

# XI'AN JIAOTONG-LIVERPOOL UNIVERSITY

## 西 交 利 物 浦 大 学

### REMOTE OPEN BOOK EXAM ANSWER SUBMISSION COVER SHEET

Name	Zhou	Wenhao
Student ID Number	1824099	
Programme	Information and Computer Science	
Module Title	Introduction to Networking	
Module Code	CAN201	
Module Examiner	Fei Cheng	

By uploading or submitting the answers of this Remote Open Book Exam , I certify the following:

- I will act fairly to my classmates and teachers by completing all of my academic work with integrity. This means that I will respect the standards and instructions set by the Module Leader and the University, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance.
- I have read and understood the definitions of collusion, copying, plagiarism, and dishonest use of data as outlined in the Academic Integrity Policy, and cheating behaviors in the Regulations for the Conduct of Examinations of Xi'an Jiaotong – Liverpool University.
- This work is produced all on my own and can effectively represent my own knowledge and abilities.

I understand collusion, plagiarism, dishonest use of data, submission of procured work, submission of work produced and/or contributed by others are serious academic misconducts. By uploading or submitting the answers with this statement, I acknowledge that I will be subject to disciplinary action if I am found to have committed such acts.

Signature: Wenhao Zhou

Date: 12/22 2020

## **1. Abstract**

## **2. Introduction**

### 2.1 Background

### 2.2 Project Requirement

### 2.3 Literature Review and Project Scope

## **3. Methodology**

### 3.1 Proposed Functions and Ideas

### 3.2 Proposed Protocols

## **4. implementation**

### 4.1 Steps of Implementation

### 4.2 Programming Techniques Used and Difficulties Encountered

#### 4.2.1 Main techniques:

#### 4.2.2 Difficulties Encountered

## **5. Testing and Results**

### 5.1 Testing Steps

### 5.2 Testing Results and Defects to Optimize

## **6. Conclusion**

## **References**

# 1. Abstract

---

File synchronization is widely applied in applications with the function of providing file downloading and file uploading. Softwares such as Dropbox, Baidu Netdisk, Ali Cloud, have the rudimentary function of file synchronization with breakpoint resume, partially update along with encryption methods to ensure security. To provide file synchronization between devices, network programming enables the interaction between different computers, which mainly containing file transferring protocols. This report will demonstrate the process of implementing a file synchronization application including several transferring protocols between three hosts using python network programming along with the testing methods and results. Also, further improvement plan with be introduced.

# 2. Introduction

---

## 2.1 Background

File sharing is a fundamental function used in every popular cloud applications, such as Baidu Netdisk, Dropbox, Google Drive. These platforms provide reliable, fast and flexible file transfer between devices and cloud. Users can download files from cloud or upload files anywhere and anytime as long as network is connected.

## 2.2 Project Requirement

To simulate techniques provided in cloud softwares, this project is required to implement a ***Large Efficient Flexible and Trusty*** File Sharing application through python network programming. ***Large*** requires that transferring size is up to one GB. ***Efficient*** requires automatic and fast file synchronization between peers along with partially changed file updating and compression. ***Flexible*** means that optional IP addresses of peers are arranged and break point resume transferring is required. More importantly, ***Trusty*** means that errors are prohibited in transferred files and data transmission security is also considered.

## 2.3 Literature Review and Project Scope

In similar applications, for example Dropbox, file synchronization and sharing are realized by listening changes in specified file directories and uploading the newest files to cloud. Furthermore, this app can access the cloud through any devices with internet connection and download files with reliability [1]. In this way, files can be accessed by multiple devices. These sorts of change listening method and flexible access ability can not only simplify users' daily work but also offers portable file storage method. However, it only provides limited storage capacity, those users with large storage demands cannot be fed by this app.

In this project, p2p transferring modal is applied to synchronize files among these three hosts, UDP socket protocol is used to download files with fast speed and it also plays crucial role in broadcasting file information news to peers. Also, compression is used before each file transmission phases to reduce package size. To make the code structure concise, object oriented programming is used to distinguish server function and client function.

## 3. Methodology

---

### 3.1 Proposed Functions and Ideas

There are 2 classes and 7 methods in `main.py`, the detailed functions of each method is shown in the chart below:

#### 1. Main class:

METHOD	FUNCTION
<code>_argparse()</code>	Parse external parameter passing.
<code>zip_new_file(file_dir, out_partial_name)</code>	Find files added into the directory and partially changed files, compress them into a whole zip file.
<code>get_file_mtime()</code>	Find the modify time of each file and save them into a dictionary.
<code>file_listener1()</code>	Listen file information in peer 1.
<code>file_listener2()</code>	Listen file information in peer 2.
<code>file_downloader()</code>	If there are news received in file listener, retrieve new files in peer 1 and peer 2.
<code>send_file_info()</code>	Broadcast file information in share folder including file number, file modified time dictionary, the number of file add time to peer #1 and peer #2.

Table 1: Functions of class Main

#### 2. Server Class:

FUNCTIONS	DESCRIPTIONS
<code>init( self, file_dir, block_size, server_Uport)</code>	Initialize a server object
<code>get_file_size(self, filename)</code>	Get the size of each required file
<code>get_file_block(self, filename, block_index)</code>	Get each computed file block
<code>make_return_file_information_header(self, filename)</code>	Make the information header of each requested file containing file name, file size and total block number
<code>make_file_block(self, filename, block_index)</code>	Pack each block with file block data and information header
<code>msg_parse(self, msg)</code>	Parse the message client sends to return the file block data and
<code>run(self)</code>	Run this thread by calling <code>run(self)</code> , and start to wait for download requests

Table 2: Functions of class Server

**3. Client Class:**

<b>FUNCTIONS</b>	<b>DESCRIPTIONS</b>
init(self, server_address, server_port, file_dir, filename)	Initialize a client object to download the requested files
make_get_file_information_header(self, filename)	Pack the file names and operation code
make_get_fil_block_header(self, filename, block_index)	Get the file header of each block containing filename and bock index
parse_file_information(self, msg)	parse the information server sends and return file size, block size and total block number
parse_file_block(self, msg)	If get right block, parse the block index, block length, and block data
get_file_size(self, filename)	Get the size of downloaded file
unzipFiles(self, addtimes)	Uncompress downloaded files to download directory
run(self)	Run this thread by calling run(self), and start to retrieve the requested file in the server side

Table 3: Functions of Client class

**4. Global variables:**

<b>VARIABLE NAME</b>	<b>DESCRIPTION</b>
file_names	A global dictionary to record the modify time and related file name
add_times	A global counter to record the number of time files are added every time files are added, add this counter by one
finish	A boolean to record the finish state of a client thread
modified_file_name	A string to record the partially updated file found in the share folder

Table 4: Description of global variables

In this application, five threads are initialized to accomplish the whole functions, which are respectively `newFile_detector`, `file_info_sender`, `file_downloader`, `server` and `client`.

- `newFile_detector` calls `zip_new_file()` which is a infinite loop to detect new files added into share folder and compress them into a whole zip file to `zip` folder, and it can also detect newly updated file, compress it to `zip` folder and broadcast the updated file names to peers.
- `file_info_sender` calls for `send_file_info()` which is a infinite loop to send the file information including `add_times`, `file_names` and `modified_file_name` in local share folder to peers.
- `file_downloader` calls `file_downloader()` which is also a infinite loop to listen to the online of the two peers. It is in charge of download the newly added files and updated files in peers' `./share` folder.
- `server` is a infinite loop in charge of listening file download requests incoming from the other two peers. And make file block to return the download request as fleeting as possible
- `client` is a thread in charge of downloading each requested file.

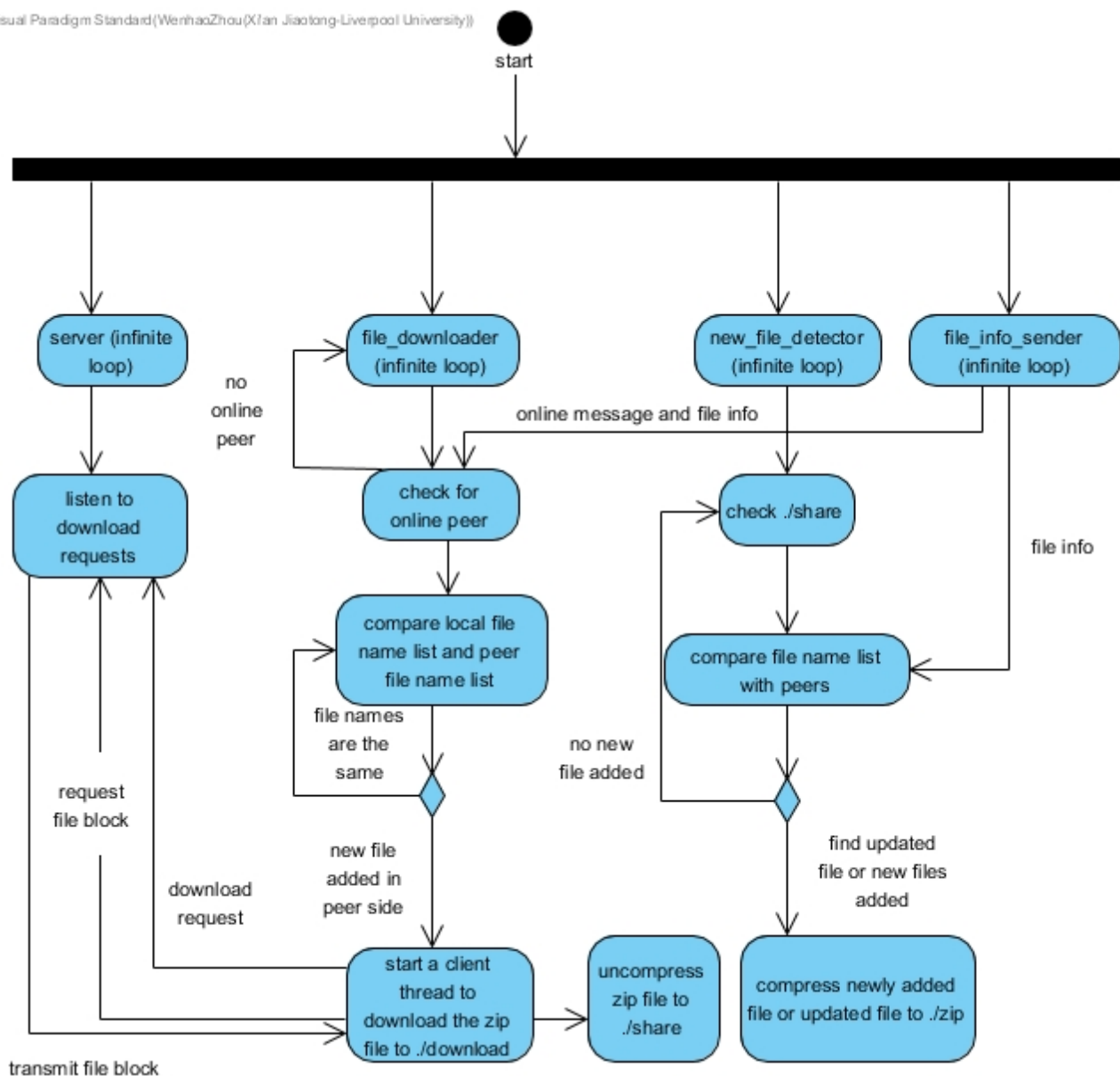
And it is required to cater for different download demands with file dividing download.

## 3.2 Proposed Protocols

In this application, five threads interact with each other through a communicative protocol. As the finite state machine describes below, the above four threads start simultaneously, with file information interchangeable with each other. Meanwhile, `client` thread is used to retrieve the files when `file_downloader` detect new file added in other peer or updated file is found.

Once a peer is online, `file_info_sender` on this peer will set a connection with other peers, and start to send file number, `add_times`, and `file_names` to peer side. If files are added into `./share` folder of local machine, `new_file_detector` will check them out by comparing the file information with other peers and compress them into `./zip` folder. Also, it increase `add_times` by one, to inform the peers that there are newly added zip file in `./zip` folder. Then, `file_info_sender` will inform other peers the news, so that other peers get the updated file list and the `file_downloader` on them will create a `client` thread to retrieve the newest zip file in `./zip` folder on the local machine. Finally `client` thread will uncompress the zip files into local `./share` folder.

UDP socket protocol is used in both file transferring and file information sharing among peers due to its relatively fast transmission speed and connectionless technique. Two ports are used in this application, which are respectively in charge of sending file information and transferring file.



## 4. implementation

### 4.1 Steps of Implementation

1. To implement the new file detector with continuous inspection, one thread with infinite loop is applied.

- As the FSM (Finite State Machine) demonstrates below, firstly, `zip_new_file` with check the local file number with peer file number. If local file number is more than peer file number which indicate new files are added into `./share` folder, then, newly added files will be compressed into a whole zip file, and `add_times` is increased by one.

- If file number is equal to peer side and no `modified_file_name` found, it will enter the next phase. To filter updated file, it checks continuous file modify time with a time interval of 1 second. If the two modify time of one file is not the same which indicates changes have occurred in this file, it will update `modified_file_name`, send it to peers and compress the updated file, finally plus `add_times` by one to wait for other peers retrieve the file.

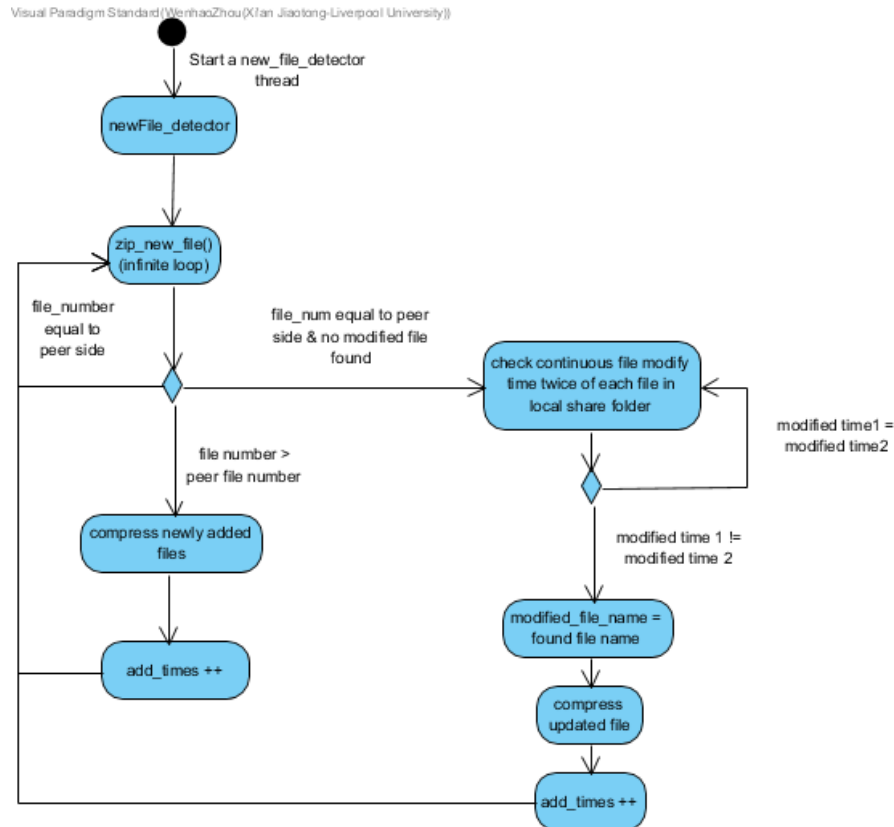


Image 1: FSM of new\_file\_detector

2. To complete the file information sharing part, a infinite loop is rendered to share the information of local machine.

- Once the application is started, it will pack the file number, `file_names` dictionary, `add_times` and send them to peers if they are online.
- Also, to update `modified_file_name` if other peer find updated file, it can receive the `modified_file_name` sent by other peers.

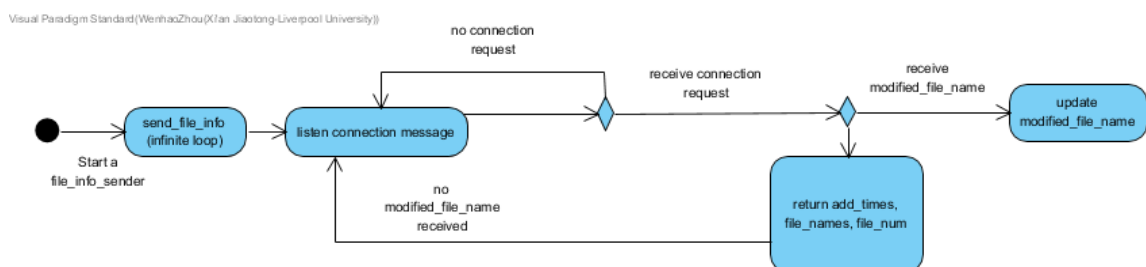


Image 2: FSM of file\_info\_sender

3. To download the files in other peers, three situations are considered:



- If peer is online, `filenames` is not zero and there are new files added in their share folder, then start a `client` thread to retrieve them.
- If the number of files `filenames` and that in peer's folder along with a larger `add_times` value which indicates that there finds partially updated file, then start a `client` thread to retrieve the updated file.
- If the number of file in `filenames` is zero and `add_times` in peer side is larger than local `add_times`, which indicates this machine has been restarted, then compare the number of files in peer side and that in the local side, if there are more files in peer side, then start a `client` thread to download them.

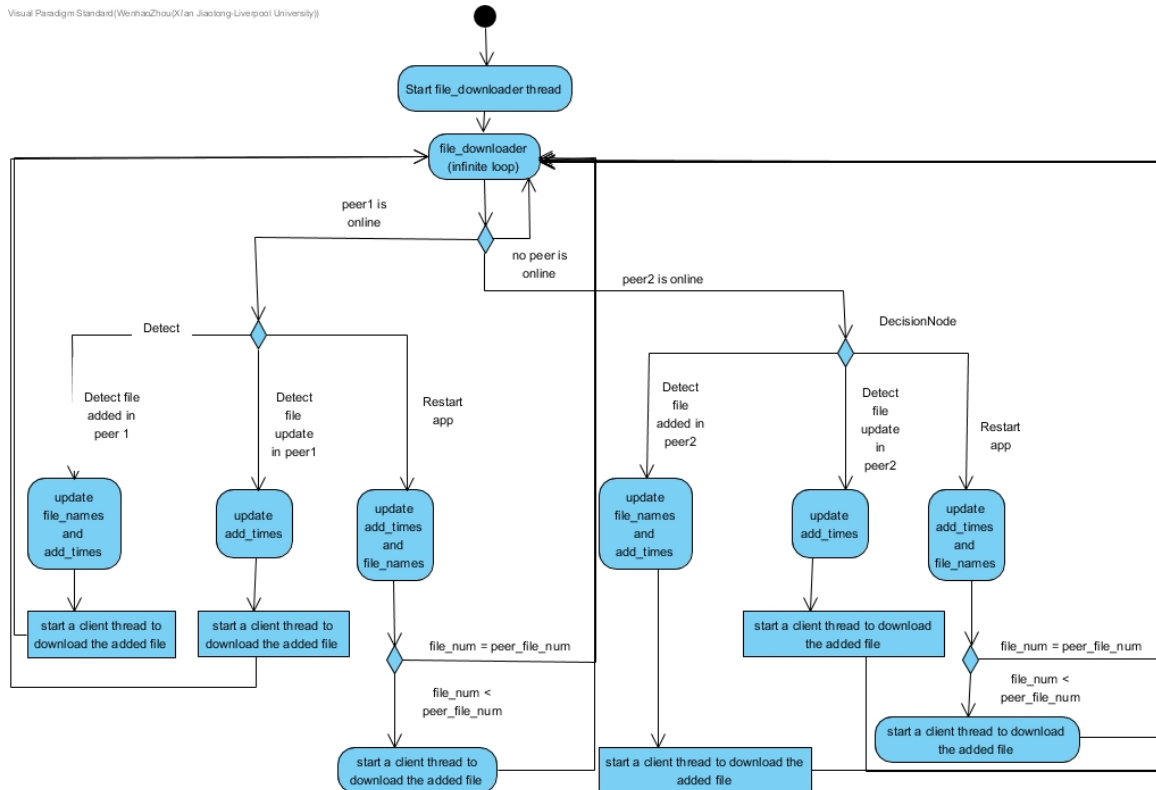


Image 3: FSM of file\_downloader

4. To answer for file download requests from other peers, `server` thread is always there to listen to requesting messages from `client` side.

- It firstly parse the message coming from the requesting peers, then according to the file download request, it divides the whole requested file into blocks.
- It then transmit the file block to the requiring peer.

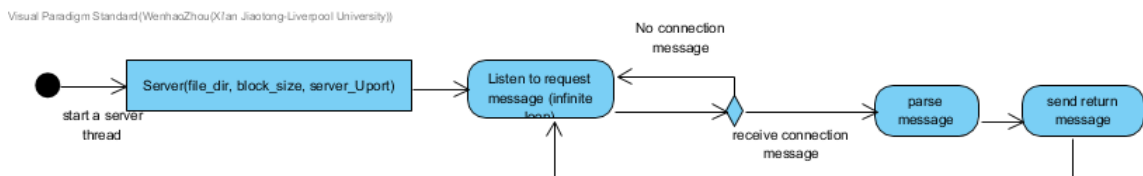


Image 4: FSM of server thread

5. To download the new files from peer side, `client` thread is initialized to retrieve zipped files in peer's `./zip` into its own `./download` folder, and then unzip the zip package to its `./share` folder.

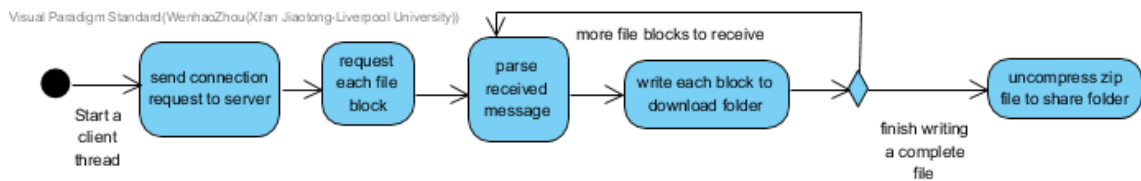


Image 5: FSM of client thread

## 4.2 Programming Techniques Used and Difficulties Encountered

### 4.2.1 Main techniques:

#### 1. **OOP** (Object Oriented Programming)

In the application, `server` and `client` are two separated classes with constructors and data encapsulations. `server` class requires `file_dir`, `block_size`, `server_port` parameters to initialize a `server` instance. Similarly, to obtain different files, `client` class requires `server_address`, `server_port`, `file_dir`, `filename` parameters to initialize a `client` object. Take the advantage of adaptation of different download requests, `client` thread can be utilized to download disparate files according to the demand.

#### 2. **Multiple Threading**

As it stated above, each function of this application is a separated thread, every thread has its own mission to accomplish. To make these threads communicative, global variables are used to hold the property of the file information, and these global variables are shared among threads, which means that if one thread changes a global variable, other threads will get the updated one. To be more concise, multiple threading make the application has multiple functions and burgeoning the utilization of CPU.

### 4.2.2 Difficulties Encountered

#### 1. **Duplicated File Transmission**

When designing the `new_file_detector`, the partially updated file can be indistinguishable with the extracting file, which is triggered by successive file modify time when uncompressing a zip file. As a consequence, the newly-added files can be misapprehended as a partially-updated file, and can be duplicatedly

retransmitted between peers.

To solve this dilemma, a process lock is introduced. The global boolean value `finish` is a recorder to record the finish state of a `client` thread. Once a `client` thread is in progress, `finish` becomes *False*, when a unzip process is finished, the value of which becomes *True*. Therefore, the extracting files are not checked when comparing the successive modify time of a file.

## 5. Testing and Results

---

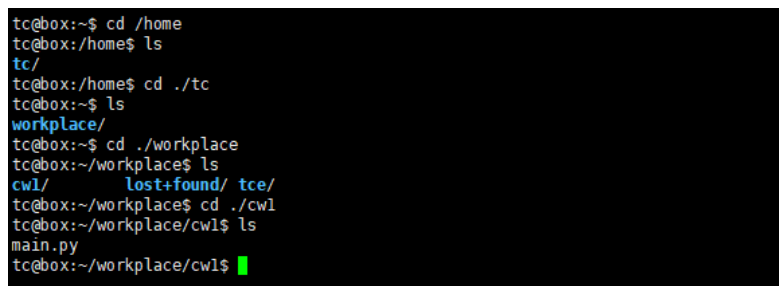
### 5.1 Testing Steps

To test the system with reliability, it is required to run the application on a Linux virtual machine (Tiny Core) with python 3.6. To simulate the running environment, 3 VMs are applied which are respectively AS1peer1, AS1peer2 and AS1peer3.

To prepare for the running environment, the following shell command is used on each VM:

```
cd /home/tc
mkdir ./workplace
sudo mount /mnt/sda1 ~/workplace
sudo chown tc /home/tc/workplace
mkdir -p /home/tc/workplace/cw1
```

To remotely run the code, Xftp is applied to transfer `main.py` to `./workplace`. And Xftp is applied to remotely control the VMs. After the preparing of running environment, the working directory shown in Xshell is the image shown below:



```
tc@box:~$ cd /home
tc@box:/home$ ls
tc/
tc@box:/home$ cd ./tc
tc@box:~$ ls
workplace/
tc@box:~$ cd ./workplace
tc@box:~/workplace$ ls
cw1/      lost+found/  tce/
tc@box:~/workplace$ cd ./cw1
tc@box:~/workplace/cw1$ ls
main.py
tc@box:~/workplace/cw1$
```

Image : Working directory shown in Xshell

To make use of the disk storage, `./sda` is mounted to `./workplace` by the following command:

```
sudo mount /mnt/sda1 ~/workplace
sudo chown tc /home/tc/workplace
```

After preparing the working environment, code is tested step by step.

1. Run the code separately on three machine by:

```
python3 main.py --ip <ip1>,<ip2>
```

2. Use soft link command to link a test file with around 10 MB with `./share` folder on VM1. After the operation, the file is compressed into `./zip` folder.

```
tc@box:~/workplace/cwl$ python3 main.py --ip 192.168.26.129,192.168.26.130
parsing args...
Peer 1 address: 192.168.26.129
Peer 2 address: 192.168.26.130
Service start!
start sending file information...
bg^H^H^Z[1]+  Stopped                  python3 main.py --ip 192.168.26.129,192
.168.26.130
tc@box:~/workplace/cwl$
tc@box:~/workplace/cwl$ bg
[1] python3 main.py --ip 192.168.26.129,192.168.26.130
tc@box:~/workplace/cwl$ ln -s /home/tc/workplace/testImage.jpg /home/tc/workplac
e/cwl/share
tc@box:~/workplace/cwl$ Detect newly added files...
start compressing newly added files
testImage.jpg
Finish compression, compression time: 0.6613633632659912
```

Image : Link a file to `./share` folder on VM1

3. Start VM2, it will get this test file.

```
main.py
tc@box:~/workplace/cwl$ python3 main.py --ip 192.168.26.128,192.168.26.130
parsing args...
Peer 1 address: 192.168.26.128
Peer 2 address: 192.168.26.130
Service start!
start sending file information...
Restart app and reconnected to peer 1...
Files in peer2 are: dict_keys(['./share/testImage.jpg']) addtimes is: 1
Start retrieving the latest added files in peer1 since last offline...
Download start!
start to download the 1 times added file...
Filename: zip1.zip
File size: 10973915
Block size: 20480
Total block: 536
100%|#####| 536/536 [00:01<00:00, 506.29it/s]
Downloaded file is completed.
Transfer time: 1.1141135692596436
Start extracting files...
Extracting time is: 0.20692682266235352
Client end.
```

Image :Start VM2 and retrieve the file

4. Start VM3, testfile is also sent to peer3.

```

tc@box:~/workplace/cwl$ python3 main.py --ip 192.168.26.128,192.168.26.129
parsing args...
Peer 1 address: 192.168.26.128
Peer 2 address: 192.168.26.129
Service start!
start sending file information...
Restart app and reconnected to peer 1...
Files in peer2 are: dict_keys(['./share/testImage.jpg']) addtimes is: 1
Start retrieving the latest added files in peer1 since last offline...
Download start!
start to download the 1 times added file...
Filename: zip1.zip
File size: 10973915
Block size: 20480
Total block: 536
100%|#####| 536/536 [00:00<00:00, 931.52it/s]
Downloaded file is completed.
Transfer time: 0.6369059085845947
Start extracting files...
Extracting time is: 0.33719801902770996
Client end.

```

Image : Start VM3 and get the file

5. Add one Videos and folder with 50 small files of size 650 MB to the `./share` folder of VM2. And VM3 will get the test video. After 2 seconds, kill the process in VM1.

```

Detect files added in peer 2...
Download start!
start to download the 2 times added file...
Filename: zip2.zip
File size: 643914325
Block size: 20480
Total block: 31442
100%|#####| 31442/31442 [00:20<00:00, 1552.84it/s]
Downloaded file is completed.
Transfer time: 20.25361728668213
Start extracting files...
Extracting time is: 6.172633171081543
Client end.

```

Image : VM3 get the folder and

6. Restart VM1, and it will get the files since last offline.

```

Restart app and reconnected to peer 1...
Files in peer2 are: dict_keys(['./share/testImage.jpg', './share/testVideo.mp4']) addtimes is: 2
Start retrieving the latest added files in peer1 since last offline...
Download start!
start to download the 2 times added file...
Filename: zip2.zip
File size: 643914325
Block size: 20480
Total block: 31442
100%|#####| 31442/31442 [00:34<00:00, 904.39it/s]
Downloaded file is completed.
Transfer time: 34.86264395713806
Start extracting files...

```

Image : Restart VM1 and get files

7. Add a test `.txt` file in `./share` folder of VM3, and it will be synchronized to VM1 and VM2. After the Partially update, peer1 and peer2 will get the newest version of the file and replace it with the old one.

```

receive modified file name ./share/testTXT.txt
Detect file update in peer 2...
Download start!
start to download the 2 times added file...
Filename: zip2.zip
File size: 233
Block size: 20480
Total block: 1
100%|#####| 1/1 [00:00<00:00, 1398.57it/s]
Downloaded file is completed.
Transfer time: 0.008184671401977539
Start extracting files...
Extracting time is: 0.0017805099487304688
Client end.

```

Image :Peers get the partially modified file

## 5.2 Testing Results and Defects to Optimize

As the testing results shows above, for the efficiency of the application, it takes 20 seconds to transmit the 650 MB files and 12 seconds to extract and compress the files. For the flexibility, no matter which peer the file is added, it can synchronize them into other peers, also, it can detect partial update among the added files and broadcast it to peers to download the newest version.

However, when referring to defects of the application, it is indicated that when the file is not easy to compress, the speed of file transferring will be constraint, conversely, when the files are easy to compress, it is fast to synchronize the files. Thus, the efficiency of this application is decided by the format of the added files. Further more, network stability is another limitation of transmission speed. With faster network bandwidth, the transferring time will be shorter, in contrast, although UDP protocol leaves out the time of setting up connection with three hand shaking with hosts compared to TCP, precarious network connection will result in slow transferring speed even package loss.

## 6. Conclusion

---

To study deep into the working pattern of file sharing system such as Dropbox, Baidu Netdisk and Google Drive, a python application with network programming is introduced to simulate functions including file synchronization, partially update, and breakpoint resume. The test results is also demonstrated in the above statements, where both advantages and disadvantages are discovered.

To carry on further study, it is required to improve the transferring performance of different formats of file without the constraint of compressing. Also, the stability of transferring needs further improvement. An RDT (Reliable Data Transferring) method including ACK and NCK mechanism can be rendered in to the server and client. If package loss happens, client send a NCK to server, so that server can retransmit the losing package.

## References

---

[1]. What is dropbox [Online]. Available: <http://www.dropboxchina.com/what-is-dropbox.html>