Xi'an Jiaotong-Liverpool University

西交利物浦大學

Large Efficient Flexible and Trusty (LEFT) Files Sharing

Author Longbin Ji

Student ID 1927852

Module CAN201-Networking Project

Teacher Dr. Fei Cheng

Date 28th/Nov/2021

Abstract

With the development and highly usage of networking technology, file sharing applications has been widely used based on network to realize complex file transmission task. This python based program is aimed at handling Large Efficient Flexible and Trusty file sharing via two computers. In this program, I implement two main threads to detect new files for broadcasting and handle communications between peers to achieve sending / receiving / and updating functions. Additionally, in the testing environment, my code can successfully transmit 500MB file within 10 seconds, realize breakpoint resume and synchronization for files and folder efficiently.

Introduction

1.1 Background

With the development of mature network system around the world, file sharing applications have been popular to offer convenient file transfer service for people. As has been learned in class, TCP containing rdt service and different kinds of retransfer methods is able to ensure there is no loss for the tranfering information. Based on TCP, it is possible to create my own protocol using python socket programming to complete a tiny file sharing program.

1.2 Literature review

As can be seen in daily network use, there are many mature file sharing applications like Baidu NetDisk, iCloud to provide this service from the help of the strong function of network layer (and sub layers) protocols. According to the study of W.Huang[1], it realizes working out a mechanism and protocols to create a monitorable P2P application. Thus, by designing proper mechanism in peers communication and useful protocols, it is possible to achieve creating a program handling the LEFT transmission of file between computers.

1.3 Task requirements and My work

This program is aimed at handling LEFT (Large Efficient Flexible and Trusty) file sharing task. Large refers to 500 MB size file and file folders. Efficient requires fast transmitting speed for file. Moreover, Flexible requires breakpoint resume mechanism. Trusty means there is no loss

in the transmission.

To achieve the requirement, I create a program based on TCP protocol with two main threads to detect the new files for broadcast and handle the communication between peers for sending and receiving.

Methodology

2.1 Protocol

The protocol I used for the program is mainly being used for the communication thread to know what to do after accept headers from peer. Except from detecting new files from share folder, all the sending and receiving functions are only being operated after receiving the operation code in headers. Figure 1 shows the detail information of my protocol.

The protocol can be divided into two parts:

- **Operation code**: different code means different actions
- **File information**: the basic information of files such as: name, file size, write position.

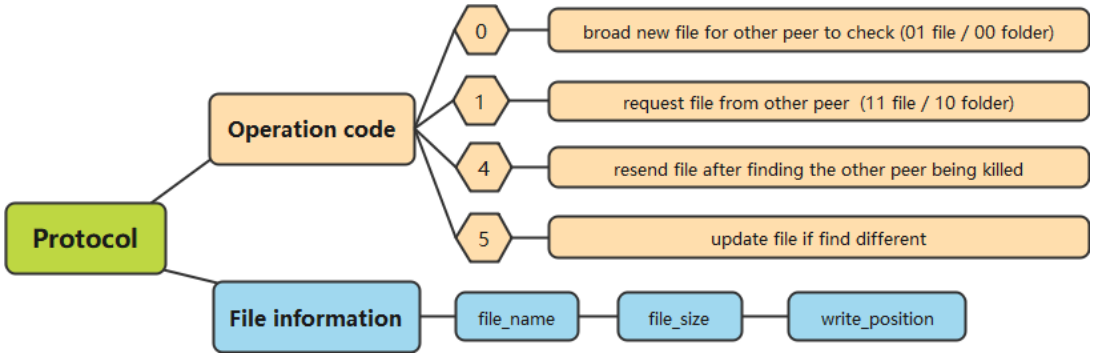

Figure 1: Information in protocol

2.2 Basic idea with functions

The basic ideas of the networking program:

- **Breakpoint resume:** By using try-except mechanism, the sending part of the program can detect whether the peer has been killed. Then the sender will send a header with operation code 4 to start resend process. Figure shows the detail of breakpoint resume.

- **Low coupling design:** Each function can individually create a socket to send or receive in specific port. When the program need to receive or send file, it only have to tell port, filename. These will simplify the programming process and reduce bugs.
- **Using different ports:** The program will use different ports for single file and files in one folder. Moreover, the program will start a testing socket in specific port to let peer know whether it start. This can help to solve 'port all already in use' problem.

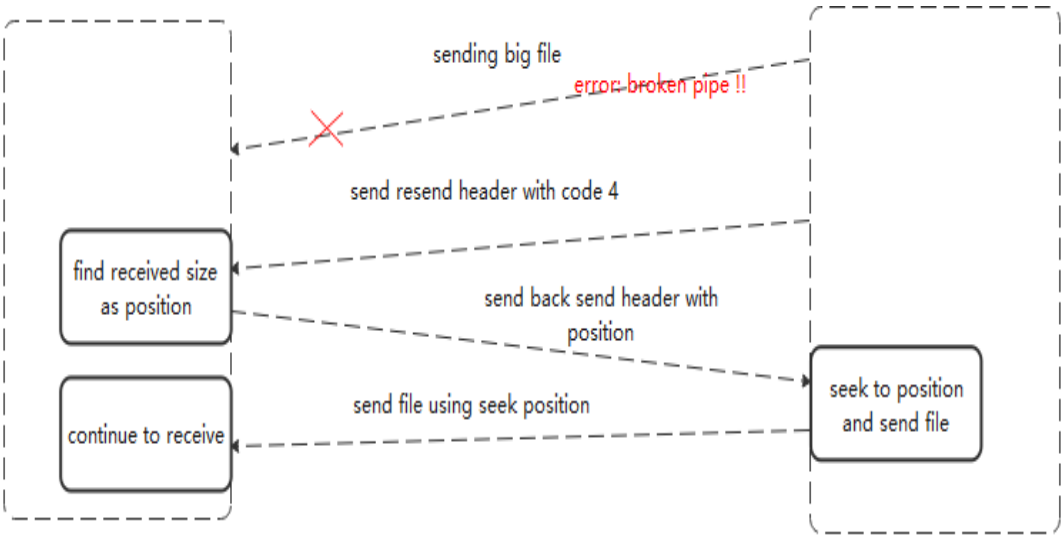

Figure 2: Break point resume process

The program has **3** main thread functions as shown in figure 3:

1. Keeping detect whether there are any new files in the folder. If there is new folder, broadcast to the other.
2. Receive thread with one server socket which will be always bind on communication port to handle communication from the peer. After received the header from other peer, will send/receive/resend/update file depends on the operation code in the header.
3. There is a test port having a socket binding on for other peers to test whether it is online.

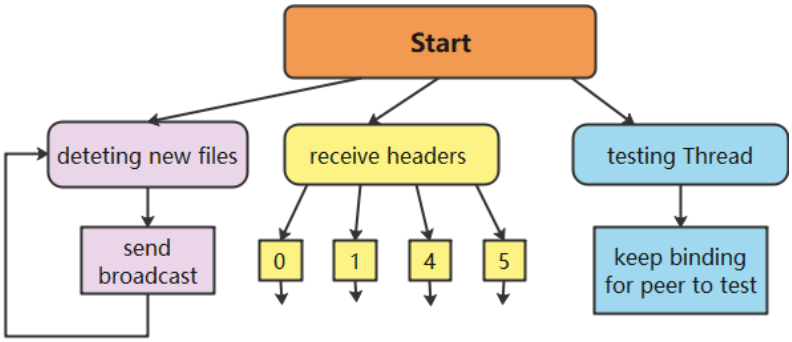

Figure 3: Three main thread

Implementation

3.1 Steps of implementation

1. Global design: Think about the whole design of the project to satisfy the needs of the task and other basic needs as a python program.
2. Detailed design: Think about the design of the steps in individual thread to realize the task requirement.
3. Implementation in programming for a whole structure of 3 threads.
4. Start to realize different functions step by step in programming.
5. Debug in testing environment. Make several changes in the design of program.

3.2 Flow chart of program

The figure 4 is the whole flow chart for two main thread of my program. It can be seen that, each thread is a self looping function. The detect thread will keep detecting whether there are new files. At the same time, the receive thread will keep receive headers from the other peer and make response depend on the operation code. Then functions of send file/folder, receive file/folder, update file, resend file will then be called to finish the tasks of each file sharing process.

The programming skills being used:
OOP (Object oritented programming) which been used for the management of sockets and for individually design of server and client functions in classes.

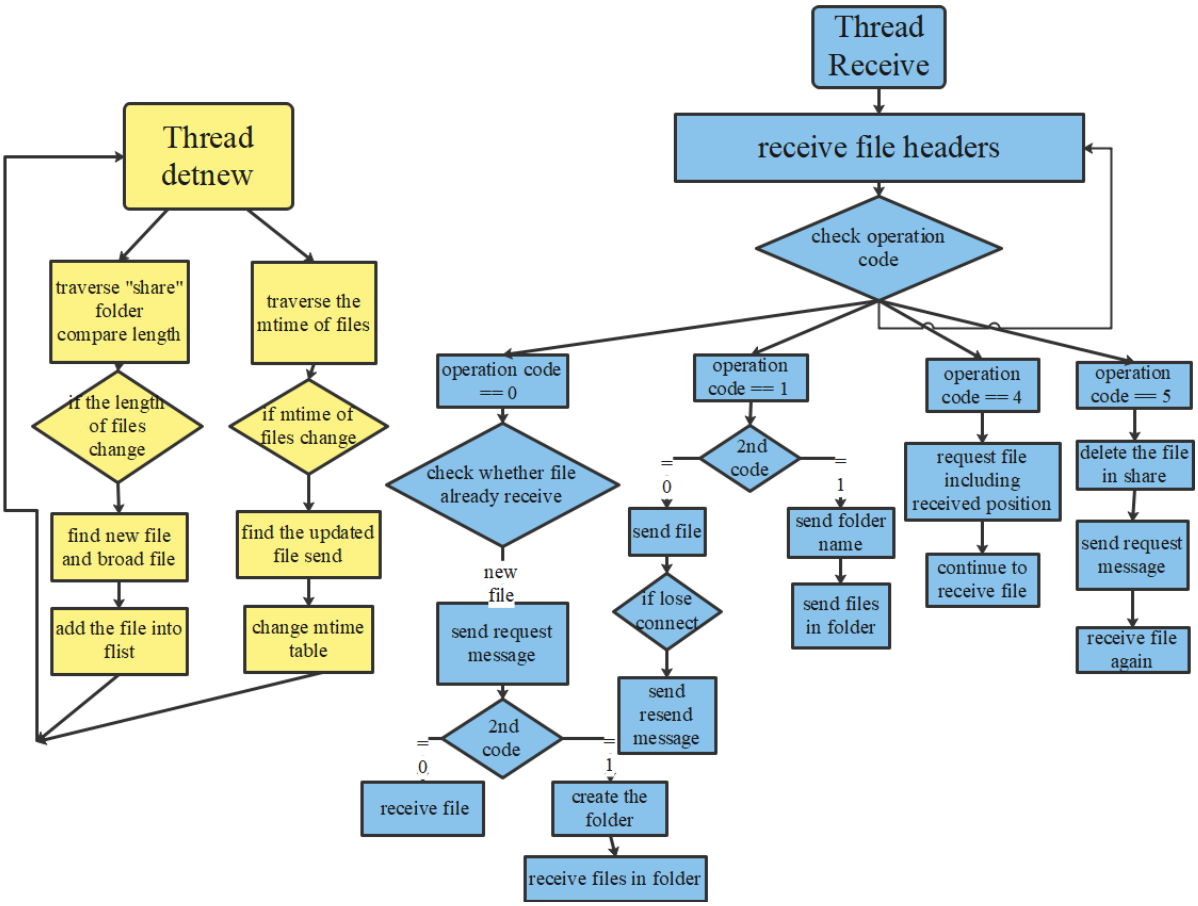

Figure 4: Flow chart for program

Multi-thread design: The program is based on three main threads to individually process different aspects of task as a whole. The detect thread is for the detecting of new files, and the communication thread is for receiving headers and handling the communication part. Using multi-thread, it simplifies the structure of my program and realize more complex tasks.

3.3 Problems encountered

Infinite retransmission: The problem occurs because the program which only detect new files by itself instead of exchange file lists can not tell if it's a synchronized file or a new file. To solve this problem, at the receiving stage, the file will have a '120' prefix and the origin name will be added to local file list.

Port already in use: It occurs that by using only one or two port, because the socket in specific port has to be handling huge task which costs lot of time, the interruption may not close the binding socket. To solve this, I use more different ports and make the always binding port can only being used to receive headers. Moreover, before bind function, I will make the binding

port for socket reusable for future rebind action.

Testing and results

4.1 Testing environment

Through the testing period for my code, there are three stages of testing environment:
Initial test: Using one computer to test the basic logic and grammar of my program step by step using two different versions.
Real network test: Using two computers connected to the same LAN. To check whether there is any loss or other problems with my code considering real condition. (Time consuming, logic sequence order in real network environment)
Final score testing environment: using the oracle VM box application to create two virtual machines in one computer with two IP addresses. Check the md5 of every file and record the time consuming.

4.2 Testing plan

Precondition: two computers (PCA,PCB), one main python file which can be run.

1. Run the code on PCA and PCB successfully.
2. Putting one file in the PCA to check if the file can be shared to PCB.
3. Putting one big file (500MB) and folder to PCB.
4. Kill code in PCA and restart to test breakpoint resume then check the file and folder md5 in PCA.
5. Update one file in PCA and check whether can be synchronized to PCB.

The testing plan can successfully test whether the program can handle different kinds of file sharing , file resume in interruption and synchronization in updating.

4.3 Testing result with improvement

Memory occupy problem: Figure 5 reflects the memory occupy problem in while true loop in specific thread. Without the sleep() function, the thread loop itself too fast which leads to

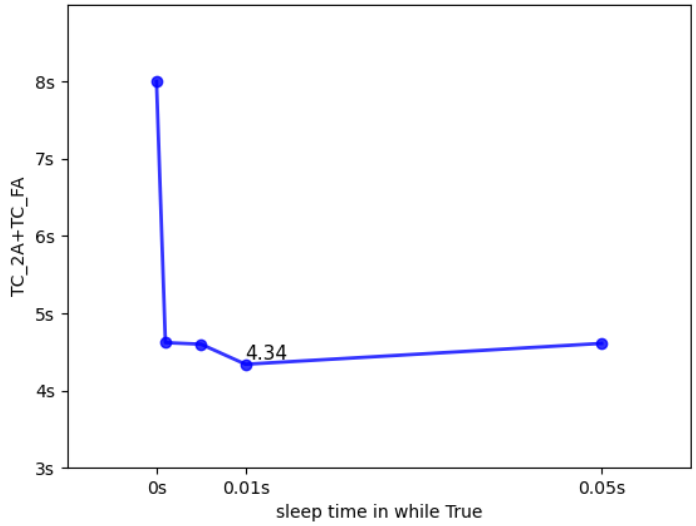

Figure 5: Thread memory occupy problem

serious memory occupy. It finds out that sleep 0.01s can increase the big file transmission speed to 4.3 seconds.

Buffer size too small: Through the testing period, I find that too small buffer size in receive effects the speed for big file transmission. As been shown in figure 6, 2048 or 20480 is too small which are decreasing the speed. After testing, I find that 2048*1000 can achieve good performance (highest speed) in big file transmission.

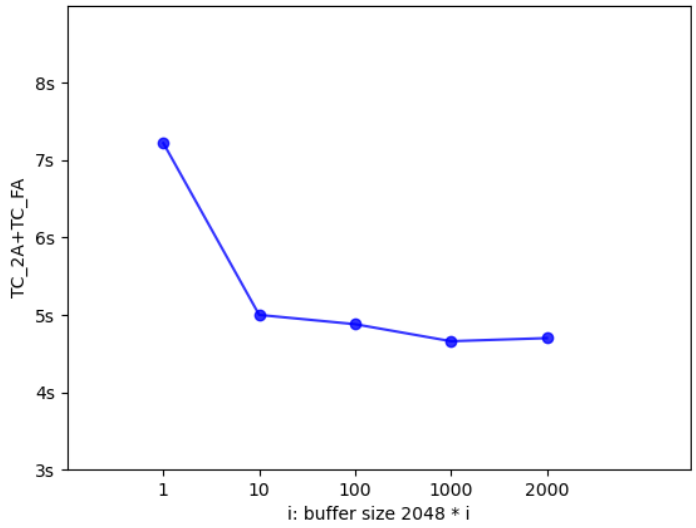

Figure 6: Buffer size problem

Final testing result: Through multiple times of testing and improvements in different aspect, figure 7 shows the average running result of my program.

Figure 7: Final testing result

Conclusion

In this network project, I successfully design and implement a python based program which can satisfy the need for LEFT file sharing in two computers using socket and multi-threading. Moreover, dependent on my program, I also design my own protocol for header communication. Through the running of the program, it can detect new manually added files with folders in share folder and transmit the file in fast speed using TCP protocol to other peer.

However, in the improvement phase, I failed to using multi-threading to create multiple threads for the transmission of file blocks in one file to increase the speed for big file transmission. The program seems to be skilled because too many thread running parallelly. In the future, I will try to implement multi-thread to handle the big file transmission.

References

[1] W. Huang, L. Yeh and J. Huang, ”A Monitorable Peer-to-Peer File Sharing Mechanism,” 2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS), 2019, pp. 1-4, doi: 10.23919/APNOMS.2019.8892963.