# Coursework 1

# Large Efficient Flexible and Trusty (LEFT) Files Sharing

# Report

**Name:** Jingyang Min

**Student ID:** 1822795

# 1. Abstract

In recent three decades, network has emerged as one of the most important tools in the world. It could assist remote communication among people every day. In the network, files sharing is a significant method to communicate with different people, and a number of platforms have implemented relevant software to support files sharing. It is noticeable that files sharing is powerful in reality. In this coursework, we were required to use Python Socket Programming to design and implement a program which could automatically synchronize files among multiple users. The program which has been designed and implemented in this report has achieved all requirements. Meanwhile, the protocols and architecture designed in the program was analysed in detail. Despite of some defects could be detected by inspections; the entire program is still trusty and these defects are fixable in the future.

# 2. Introduction

## 2.1. Background

Currently, internet is one of the most important technologies in the world. It is capable of sharing information among different connected nodes in each region. For instance, file sharing is one commonly used strategy to share information, it has been applied in different domain to achieve corresponding functions. However, a number of less professional file sharing software can also serve us in daily life, such as Dropbox, Google Drive, and XJTLU BOX in our campus [1].

It is noticeable that there are numbers of software which could be adopted as file sharing tools, but the most of them have similar architectures including C/S (Client-Server) or Peer-to-Peer (P2P). In specific, C/S is commonly used to access servers in large companies' content distribution network (CDN), they usually have large number of expensive data centre to maintain all accessing pipes [2]. However, due to the high expense of maintaining an efficient and reliable transfer of C/S architecture, P2P is more efficient and lower cost compared with C/S when file sharing function required among end-users. In P2P, file could be divided into chunks and distributed into whole user network. For example, BitTorrent could use various of strategies, such as tit-for-tat to weigh the balance of sending and requesting between each peer to achieve high

transferring rate [2].

### 2.2. Project Requirement

In this project, we were required to utilize Python Socket Programming to design and implement files sharing, which has the capability to achieve large file or directory reliable, flexible, and efficient transfer. For the large requirement part, the program should support any format of items including files and folders. Meanwhile, a single item size could be 1GB. For the efficient part, the transferring speed should be fast and new items should be synchronized automatically, and the partially updated item could also partially update at receiver side. To achieve fast transferring, applying compression is one feasible solution. In flexible part, the IP addresses would be passed into program as an argument, and interrupted transfer could be resumed. For reliable or trusty part, the transferred item should be completely consistent after every time of transferring, and data security should also become one optional choice which would be set as an argument of the program.

It is noticeable that there are numbers of freedom for us to design and implement our own program, but specification has some restriction on some points. For the application layer protocol design, we are not allowed to implement HTTP but our own protocol is acceptable. Meanwhile, we can utilize TCP or UDP which are transport layer protocols in this project, mixed both of them is also allowed. The program architecture could be C/S or P2P, mixed mode is allowed. The initialized port can be located in 20000 to 30000, and total port number is not limited.

### 2.3. Implemented Functions in Program

In the Python program, UDP was adopted as the transport layer protocol to support my program. According to the example codes of reliable transferring based on UDP, all required functions stated in the coursework specification were designed and implemented. The program includes fast and reliable items sharing, automatically synchronization, partially updating, resume from interruption, compressing, and encryption. The detailed functions implementation and architecture design would be illustrated and analysed in following sections.

# 3. Methodology

## 3.1. Proposed Protocols

To ensure that UDP which is a not reliable transferring protocol become reliable, the reliable data transfer protocol (rdt) should be implemented manually. According to the description on textbook, the rdt3.0 was implemented in this program. Meanwhile, the efficient communication protocol (ecp) was design and implemented. The whole descriptions can be demonstrated as below.

**Table 1:** The Internet Protocol Stack in my Program

| |
|---|
| Efficient Communication Protocol (ecp) → Application Layer |
| Reliable Data Transfer (rdt) → Application Layer |
| UDP → Transport Layer |
| Network Layer |
| Link Layer |
| Physical Layer |

However, the bit errors detection in my protocol was excluded because it could be guaranteed by link layer protocols, such as Parity Checking, checksum, and Cyclic Redundancy Check (CRC) [3]. Meanwhile, the application layer message header is necessary to support rdt. The specific design of this protocol presented as below. In these mind maps, the sequence number, such as 1. 2.1. 2.2. 3. represents the real header information which is packed into the message header.

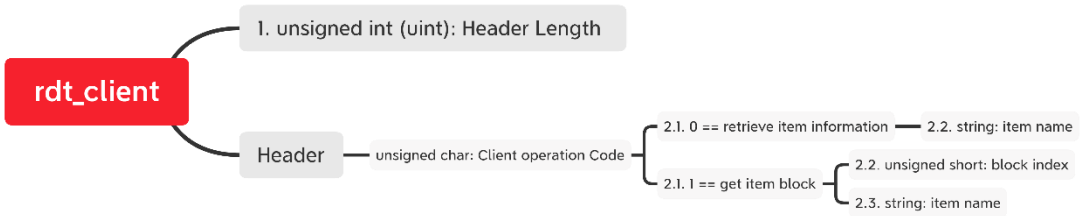


**Figure 1:** The Protocol Structure on rdt Client

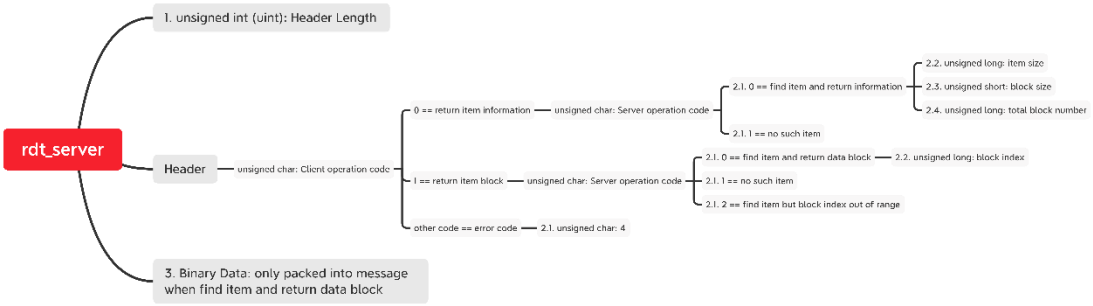


**Figure 2:** The Protocol Structure on rdt Server

Due to the request of item information is always proceed before transferring item blocks, the block size in return item block message header is not necessary to appear. Therefore, when server operation code is 0, the header information only contains server operation code and block index. Meanwhile, it is noticeable that when client attempt to retrieve item information, the return information of block size is unsigned short type. This is because for Socket Programming in Python, the maximum buffer size for UDP is 65535. It is the maximum number of unsigned short type in C language. To remain sufficient space of header, the block size cannot be bigger than this number. Actually, in the program, block size has been set to 65300 to provide high transferring speed and avoid buffer overflow simultaneously.

After implemented rdt for both client and server, the ecp is required to provide ordinary communication among different end-users. The ecp has been designed as a plaintext protocol, which contains a single character header to represent different requests and responses. The specification of this protocol provided as below.

**Table 2:** Efficient Communication Protocol (ecp) Specification

| Header | ‘h’ | ‘u’ | ‘w’ | ‘s’ | ‘e’ |
|---|---|---|---|---|---|
| Represented Information | ‘have’ (Request) | ‘update’ (Request) | ‘working’ (Response) | ‘same’ (Response) | ‘excess’ (Response) |

Explanation:

**Multiple items would be concatenated by a special string ‘|*|’.**

**For example, if this machine will send ‘h’ request to another machine, and it has a.txt and b.mp4. The request would be ‘ha.txt|*|b.mp4’ in bytes type.**

‘have’: The items appeared in this machine.

‘update’: The detected updated items in this machine.

‘working’: This machine is working for other tasks, such as sending, receiving or compressing.

‘same’: The items owned by other sending ‘h’ request machine equals to the items in this machine.

‘excess’: The items in this machine are not appeared in the ‘h’ request received from another machine.

Initially, the ‘l’ header which represents ‘lacking’ response was designed but it is not necessary. Because of other machines can retrieve lacking items by sending rdt_client format message, the ‘lacking’ response is deprecated. Meanwhile, the separation string was set as ‘|*|’ because a simple separation mark could be ‘collision’ with encrypted communication message. The separation string applied in the program was tested repeatedly which is capable of avoiding collision.

### 3.2. Proposed Functions and Ideas

To ensure that program can recover from interruption, the program would read from log and remove the broken item from items list, then run shutdown_recovery method to contact with all provided ip addresses. This method would complete after successfully recovery. Next, the program would initialize itself by attempting to contact with all provided ip addresses. After contact complete, the program would run create_trigger_core_process to initialize a process for items modification detection. Following this, program would proceed into steady state to wait for requests from other machines. Meanwhile, if the trigger core process has been triggered by items modification, it will attempt to send corresponding request to all provided ip addresses, then process their response correspondingly.

## 4. Implementation

### 4.1. Steps of Implementation

Initially, the core function which needs to be implemented is rdt. Due to the unreliable data transfer characteristic of UDP, this function is the fundamental task to support all other features in the program. After rdt implemented, in order to support compression, encryption, and detection of items modification, 'utils' package has been created. All encapsulated tools could help programmer to invoke them efficiently. Meanwhile, these separated tools are also maintainable compared with coupling them into the main structure. For example, __init__.py in 'utils' could enhance the efficiency of configuring environment for users. Following encapsulated tools, ecp was implemented to support top layer communication among different machines. Eventually, the main.py was designed and implemented to support handling user input parameters, and automatic environment construction.

### 4.2. Program Flow Charts

This program was designed as a finite state machine (FSM), and it is appropriate to be demonstrated by state machine diagram. The following figure presented each operating state of the program.

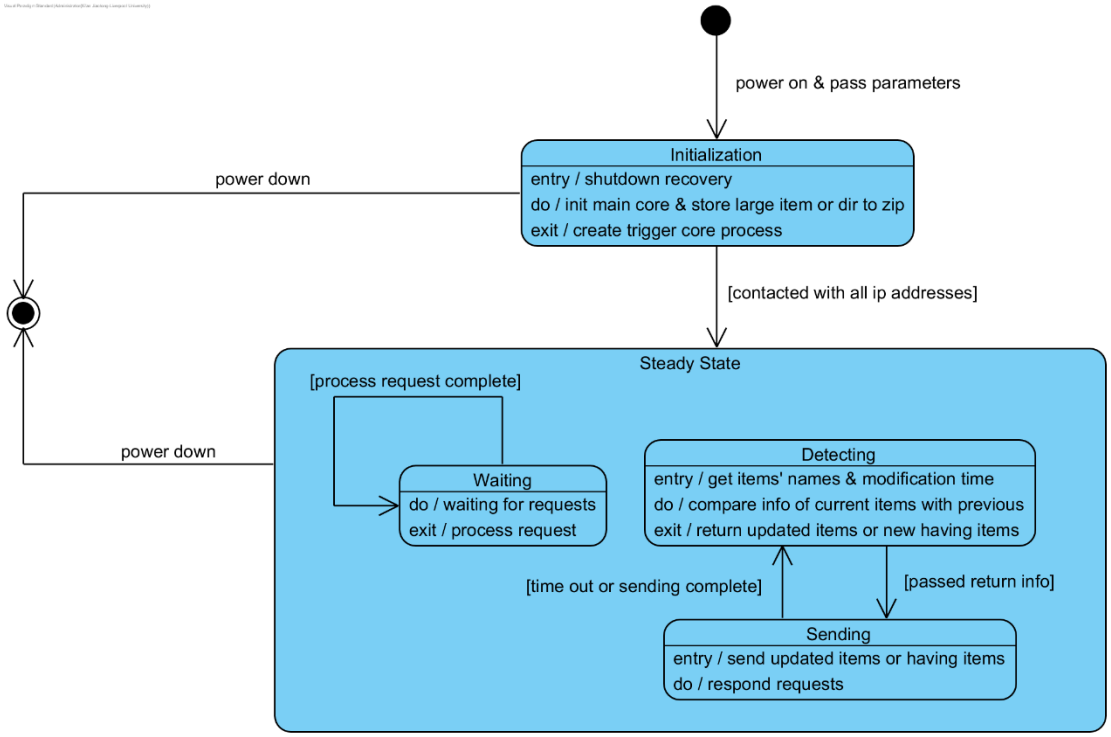


**Figure 3:** State Machine Diagram of the Program

After detailed analysis of the architecture, it is noticeable that concurrent running exists in the program. However, state machine diagram is not suitable to demonstrate concurrent running. Therefore, an activity diagram which also represents the operating flow provided as below.

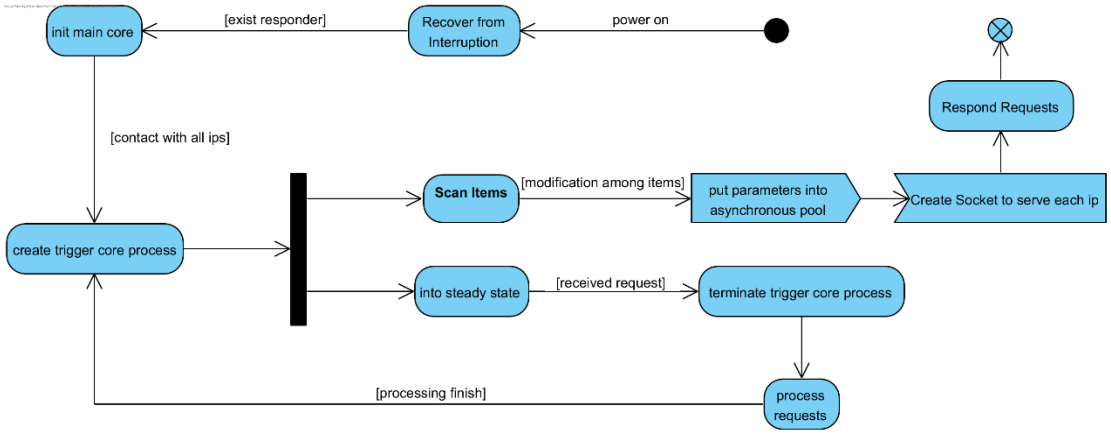


**Figure 4:** Activity Diagram of the Program

### 4.3. Programming Skills and Difficulties Solving

The main programming approach in this program is procedure-oriented programming, but some encapsulated tools, which have been utilized in the whole program are consistent with object-oriented programming (OOP) technique. For instance, the Zipper tool in the program was constructed as a class to support compressing and decompressing functions. It also has an option that user could choose compress contents of item into memory or store to local. In short, this class assisted major components to achieve low coupling principle in the program.

For the parallel programming part, multiprocessing package has been utilized to achieve all parallel operations. Initially, main process would run main core to process contacting with other machines. Next, main core would create trigger core process by using **Process** in multiprocessing package. Afterwards, main core would proceed into steady state. Concurrently, trigger core is scanning items in share folder to detect modification. If any item has been modified or new item added, trigger process would proceed relevant information to an asynchronous **Pool**. At the early stage of programming, **Pool** was not be considered because **Process** is intuitive to be created by loop. However, creating Processes by loop is dangerous because it will exhaust

system resources immediately if loop was not operating expectantly. Therefore, **Pool** has been used in the final version of program, and it is also a normally adopted solution in software industry.

During the programming stage, it is noticeable that scanning process could be interrupted by received items from other machines. In this program, the solution is to terminate trigger process during the receiving procedure. In order to guarantee the process has been terminated, join method should be applied after terminate. This solution is efficient because serving pool would not be terminate until they finish the service.

# 5. Testing and results

## 5.1. Testing Environment

All testing was assembled on Python 3.6.9 Interpreter on Linux platform. In order to unify the running environment of Linux, Dr. Fei Cheng provided us core Linux which could occupy tiny space to run the system. All allowed modules in Python from third-party listed as below.

Table 3: Allowed Third-Party modules on Python

| | | | |
|---|---|---|---|
| numpy | PTable | pycryptodome | tqdm |

Although core Linux has the advantage of less space occupation, most of us using MacOS or Windows as our operating system. It has less opportunity to replace these installed systems in our computers, so the core Linux was installed into virtual machine provided by Oracle, named ‘Oracle VM VirtualBox’. Due to the space is not sufficient to store all testing data in virtual machines (VMs), we need to mount local disk to VMs. Next, transfer the testing data to VMs by using command scp in MacOS or pscp in Windows. The running parameters contains ip and encryption. For the parameter of ips, it could be obtained by ‘ifconfig’ command in Linux terminal. The IPv4 addresses of VMA, VMB, and VMC in the following testing phases are 192.168.56.118, 192.168.56.119, 192.168.56.120 separately. The encryption is optional and it will be passed into program as ‘--encryption yes/no’. After configured relevant parameters in provided script remoterun.py, then run the program on core

Linux. In following testing phases, all testing data added into share folder was operated by soft link in Linux.

## 5.2. Testing Plan and Results

The testing procedure strictly referenced the procedure provided in coursework specification. All testing procedure was operated with remoterun.py script in PyCharm. In general, there are four phases in the testing specification, it will be demonstrated with screenshot in the following contents.

### 5.1. Phase 1

The ‘param’ and ‘remote_ip’ variable presented in the figure below was the settings of VMA. Likewise, VMB and VMC have similar configuration compared with VMA. The settings of them would not be provided repeatedly.

```
# Settings
py_files = ['main.py', 'LEFT_main_core.py', 'LEFT_trigger_core.py', 'utils/scanner.py',
            'utils/zipper.py', 'utils/encipher.py', 'utils/__init__.py']
remote_python_interpreter = '/usr/local/bin/python3'
remote_current_working_directory = '/home/tc/workplace/cw1'
param = '--ip 192.168.56.119,192.168.56.120'
remote_ip = '192.168.56.118'
remote_username = 'tc'
remote_password = '123'
```

**Figure 5:** Configured Parameters for VMA

1. Run VMA without error and add File 1 to the share folder located at current working directory (CWD) of VMA.

```
Send main.py to 192.168.56.118:/home/tc/workplace/cw1
Send LEFT_main_core.py to 192.168.56.118:/home/tc/workplace/cw1
Send LEFT_trigger_core.py to 192.168.56.118:/home/tc/workplace/cw1
Send scanner.py to 192.168.56.118:/home/tc/workplace/cw1/utils
Send zipper.py to 192.168.56.118:/home/tc/workplace/cw1/utils
Send encipher.py to 192.168.56.118:/home/tc/workplace/cw1/utils
Send __init__.py to 192.168.56.118:/home/tc/workplace/cw1/utils
###################### RUN main.py ######################
after break recovery
send have request:h to ip:192.168.56.119
time out
send have request:h to ip:192.168.56.120
time out
into steady state
```

```
Send main.py to 192.168.56.118:/home/tc/workplace/cw1
Send LEFT_main_core.py to 192.168.56.118:/home/tc/workplace/cw1
Send LEFT_trigger_core.py to 192.168.56.118:/home/tc/workplace/cw1
Send scanner.py to 192.168.56.118:/home/tc/workplace/cw1/utils
Send zipper.py to 192.168.56.118:/home/tc/workplace/cw1/utils
Send encipher.py to 192.168.56.118:/home/tc/workplace/cw1/utils
Send __init__.py to 192.168.56.118:/home/tc/workplace/cw1/utils
###################### RUN main.py ######################
after break recovery
send have request:h to ip:192.168.56.119
time out
send have request:h to ip:192.168.56.120
time out
into steady state
return items:['File_1'] because of added new items
send items:hFile_1 to ('192.168.56.119', 22001)
send items:hFile_1 to ('192.168.56.120', 22001)
```

**Figure 6, 7:** Testing of Phase 1 on VMA

2. Run VMB without error and get File 1 and store it to the share folder located at

CWD of VMB.

```
###################### RUN main.py ######################
after break recovery
send have request:h to ip:192.168.56.119
time out
send have request:h to ip:192.168.56.120
time out
into steady state
return items:['File_1'] because of added new items
send items:hFile_1 to ('192.168.56.119', 22001)
send items:hFile_1 to ('192.168.56.120', 22001)
in steady state, received from ('192.168.56.119', 22001)
process common response for ('192.168.56.119', 22001)
respond have request for ('192.168.56.119', 22001)
lacking_items:[]
excess_items:['File_1']
send excess response
set time out to 6s
data:b''
sender start
received break msg
```

```
###################### RUN main.py ######################
after break recovery
send have request:h to ip:192.168.56.118
received msg from ('192.168.56.118', 22001)
msg:b'eFile_1'
char:e
received excess responses
excess_items_list:['File_1']
receiver start
item name:File_1
core address:('192.168.56.118', 22001)
start block index is:0
File_1: 100%|################################| 18.4M/18.4M [00:00<00:00, 19.6MB/s]
Downloaded content is completed.
receive time consumed:1.0001556873321533
send have request:hFile_1 to ip:192.168.56.120
time out
into steady state
```

**Figure 8:** Testing of Phase 1 on VMA **Figure 9:** Testing of Phase 1 on VMB

3. Run VMC without error and get File 1 and store it to the share folder located at CWD of VMC.

```
into steady state
return items:['File_1'] because of added new items
send items:hFile_1 to ('192.168.56.120', 22001)
send items:hFile_1 to ('192.168.56.119', 22001)
in steady state, received from ('192.168.56.119', 22001)
process common response for ('192.168.56.119', 22001)
respond have request for ('192.168.56.119', 22001)
lacking_items:[]
excess_items:['File_1']
send excess response
set time out to 6s
data:b''
sender start
received break msg
in steady state, received from ('192.168.56.120', 22001)
process common response for ('192.168.56.120', 22001)
respond have request for ('192.168.56.120', 22001)
lacking_items:[]
excess_items:['File_1']
send excess response
set time out to 6s
data:b''
sender start
received break msg
```

```
send have request:hFile_1 to ip:192.168.56.120
time out
into steady state
in steady state, received from ('192.168.56.120', 22001)
process common response for ('192.168.56.120', 22001)
respond have request for ('192.168.56.120', 22001)
lacking_items:[]
excess_items:[]
send same response
in steady state, received from ('192.168.56.118', 60421)
process common response for ('192.168.56.118', 60421)
respond have request for ('192.168.56.118', 60421)
lacking_items:[]
excess_items:[]
send same response
```

```
###################### RUN main.py ######################
after break recovery
send have request:h to ip:192.168.56.118
received msg from ('192.168.56.118', 22001)
msg:b'eFile_1'
char:e
received excess responses
excess_items_list:['File_1']
receiver start
item name:File_1
core address:('192.168.56.118', 22001)
start block index is:0
File_1: 100%|################################| 18.4M/18.4M [00:00<00:00, 35.7MB/s]
Downloaded content is completed.
receive time consumed:0.5552070140838623
send have request:hFile_1 to ip:192.168.56.119
received msg from ('192.168.56.119', 22001)
msg:b's'
char:s
into steady state
in steady state, received from ('192.168.56.118', 40764)
process common response for ('192.168.56.118', 40764)
respond have request for ('192.168.56.118', 40764)
lacking_items:[]
excess_items:[]
send same response
```

**Figure 10, 11, 12:** Testing of Phase 1 on VMA, VMB, VMC

## 5.2. Phase 2

1. Add File 2 and folder to VMB, after 2 seconds the app on VMA would be terminated and VMC would obtain all these items.

```
###################### RUN main.py ######################
after break recovery
send have requestthFile_1 to ip:192.168.56.119
received msg from ('192.168.56.119', 22001)
msg:b'eFile_2|*|folder'
char:e
received excess responses
excess_items_list:['File_2', 'folder']
receiver start
item name:File_2
core address:('192.168.56.119', 22001)
start block index is:0
File_2: 100%|################################| 1.25M/1.25M [00:00<00:00, 30.9MB/s]
item:File_2 is a zip file
Downloaded content is completed.
item name:folder
core address:('192.168.56.119', 22001)
start block index is:0
folder: 100%|################################| 8.53M/8.53M [00:00<00:00, 11.8MB/s]
item:folder is a zip file
Downloaded content is completed.
receive time consumed:31.607824563980103
send have requestthFile_1|*|File_2|*|folder to ip:192.168.56.120
received msg from ('192.168.56.120', 22001)
msg:b's'
char:s
into steady state
```

```
in steady state, received from ('192.168.56.120', 22001)
process common response for ('192.168.56.120', 22001)
respond have request for ('192.168.56.120', 22001)
lacking_items:[]
excess_items:[]
send same response
return items:['File_1', 'File_2'] because of added new items
zip File_2 to local
return items:['File_1', 'File_2', 'folder'] because of added new items
complete zip File_2 to local
send items:hFile_1|*|File_2 to ('192.168.56.118', 22001)
send items:hFile_1|*|File_2 to ('192.168.56.120', 22001)
zip folder to local
complete zip folder to local
send items:hFile_1|*|File_2|*|folder to ('192.168.56.118', 22001)
send items:hFile_1|*|File_2|*|folder to ('192.168.56.120', 22001)
in steady state, received from ('192.168.56.118', 22001)
process common response for ('192.168.56.118', 22001)
respond have request for ('192.168.56.118', 22001)
lacking_items:[]
excess_items:['File_2', 'folder']
send excess response
set time out to 6s
data:b''
sender start
in steady state, received from ('192.168.56.118', 22001)
process common response for ('192.168.56.118', 22001)
set time out to 6s
data:b'\x00\x00\x00\x07\x00folder'
sender start
received break msg
```

```
in steady state, received from ('192.168.56.119', 49539)
process common response for ('192.168.56.119', 49539)
respond have request for ('192.168.56.119', 49539)
lacking_items:['File_2']
receiver start
item name:File_2
core address:('192.168.56.119', 49539)
start block index is:0
File_2: 100%|################################| 1.25M/1.25M [00:00<00:00, 32.8MB/s]
item:File_2 is a zip file
Downloaded content is completed.
receive time consumed:30.49102520942688
excess_items:[]
in steady state, received from ('192.168.56.119', 49539)
process common response for ('192.168.56.119', 49539)
respond have request for ('192.168.56.119', 49539)
lacking_items:[]
excess_items:[]
send same response
in steady state, received from ('192.168.56.119', 51910)
process common response for ('192.168.56.119', 51910)
respond have request for ('192.168.56.119', 51910)
lacking_items:['folder']
receiver start
item name:folder
core address:('192.168.56.119', 51910)
start block index is:0
folder: 100%|################################| 8.53M/8.53M [00:00<00:00, 18.4MB/s]
item:folder is a zip file
Downloaded content is completed.
receive time consumed:0.7694029808044434
excess_items:[]
in steady state, received from ('192.168.56.118', 22001)
process common response for ('192.168.56.118', 22001)
respond have request for ('192.168.56.118', 22001)
lacking_items:[]
excess_items:[]
send same response
```

**Figure 13, 14, 15:** Testing of Phase 2 on VMA, VMB, VMC

### 5.3. Phase 3

1. Add File 3 to VMC, after VMA and VMB obtain the entire file, partially updated part on VMC would be automatically send to VMA and VMB.

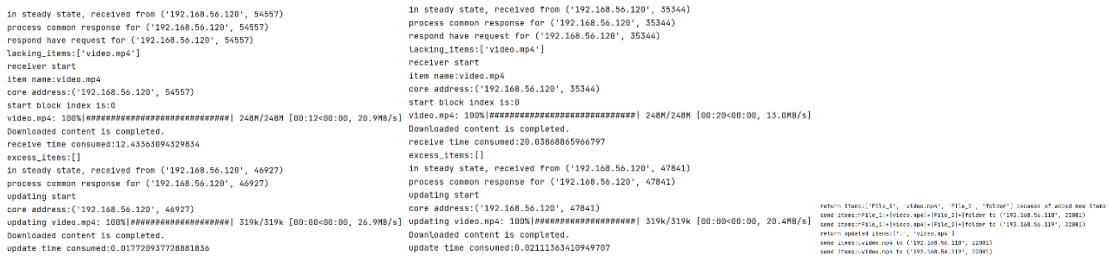


**Figure 16, 17, 18:** Testing of Phase 3 on VMA, VMB, VMC

### 5.4. Encryption Phase

The parameters in VMA set as below. VMB and VMC have similar settings so it will not be presented below repeatedly.

```
# Settings
py_files = ['main.py', 'LEFT_main_core.py', 'LEFT_trigger_core.py', 'utils/scanner.py',
            'utils/zipper.py', 'utils/encipher.py', 'utils/__init__.py']
remote_python_interpreter = '/usr/local/bin/python3'
remote_current_working_directory = '/home/tc/workplace/cw1'
param = '--ip 192.168.56.119,192.168.56.120 --encryption yes'
remote_ip = '192.168.56.118'
remote_username = 'tc'
remote_password = '123'
```

**Figure 19:** Encryption Testing Parameters on VMA

To test robustness, the starting order and adding testing data would not be same as previous phases.

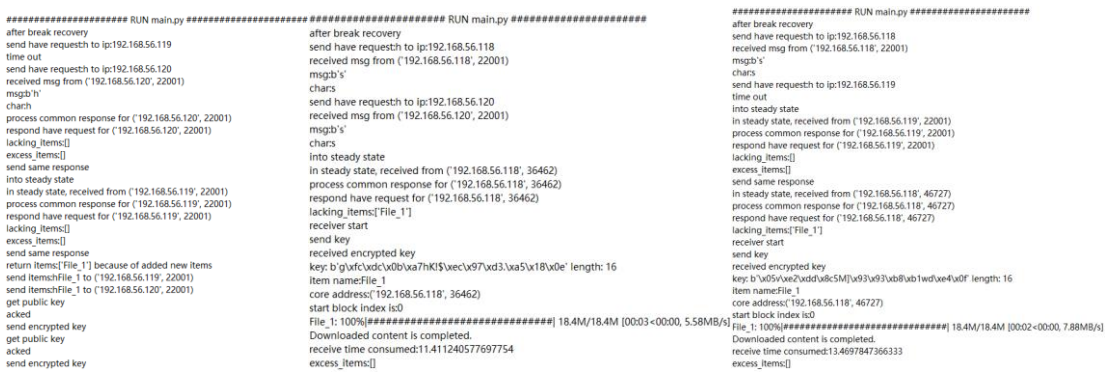


Figure 20, 21, 22: Encryption Testing on VMA, VMB, VMC

Therefore, all testing phases complete and all testing achieved appropriate results.

## 6. Conclusion

In summary, it is noticeable that files sharing is one of the most significant information sharing approaches in the world, and numbers of maturely developed software could be utilized by us. In this coursework, we were required to use Python Socket Programming to achieve files sharing among multiple machines. In this program, architecture design and code implementation are crucial because both of them could affect the performance entirely. Although the goals of all phases are achieved, there are still some potential bugs in the program. For example, after terminate the trigger core process. It is possible to cause the pool raising BrokenPipeError because the controller process, which is trigger core process has been terminated to avoid sensing of received items. Meanwhile, the scanner in trigger core has a time gap presented as below.

```
def create_trigger_core(ip_addresses: List[str], port: int, compressing, encryption: bool):
    pool_size = len(ip_addresses)
    remain_cpu_core = cpu_count() - 2
    max_pool_size = remain_cpu_core if remain_cpu_core > 0 else 1
    lock = Lock()
    pool = Pool(pool_size if pool_size <= max_pool_size else max_pool_size, initializer=init_poo
    while True:
        items_list = get_new_or_update_items()<-scanner
        if items_list[0] == '.':
            items = str('u' + '|*|'.join(items_list[1:]))
        else:
            items = str('h' + '|*|'.join(items_list))
        parameters = [((ip, port), items, compressing, encryption) for ip in ip_addresses]
        pool.starmap_async(init_trigger_core, parameters)
```

Time Gap

**Figure 23:** Time Gap Defect in the Program

When item list has been modified during this time gap, program cannot sense of it. It is an architecture problem, so it needs substantial amount of time to address it. However, the entire program has been designed with high extensibility and maintainability. Thus, this issue is solvable compared with re-construct whole program.

## 7. Reference

[1] F. Cheng. *Coursework 1 Large Efficient Flexible and Trusty (LEFT) Files Sharing*. (2020). Accessed: Nov. 16, 2020. [Online]. Available: https://learningmall.xjtlu.edu.cn/pluginfile.php/52734/mod_assign/introattachment/0/CW1.pdf?forcedownload=1

[2] J. F. Kurose and K. W. Ross., "Application Layer," *Computer Networking: A Top-Down Approach*, M. Goldstein, Ed., 7th edition. Pearson, 2017, ch. 2.

[3] J. F. Kurose and K. W. Ross., "The Link Layer and LANs: Error-Detection and Correction Techniques," *Computer Networking: A Top-Down Approach*, M. Goldstein, Ed., 7th edition. Pearson, 2017, ch. 6, sec. 2, pp. 444-451.