



Xi'an Jiaotong-Liverpool University

西交利物浦大学

CW1 LEFT File-sharing System

Author Dongheng Lin

Module **CAN201**

Teacher Fei Cheng

Date 28th/November/2021

CAN201 – Introduction to Networking Assignment 1

LEFT File Sharing System (2021-22)

Student: Dongheng Lin

ID Number: 1929066

Abstract

As the given requirement of the coursework, students are requested to develop a Large Efficient Flexible and Trusty (LEFT) Files Sharing network realizing reliable and swift transmission between 2 peers and being robust when encountering interruption. This report will represent an approach using TCP connections and basic P2P models with only 2 peers.

1 Introduction

1.1 Project requirement

The coursework requirements are:

1. Capable of transferring files larger than 500MB
2. Be sensitive to all kinds of files within a given sharing folder, including cascading files and hidden files.
3. The application should respond and act quickly.
4. It should take the IP addresses as arguments so as it can be implemented on any machine with different ipv4 addresses.
5. Also, it must be able to restore the transmission when one of the peers is shut down or the process is killed.
6. It should be trusty, in which no errors happen to the transmission and execution.

1.2 Background

Nowadays, with more and more internet devices, the file-sharing system has become essential for many companies and individuals [1]. Many applications with file-sharing functions are developed and widely used, for example, Dropbox, Airdrop, and Baidu Netdisk, etc [1] [2]. However, developing and implementing one's own application could be helpful in improving one's knowledge in networking as well as having a much flexible and customized file-sharing system [3]. This report will be a good description of a student's approach to a file-sharing system.

2 Methodology

2.1 Overall Structure

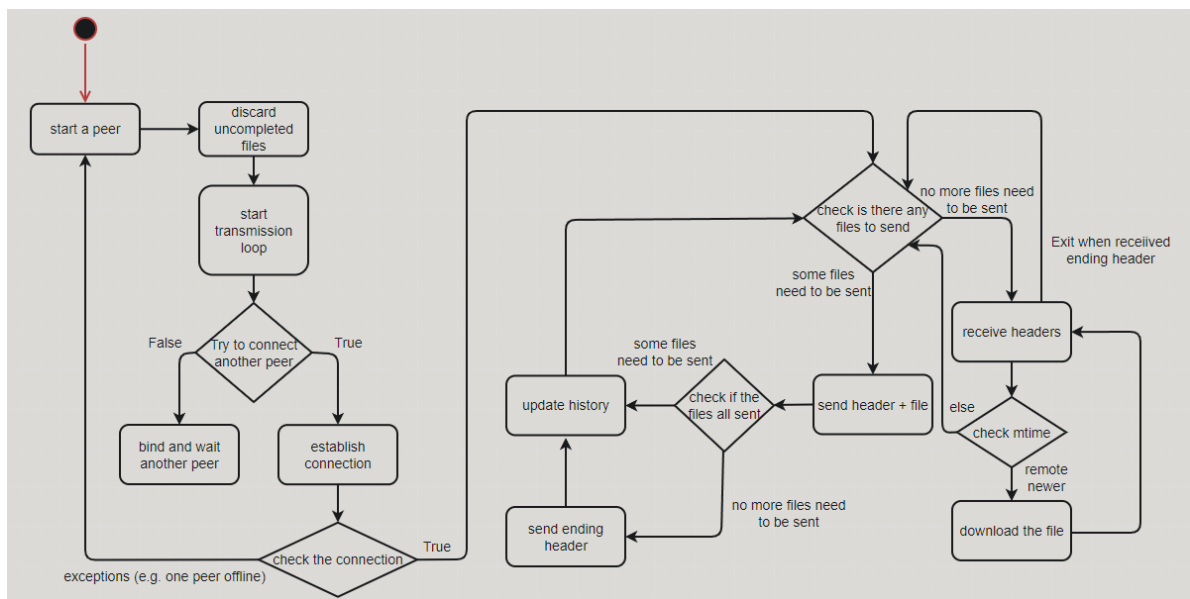


Figure 2.1.1 Mind map of the application

2.2 General Ideas

Instead of exchanging local information every time when another peer online by sending messages to judge which files to send, this application may just send all the files that have not been sent before or have been modified by checking a history dictionary containing modification time and letting the

other peer decide whether it is the file to receive by checking the file headers. In that case, the structure of this application is quite simple, however, some resources might be wasted when the file being sent is older than the files which are claimed by the remote peers.

2.3 Connection Establishment

Every time when a machine becomes alive, it may automatically check whether the socket is vacant or not, if so, the peer may try to create the socket and bind the port as the first peer, else to be the second peer by connecting to the existing socket. When one of the peers disappear, the exception may be raised to inform the applications and try to refresh the socket in another loop.

2.4 History dictionary

The history is quite crucial in this application, the application may check it to determine whether the files should be sent or not. If a file is already in the history dictionary and no modifications change its “mtime” variable, the file would not add to the sending queue. This dictionary is realized by python class and store in a local .json file in order to record the dynamic history and can be read again after the interruption. However, the history may need to be refreshed from time to time to prevent memory consumption.

2.5 Header and files

In this scheme, every file is sent with a header containing the size, modify time and path name of the file. The structure of the header is like:

<i>Index</i>	<i>0 - 4</i>	<i>4 - 8</i>	<i>8 - 16</i>	<i>16 - 16 + file name size</i>
<i>Content</i>	<i>File size</i>	<i>File name size</i>	<i>Modify time</i>	<i>File name</i>

Also, a special header is defined to inform another peer to stop receiving by setting the path size to 0.

2.6 Sender function

As mentioned above, the sender would fetch files do not exist in histories to avoid repetitive transmission and if a file is newer than the value in the history dictionary, the retransmission would be invoked by adding the file into the sending queue. The sending queue will update itself every 0.1 sec and submit the files to the socket like a pipeline.

The sending function will pack a header for each file within the sending queue. Whenever a file is sent, a header must be sent prior to this operation to inform another peer the corresponding file information.

2.7 Receiver function

When the sending queue of a peer becomes empty, it would switch to the receiving phase. The receiver will firstly try to receive and parse the remote headers. If the values are valid, it will try to download the files using header information. When the receive procedure is done, a checking mechanism will be activated to check the file size to avoid data loss or duplicate during writing.

However, sometimes, the receiving process may be shut down before it is done, to solve this issue, the application adopted a mechanism which is

common in many downloaders, the temporary files. Before the file is confirmed to be completely received the file will be named as “filename.temp”, which will be deleted when restarting the whole application so as to avoid mistakes of unaccomplished files. Finally, the file will be renamed to its original name and dequeued from the sending queue as it has been successfully sent.

3 Implementation

3.1 General implementation

<i>Function</i>	<i>Usage</i>
<i>main()</i>	<i>Initialize and start everything</i>
<i>discard_temps()</i>	<i>Scan and delete all the temporary files</i>
<i>traverse(path)</i>	<i>Scan all the files with in given directory except the folders</i>
<i>transmission_loop()</i>	<i>Try to establish links for different peers and then try to send files</i>
<i>firstpeer(socket)</i>	<i>When the socket is empty, bind and listen.</i>
<i>History(self)</i>	<i>Class to manipulate and load from history.json and thus to determine sending</i>
<i>send_and_receive(socket, history)</i>	<i>Using the history to decide whether a peer should send or receive every 0.1 sec</i>
<i>local_directory(self)</i>	<i>Class to query local file info, and thus determine whether to receive</i>
<i>checking_pipe(socket)</i>	<i>If the socket is closed, raise exception.</i>
<i>send_routine(history, path, socket)</i>	<i>When the sending queue is empty, try to send the files together with header and append to sent files into the history.</i>
<i>send_header(path, socket)</i>	<i>Send all the header</i>
<i>make_header(path)</i>	<i>Make a header with (16+path size) length</i>
<i>recv_file(socket, history)</i>	<i>Try to receive files when there is no file to send for a peer, it will return a Boolean to indicate whether the receive is complete.</i>
<i>parse_header(header)</i>	<i>Before start to receive, the receive should check the header to get necessary information or end the phase of receiving.</i>
<i>write_files_and_folders(path)</i>	<i>When the path is in a sub-directory, make a sub-directory and then write into it.</i>
<i>downloader(socket, received_bytes, path, file_size, modifytime)</i>	<i>Try to download the files with the temporary file path, and when the download completed the file size will be checked to avoid data loss. After everything is done, the file will be renamed to its original name.</i>

3.2 Difficulties and solution

3.2.1 Structural design

The application is based on countless failures of design. Firstly, the application scheme was based on natural thinking of obtaining requesting file lists from the comparison between remote file information messages and local information recursively on different threads.

However, the implementation of this structure soon become impossible because too many messages generated by this structure had made the socket become so busy and the performance was a total disaster. It can somehow run without too many mistakes on 2 robust machines but act poorly on testing virtual machines.

Thus, the new structure decides to discard unnecessary interaction between peers (too many messages), but to focus on transmitting files and make the header become an identifier of corresponding operations for another peer.

3.2.2 Interruption restore

Some parts are encapsulated in the Python Class to ensure integrity by relocating to .json files to preserve the data after interruption, and thus the application can restore itself after being shut down.

3.3 Programming skills

3.2.1 Object-Oriented Programming

After the overall program is completed, some modification is made to enhance the performance. History and local file information are thus defined as 2 classes which can be easily queried and modified by invoking corresponding functions.

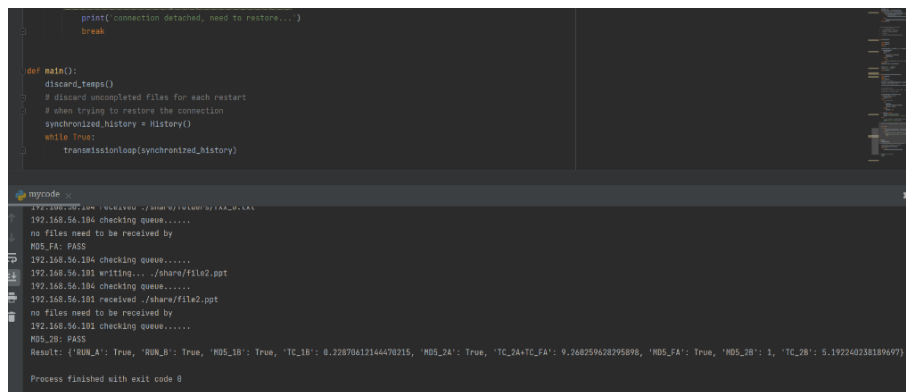
3.2.2 Modularity

The codes are broken into 2 files (“main” and “peer_activity”) and within each file, clear segments are divided for different purposes, like sender functions, receiver functions and PC configuration functions. This kind of coding structure has made the modification and debugging is quite efficient because it is so convenient to locate different function chunks and typos of codes.

4 Testing and results

4.1 Test scripts

Following the given test scripts and virtual machine environments, the using scenario of 2 linux machines are tested, and the testing results are as follows:

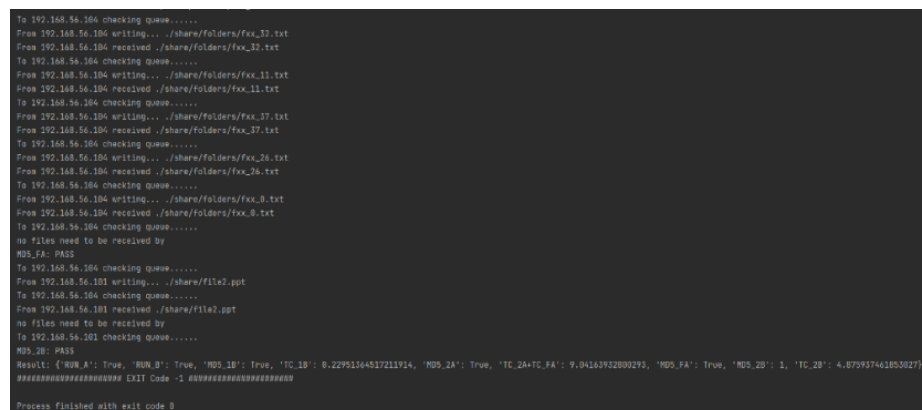


```
print(connection detached, need to restore ...)
break

def main():
    discard_uncompleted()
    # discard uncompleted files for each restart
    # when trying to restore the connection
    synchronized_history = History()
    while True:
        transmissionloop(synchronized_history)
```

```
192.168.56.104 checking queue.....
no files need to be received by
MDS_FA: PASS
192.168.56.104 checking queue.....
192.168.56.101 writing... ./share/file2.ppt
192.168.56.104 checking queue.....
192.168.56.101 received ./share/file2.ppt
no files need to be received by
192.168.56.101 checking queue.....
MDS_2B: PASS
Result: ('RUN_A': True, 'RUN_B': True, 'MDS_1B': True, 'TC_1B': 0.2287061214470215, 'MDS_2A': True, 'TC_2A+TC_FA': 9.268259628295896, 'MDS_FA': True, 'MDS_2B': 1, 'TC_2B': 0.192240238189697)
Process finished with exit code 0
```

Figure 4.1.1 test1



```
To 192.168.56.104 checking queue.....
From 192.168.56.104 writing... ./share/folders/fxx_33.txt
From 192.168.56.104 received ./share/folders/fxx_33.txt
To 192.168.56.104 checking queue.....
From 192.168.56.104 writing... ./share/folders/fxx_11.txt
From 192.168.56.104 received ./share/folders/fxx_11.txt
To 192.168.56.104 checking queue.....
From 192.168.56.104 writing... ./share/folders/fxx_37.txt
From 192.168.56.104 received ./share/folders/fxx_37.txt
To 192.168.56.104 checking queue.....
From 192.168.56.104 writing... ./share/folders/fxx_26.txt
From 192.168.56.104 received ./share/folders/fxx_26.txt
To 192.168.56.104 checking queue.....
From 192.168.56.104 writing... ./share/folders/fxx_0.txt
From 192.168.56.104 received ./share/folders/fxx_0.txt
To 192.168.56.104 checking queue.....
no files need to be received by
MDS_FA: PASS
To 192.168.56.104 checking queue.....
From 192.168.56.101 writing... ./share/file2.ppt
To 192.168.56.104 checking queue.....
From 192.168.56.101 received ./share/file2.ppt
no files need to be received by
To 192.168.56.101 checking queue.....
MDS_2B: PASS
Result: ('RUN_A': True, 'RUN_B': True, 'MDS_1B': True, 'TC_1B': 0.2295164537211914, 'MDS_2A': True, 'TC_2A+TC_FA': 9.04163932800293, 'MDS_FA': True, 'MDS_2B': 1, 'TC_2B': 4.075937461853027)
EXIT Code -1
Process finished with exit code 0
```

Figure 4.1.2 test2

5 Conclusion and some possible enhancements

In conclusion, even though the general approach accomplished all the requirements, its algorithm is quite brute-force and the performance is far from perfect. Perhaps it can perform better in speed by removing some useless flow-control and adding more complex interaction mechanisms to reduce unnecessary sending.

Also, it seems that the application does not ensure data integrity for it only checks the data size while the md5 value is unchecked. Although the TCP connections are quite reliable, it is essential for an application protocol to have some error detection of altering data in the application layer. It is possible to introduce compression and encryption methods to ensure privacy and integrity.

6 Reference

1. G. Goncalves, I. Drago, A. P. Couto da Silva, A. Borges Vieira and J. M. Almeida, "Analyzing the impact of dropbox content sharing on an academicnetwork", XXXIII Brazilian Symposium on Computer Networks and Distributed Systems (SBRC), pp. 100-109, 2015.
2. H. Wang, R. Shea, F. Wang and J. Liu, "On the Impact of Virtualization on Dropbox-like Cloud File Storage/Synchronization Services", Proceedings of the IEEE 20th International Workshop on Quality of Service, vol. 11, pp. 1-11, 2012.
3. F. Cheng, "CAN201-Project", 14 Nov 2021. 1