CAN201 Introduction to Networking
Coursework 1
Large Efficient Flexible and Trusty (LEFT) File Sharing

Student Name: Yuxuan.Ren

Student ID: 1823678

Large Efficient Flexible and Trusty (LEFT) File Sharing

Abstract

Nowadays, file sharing is very common and has a lot of related applications, such as Baidu Net-Disk and Dropbox. It can provide people with file synchronization function among different devices. The purpose of this report is to give a method to design and implement large efficient and trusty (LEFT) file sharing using Python Socket network programming.

Keywords: file sharing, LEFT, Python

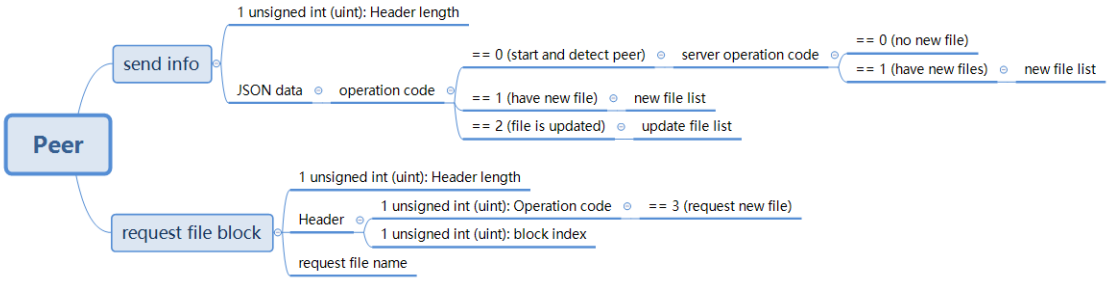
1. Introduction

File sharing is the active sharing of a person's computer files over the network. Generally, file sharing uses peer-to-peer mode, and the files themselves are stored on the user's personal computer. Most people who participate in file sharing also download shared files provided by other users. Sometimes the two actions go hand in hand. Dropbox, which is one application of file sharing, can be used as a mean to share course outlines, bibliography and academic requirements which is based on its ability to share files [1]. However, even the most popular file sharing applications can have security problems [2]. The aim of this project is to provide a method to design and implement large efficient and trusty file sharing using Python Socket network programming. The project should be able to share a file up to 1GB in any format in a high speed with no error. In addition, it should be able to resume from interruption, synchronize partially and provide data transmission security. The remainder of this report is organized as follows. Section two discusses the methodology. Section three shows how the project is really implemented. In section four, a testing plan and its results will be given to show the performance of this project. Finally, the report will end with a conclusion.

2. Methodology

2.1. Proposed protocols

Operation code 0 for greeting, operation code 1 for informing new files, operation code 3 for informing updates, operation code 3 for requesting file blocks.



2.2. Proposed functions

Each peer can be seen as a combination of local functions, client functions and server functions. For local functions, the aims for these functions are to reflect local information, including local files information, local changes and peer changes. The local information should be detected from time to time use an infinity while loop so that any change can be found at once. In addition, some simple functions for getting file size and certain file block are also needed. For client functions, when some local new files or updated files are detected, this peer needs to inform other peers, and when peer changes are detected, this peer needs to request the new files or the changed blocks. In addition, when the program starts, it needs to say 'Hello' to other peers. For server functions, when receiving information from peers, it needs to update the local variables to reflect peers' changes, when receiving requests from peers, it needs to give the file blocks they want. Also, it needs to response to other peers' greeting.

After all these functions are implemented, the data transfer security should be concerned using encryption techniques.

2.3. Ideas

1. Use several global variables. Local functions are used to maintain these variables. All interactions are based on the changes of these variables. When the variables for local new files or local update files change, the peers need to inform each other and update the

variables for maintaining the variable for new files or updated files from peer. Based on the changes of these variables, peers request files from other peers.

2. When greeting and responding to other peers' greeting, a peer should send what it has now.
3. Use two sockets to separate normal information exchanges and file blocks transmissions.
4. Fundamental functions on two peers should firstly be implemented, then the program can be expanded to support three peers.
5. AES is an symmetric encryption technique that can be easily used in Python.

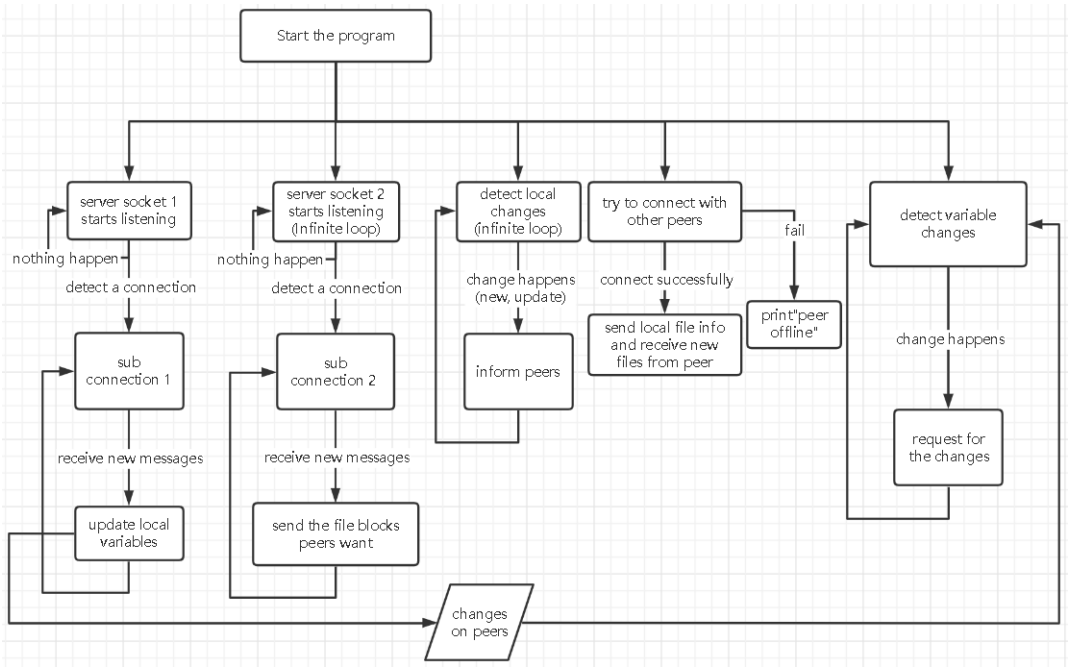
3. Implementation

3.1. Steps of implementation

The variables used are total_file (all local files the peer have), new_add_file (new files in local folder), new_update_file (the updated files in local folder), new_file_from_peer (new files information received from peers) and new_update_from_peer (updated files information received from peers). The main part of implementation is divided into three steps: 1) Implement the local functions. 2) Assuming that the server functions have already been implemented, implement the client functions based on the protocols. 3) Implement the server functions. The first step (local methods): The detect_change method is to detect every local changes, and then try to inform the peers. The file changes in this project are new added files and new updated files. total_file_after_detect is used to store the file in the 'share' folder now. For new added files, they must be recorded in total_file_after_detect, but not in total_file. The method should find them and add them into new_add_file. For new updated files, their current last modified time must be different from the last modified time stored in total_file. The method should find them and add them into new_update_file. The update_new_file_from_peer method is called when the peer receives new information which means new or updated file defined by the protocol from other peers. According to the file list received, the peer updates its new_file_from_peer list or ask for further transfer (In this project, it is done only if interruption happens). This method can be classified into server functions, but to make server simple, it is regarded as local functions. The detect_new_file_from_peer method is to detect whether new_file_from_peer and new_update_from_peer are empty. If they are not empty, request new files or just request some blocks. The second step (client methods):

The detect_peer method is to say 'hello' to others (try to build TCP connection between each other and inform peers the new files). The inform_new_file and inform_update_file methods make relevant messages for informing other peers the changes. The request_new_file_from_peer, request_update_from_peer and further_transfer methods are responsible for requesting new files or updated file blocks or last several blocks (for resuming). The third step (server methods): Start two server sockets to listen to each peer. One for information exchange (I use JSON and struct), the other one for data transfer (I just use struct). Once accepting a new connection, create a new thread to maintain this connection. It will deal with different types of messages using different local methods. For this project, although the data transfer socket only needs to send the required blocks, in order to increase future functional scalability, I also assigned it an operation code.

3.2. Program flow charts



3.3. Programming skills

In this project, the programming skill being used mostly is parallel. Since there are many infinite loops, for example, detecting changes, receiving messages, the only way to do this simultaneously is to use the parallel methods. Both multithreading and multiprocessing have been tried before, multithreading is finally chosen because it provides a better performance after many tests.

3.4. Difficulties

There are many difficulties during this project.

1. When the file is being modified, the last modified time always changes. This will lead to misunderstanding.

   Solution: Before the file is modified, I clear its information in total_file and replace it with an integer 1. If the file is a new file, just insert it into the total_file and assign it with an integer '1'. If the information of a file in total_file is only a '1', the detect_change method will know that this file is being modified, caring about it now makes no sense. After the modify finishes, its information should be updated into total_file variable at once. This action has two meanings: 1.The write operation is done. 2.It is a file get from other peers, not a new file.

2. I implement the project for two peers first, then expands the codes to satisfy three peers. For two peers, I have two client sockets and two server sockets for each peer. If there are three peers, the total number of sockets for each peer will be six and there will be a lot of redundancy if I do it in the same way for implementing two peers.

   Solution: Use a dictionary to store the four client sockets and a list to store the ports I can use. Then I can create sockets using for loop (pop a new port number from the list each time). In addition, the parameters for many methods can be addresses instead of sockets. It is more clear and less redundant.

3. If every message just contains the content without providing the length of this message using TCP, possibly the receiver will receive more data than one message. Then error occurs.

   Solution: Each time, send a length, which is 4 bytes length, together with the message, the receiver will first receive 4-byte data to get the length of the message. Then it will receive the data according to the length.

4. I judge whether a file has been updated according to the last modified time. But at first, I used "not equals". Then a problem occurs. After a peer gets new file from other peers, this file will be judged as being updated.

   This problem is very puzzling. It is caused by a time difference which does not take into account. When comparing the last update time happens, the program just updates the

information into total_file, so the last modified time in total_file is a little bigger than the last modified time in total_file_after_detect. Misunderstanding occurs. To solve this, just use "larger than", because true last modified time is always the latest.

5. I use the method of detecting whether the variable is null to determine whether a file needs to be requested from peer, which is very easy in the case of only two hosts. But once the number of hosts becomes three, the situation is very complex. Two threads are working at the same time (for request file from each peer), there is a certain probability that the peer will ask for the peer without this file or without the updated block because the peer is only told what file he needs to have but no information about from who, then error happens.
   To solve this problem, if one peer receives new file information from another peer, attach the peer's IP address together when adding the file information to the list or dict. For the threads, they will visit the variables, find the elements that have the same IP address as their parameters and request them.
6. In the case of not using encryption, my approach for receiving new data block is to write in as much as received. It can ensure efficiency. But when encryption mode is on, since I use AES, the length of each plain text should be the multiple of 16, or there will be an error.
   Solution: If the size of the remaining file that needs to be received is bigger than one block size (2MB for this project), it will use a variable to store all the data until the size of it is 2MB. If If the size of the remaining file that needs to be received is smaller than one block size, assuming that the size is a, it will try to receive and store data with size equals to math.ceil(a/16) * 16 and then decipher and write into the file.

4. Testing

4.1. Testing plan

The test environment is three virtual machines with Tiny Core Linux. Its memory size is 512 MB. Before test, A 8GB disk should be mounted. The three peers can be called A,B and C. The test plan can be divided into five parts. The first step is trying to share a 10MB file (put it on A) to make sure that fundamental file sharing function can run successfully. Then, the second step is to share a 1GB big file and a folder with 50 small files (put them on B). Two seconds after the files are ready on B, peer A will be killed and restarted again after C has successfully received

all these new files (this interruption can be done for several times). This step is to ensure that interruption resuming and sharing folders can be done without any problem. The 1GB file is a txt file with random numbers (first 1KB) and 0 (the rest). After that, a 200MB file (txt file) will be shared (put it on C). Then the first 0.1% of that file will be modified on one virtual machine (This can be done for several times on different peers). The other two should immediately detect this change and synchronize this change. After that, the md5 of every file will be checked. All of these files are put into the folder using soft links. Finally, the encryption part. If all the first three parts can be done successfully without any error when encryption function is on, it is implemented successfully. Wireshark will also be used to capture some massages.

4.2. Testing results

The following is a table shows the testing results for 10 times without encryption and the screenshots of one test. The speed for sharing 1GB file and 200MB file is not proportional. It is because when sharing the 1GB file, one peer is killed, which means that the burden of the bandwidth is much lower than when sharing the 200MB file. In addition, the longest data is always obtained in the first test after starting the virtual machine. It may be caused by the time used for resource allocation.

	shortest(s)	longest(s)	average(s)
10MB	0.27	0.37	0.31
1GB	23.4	26.3	24.8
200MB	6.3	11.8	10.7

Figure 1: 10MB

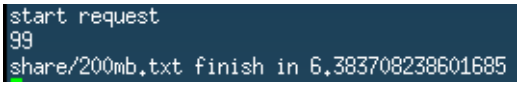


Figure 2: 200MB

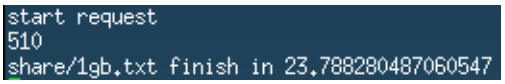


Figure 3: 1GB

5. Conclusion

In conclusion, the aim of this project is to design and implement a reliable file sharing function which can share big file, resume from interruption and synchronize undated files without re-transferring the whole files again, which can also be called LEFT file sharing. This project can be an efficient way to understand the basic knowledge behind the currently popular file sharing applications. For this project, there are many improvements I can make in the future: 1. Try to use the asymmetric encryption algorithm 2. For this project, the disk space is big enough so that I don't need to consider the problem that whether there is enough space for a file. To make the project more practical, it is better to use a placeholder to occupy enough space before transferring. 3. Get more information about error recovery without re-transmission. 4. If peer A and peer B both have one file, when peer C is restarted, he needs to get this file, then, he can get even blocks from A and odd blocks from B which can make transfer even more quickly. To make this project more generic to solve any problems may meet in real life usage, there is still a long way to go.

Reference

[1]J. Moreno, "USING SOCIAL NETWORK AND DROPBOX IN BLENDED LEARNING: AN APPLICATION TO UNIVERSITY EDUCATION", Business, Management and Education, vol. 10, no. 2, pp. 220-231, 2012. Available: 10.3846/bme.2012.16.

[2]Cheng-Kang Chu et al., "Security Concerns in Popular Cloud Storage Services," Pervasive Computing, IEEE, vol. 12, no. 4, pp. 50-57, October-Desember 2013.