading files allows for breakpoint transfers by detecting blocks of files received after the program is unexpectedly closed, which has been shown in Figure 6.

❖ File update

'RemoteDatabase' will be updated at the end of each file transfer, and each time the remote file list is received it will be compared and the files that should be updated will be retrieved as shown in Figure 6 and 7.

5. Testing and results

5.1 Testing environment and plan

- ❖ **Test environment:**  Manjaro Linux + VirtualBox ( Ubuntu Linux and  tinycore linux)

The program was developed in an environment where the host computer simulated PC_A and the Ubuntu virtual machine simulated PC_B, and the test script was used as the final test standard.

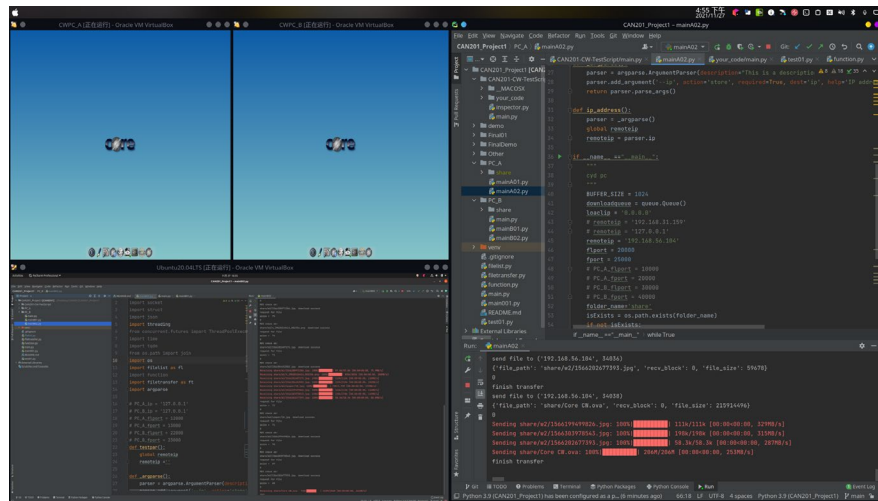


Figure 8: Screenshot of test environment

- ❖ **Test plan**

Plan	Result
Test the stability by running only PC_A to see if there is an error when there is no connection answer for a long time.	pass
Turn PC_A off and restart on the way to transfer large files to test breakpoint transfer.	pass
After ending a transfer, add some new files to PC_A's <code>./share</code> folder and observe if the transfer is restarted between PC_A and PC_B.	pass
After finishing a transfer, modify some of the files in the <code>./share</code> folder of PC_A and check the results in PC_B.	pass

Table 1: Test plan and result

5.2 Results and discussion

This section will analyze the impact of several important parameters of the program on the efficiency in terms of the execution results of the test script.

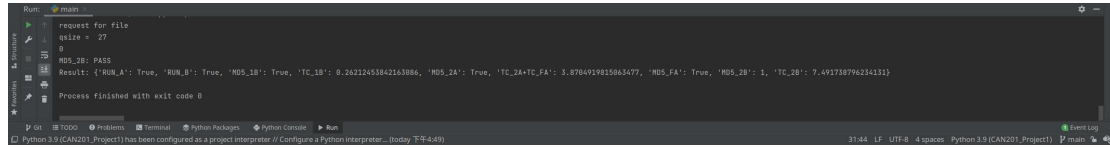


Figure 9: Screenshot of test result by test script

<pre> start file transfer to ('192.168.56.101', 37230) {'flag': 1, 'file_path': 'share/folders/fxx_13.txt', 'recv_block': 0, 'file_size': 36} MD5_2A: PASS Receiving share/file2.ppt: 82% #####4 409M/500M [00:02<00:00, 107MB/s] Sending share/file2.ppt: 83% #####3 416M/500M [00:02<00:00, 108MB/s] </pre>	<pre> Sending share/file1.bin: 0% 0.00/10.0M [00:00<?, ?B/s] finish transfer SIZE check ok! share/file1.bin download success request for file qsize = 36 0 Sending share/folders/fxx_29.txt: 100% ##### 1.00k/1.00k [00:04<00:00, 241k] Receiving share/folders/fxx_17.txt: 100% ##### 1.00k/1.00k [00:04<00:00, 241k] Sending share/folders/fxx_17.txt: 100% ##### 1.00k/1.00k [00:04<00:00, 241k] Receiving share/folders/fxx_23.txt: 100% ##### 1.00k/1.00k [00:04<00:00, 241k] Sending share/folders/fxx_23.txt: 100% ##### 1.00k/1.00k [00:04<00:00, 241k] Receiving share/folders/fxx_36.txt: 100% ##### 1.00k/1.00k [00:04<00:00, 241k] Sending share/folders/fxx_36.txt: 100% ##### 1.00k/1.00k [00:04<00:00, 241k] Receiving share/file1.bin: 100% ##### 10.0M/10.0M [00:04<00:00, 2.47M] Sending share/file1.bin: 100% ##### 10.0M/10.0M [00:04<00:00, 2.47M] Receiving share/folders/fxx_5.txt: 100% ##### 1.00k/1.00k [00:04<00:00, 246k] Sending share/folders/fxx_5.txt: 100% ##### 1.00k/1.00k [00:04<00:00, 246k] Receiving share/file2.ppt: 100% ##### 500M/500M [00:04<00:00, 126M] Sending share/file2.ppt: 100% ##### 500M/500M [00:04<00:00, 126M] Receiving share/folders/fxx_7.txt: 100% ### 1.00k/1.00k [00:00<00:00, 10.1k] </pre>
---	--

Table 2: Screenshot of testing log

❖ The effect of the maximum number of download threads on efficiency

The graph below shows the relationship between the maximum number of file transfer threads and the test results. It can be shown that there is almost no improvement in the efficiency of multi-threading on a single-core VM.

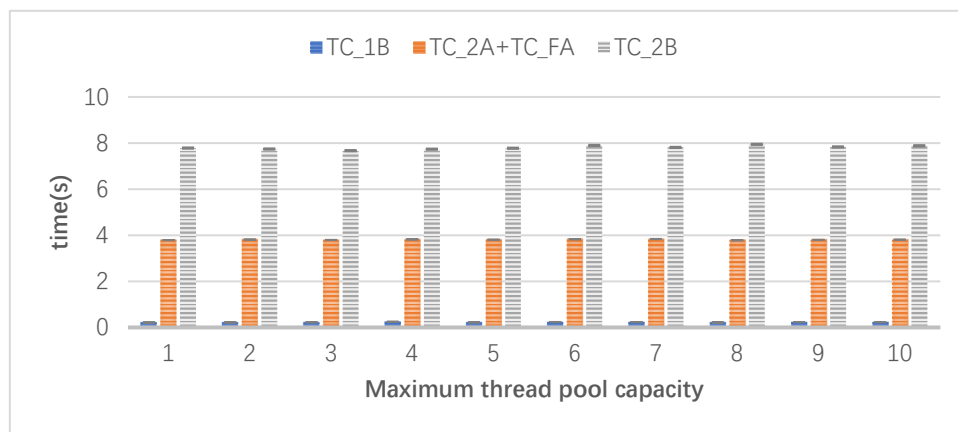


Figure 10: Maximum number of threads versus execution time

❖ The effect of buffer size on efficiency

The following curves indicate the relationship between the buffer size at the receiver side and the test results, which shows that an increase in buffer size is noticeable on the positive performance improvement in the lower range.

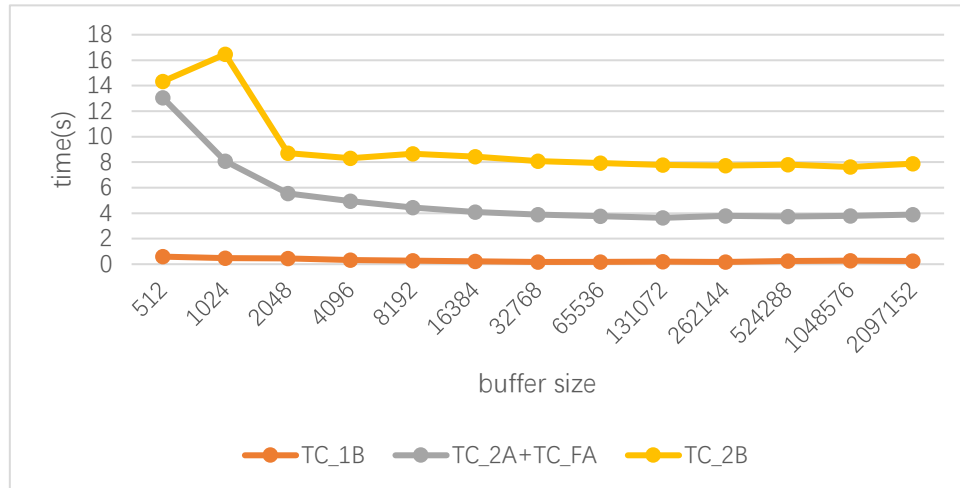


Figure 11: Buffer size versus execution time

6. Conclusion

This project implements the construction of a TCP socket-based file synchronization protocol through Python, which expands the practice of computer networking while improving the ability to develop multi-threaded applications.

7. Reference

- [1] C. C. Marshall, T. Wobber, V. Ramasubramanian, and D. B. Terry, "Supporting research collaboration through bi-level file synchronization," in *Proceedings of the 17th ACM international conference on Supporting group work*, 2012, pp. 165-174.
- [2] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach, eBook, Global Edition*. Pearson Education, 2018.