CAN201 Coursework1 Report

Student ID: 1822766

Name: Hangming Ye

Abstract

The report intends to convey the development progress of CAN201 Coursework 1: Large Efficient Flexible and Trusty (LEFT) Files Sharing, mainly focus on analysis of the requirement, information about methodologies, details of implementation the requirements and testing progress. TCP is chosen as the transport layer protocol due to its reliability and exceed remarkable results.

1. Introduction

Project requirement:

The project could be generalized as a file synchronization program between different hosts. Specification of the requirement could be divided into functional requirement and non-functional requirement as below:

Functional requirements

1. File detection: the program shall detect new files and modified files in the 'share' folder and return its information.
2. File send: the program shall send file using certain IP address and port to host.
3. File receive: the program shall receive files from other host on a certain port.
4. Encryption: the program shall provide an option to check whether to encrypt the file during the transmission.
5. Partial update: the program could partially update the modified file by only send the modified part of the file to reduce the network load.

Nonfunctional requirement:

1. Accident resume: when the host is accidently killed, the host could resume the progress not finished yet when it restarted.
2. Memory size optimization: The program shall not use too much memory to meet most hardware conditions of the host machine.
3. Time-sensitive: The program shall transfer the file as fast as possible.

Background and Literature review

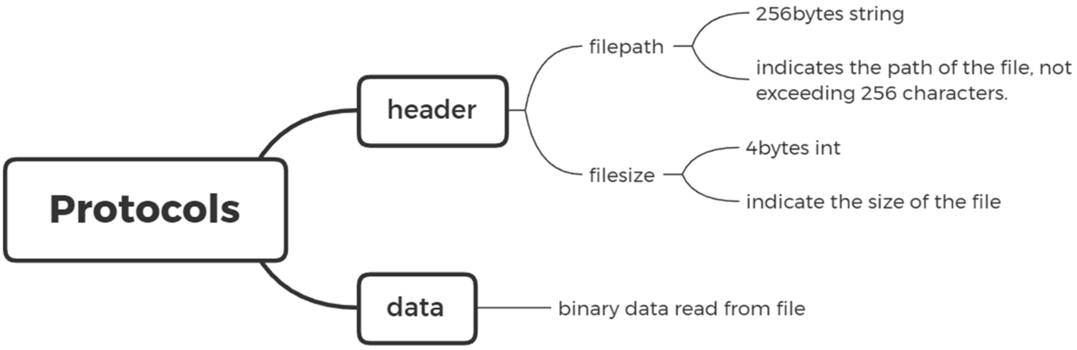
The existing popular file sharing programs are well developed, such as iCloud, Dropbox, Onedrive and Baidu Netdisk. The main difference of them and the project is whether need a remote server [1], which cost a lot and need further maintenance. They are more like a disk to storage data rather than synchronization. Other synchronization program based on BitTorrent protocols using Peers-to-Peers structure like Resilio Sync [2] spending too much time to find peers and the transfer speed is slow when it in local network environment comparing to FTP service.

Achievement

The program meet requirements raised in 'project requirement' part. The program provides a file synchronization method based on TCP protocol, which could detect new files and modified files, transfer them and receive missing file from other hosts. The progress is automatic and the transmission is fast, utilize the advantage of multiple cores and the memory cost is low.

2. Methodology

Proposed protocols


Protocols
header
filepath
256bytes string
indicates the path of the file, not exceeding 256 characters.
filesize
4bytes int
indicate the size of the file
data
binary data read from file

Proposed functions

Function name	Description
_argparse()	Get the IP addresses and compression arguments.
listener(port, log, encryption)	Wait for sendManager to connect and pass connection to receiveManager
receiveManager(port, log, encryption)	Bind a port and wait for other hosts to connect, send them log file of the share folder to host at first time, then wait to receive file until the program close, when receive send file request, open a thread and pass the arguments.
	Connect to other host using 'ip' and 'port', wait for log file of others, find the missing file and pass

sendManager(ip, port, encryption, log)	them to threads, one file a thread. Then, detect the 'share' folder and when modified file and new file find, pass them to new thread.
receiver(port, log, encryption)	Create a socket using 'port', wait for a host to connect, receive its header, and repeatedly receive file data then write into file. After receive is done, add log item and write into log file.
sender(ip, port, file, encryption)	Read the file from 'file[path]', get it size, if the file is large, use compression to reduce it size, pack path, size, compression, encryption into packet and send them to host using ip, port, then send the file data by multiple times.
partialSend(ip, port, file):	This function is similar to sender() function but it only send 0.1% at the front of the file.
partialUpdate(port, log)	This function is similar to receiver() function, but when it received all data, it will replace the 0.1% data at the front of the file.
traverse(dir_path)	Scan the dir_path and return all file's path
createLog(dir_path)	Create log info for each item in dir_path
writeLog(path, log)	Write 'log' into 'path'
readLog(path)	Read log from 'path' and return log
logInit(dir_path, log_path)	When there exist a log, read and return it, otherwise scan folder, create a log, write and return it.
detectChanges(path)	Scan folder every 0.1 scend, get the file info and compared with log. When new files find, change log, return 0 and this file's info. When modified file find, change log, return 1 and this file's info.
compareDiff(log_s, log_r)	Compare the log_s and log_r return a file list containing file log_s has and log_r missing or incomplete.
heart(port)	A separate listener running forever, to indicate that the host is alive
reconnect(ip, port)	A separate connecter running forever, check the "IP" host is whether alive, if it's disconnected, restart the send part again

Ideas

Firstly, after program is started, "ip addresses" and "whether to encrypt" is parsed by _argparse(), then create "share" folder, initialize log using logInit(dir_path, log_path), set log as the shared dict between different processes. After that, create receiveManager() process and for each ip address, create a sendManager() process. The receiveManger() process will wait host to connect, when connected, it will send log information to sendManager() at another host and keep waiting. sendManager() will compare the difference of received log and its own log, return the file list contains filemissing and incomplete files. Then sendManager() will send message to receiveMager() to notify it to receive file and pass the file to sender(), one file each thread. The receiveManager() will create a thread to receive the file using receiver(). Then sendManger() will detect file changes, when a new file added or a existing file changed, it will send message to receiveManager() to notify it to receive. The receiveManager() will create a thread to receive the file using receiver().

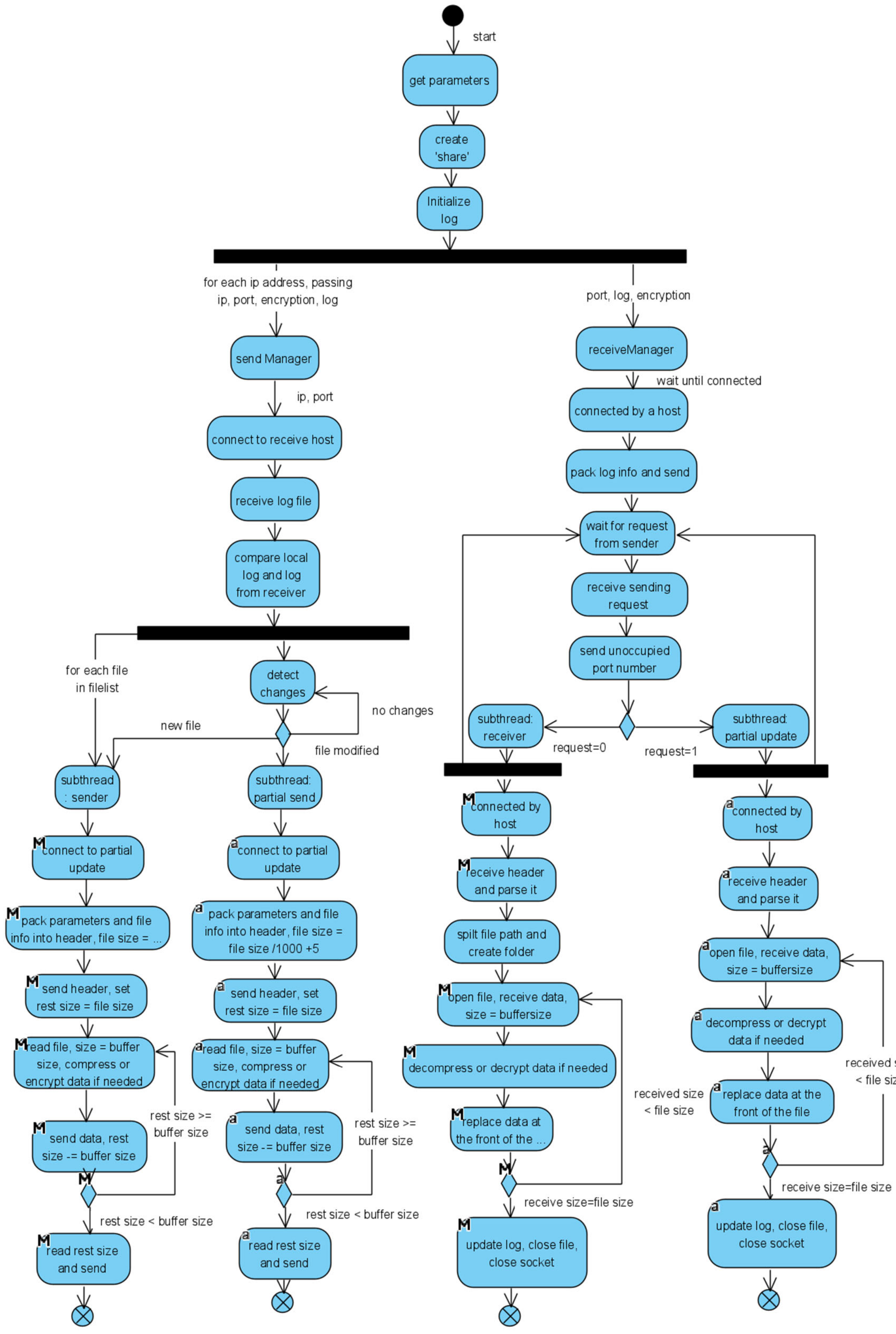
3. Implementation

Step of implementation

The plan of develop is firstly draw a mind map and analysis the requirements and program flow of the program. Then, build a program with basic function as prototype, and iterate based on the prototype, when a new function is written, it should be test separately at a independent environment, I use jupyter notebook to test the single function due to its cell mechanism is easy to test. After one function is done, it will be added to the whole program to test. After all requirement is exceeded, the code is upload to virtual machine to do the final test. The plan is aiming to find some potential problem of the program and find problem from the minimum unit ensures unit function is right and helps find problems in program design logic.

Firstly, point to point file transfer using TCP is developed, then some functions related to log is developed, json is chosen as the log storage type since it is easy to read. After that is detectChanges() and compDiff() function, which is necessary for synchronization. The sendManager() and receiveManager() is developed as the role of "gatekeepers". Finally, I add function like partial update to the program and test the whole program in three virtual machines.

Program flow charts:


start
get parameters
create 'share'
Initialize log
for each ip address, passing ip, port, encryption, log
port, log, encryption
send Manager
ip, port
connect to receive host
receive log file
compare local log and log from receiver
receiveManager
wait until connected
connected by a host
pack log info and send
wait for request from sender
receive sending request
send unoccupied port number
for each file in filelist
detect changes
no changes
new file
file modified
subthread: receiver
request=0
request=1
subthread: partial update
subthread : sender
subthread: partial send
connect to partial update
connect to partial update
pack parameters and file info into header, file size = ...
pack parameters and file info into header, file size = file size /1000 +5
send header, set rest size = file size
send header, set rest size = file size
read file, size = buffer size, compress or encrypt data if needed
read file, size = buffer size, compress or encrypt data if needed
rest size >= buffer size
rest size >= buffer size
send data, rest size -= buffer size
send data, rest size -= buffer size
rest size < buffer size
rest size < buffer size
read rest size and send
read rest size and send
connected by host
receive header and parse it
spilt file path and create folder
open file, receive data, size = buffersize
decompress or decrypt data if needed
replace data at the front of the ...
received size < file size
receive size=file size
update log, close file, close socket
connected by host
receive header and parse it
open file, receive data, size = buffersize
decompress or decrypt data if needed
replace data at the front of the file
received size < file size
receive size=file size
update log, close file, close socket
M
a

Programming skills

The program uses parallel computing, including multiple thread and multiple process to utilize the multiple core of the CPU, the program will create two process for send, each corresponding to one IP address and one process for receive, and create one thread for each file to receive or send.

Challenge and solution

One challenge met is “sticky package problem”, which refers to when receiving files, the receiver will receive more binary data since the file size could not always divide by receive buffer size exactly. The key point of the solution is to understand that data is actually stream, not a package. Therefore, we could update the ‘send size’ and ‘receive size’, add buffer size to them once before send or receive, when the receive/send size is about to exceed, then only send/receive the rest size of the binary data.

Another challenge I met is the memory problem, the program is designed to read the whole data and transfer them by multiple times and write file until received all data at early version. However, when testing, I found the transfer speed of large file like a 1.4GB video, is too low, then I open task manager to check the progress and find that the program use too much memory and python need to wait for memory recycle to continue. Then I change the send/receive way to read/write buffer size and send/receive buffer size to solve it.

4. Testing and result

Testing environment

The testing is proceeded in three virtual machines on windows, named VMA, VMB, VMC, using the environment “cw1.ova”, each virtual machine is allocated with 4 different CPU logical cores, and 2 GB of memory. CPU clock speed is locked at 4.6GHz for all cores to reduce the hardware difference.

Testing plan and result

The testing will divide into 5 parts and run three rounds, take average results, virtual machine will be restarted and cleared after one round test:

1. Multiple small files to test concurrency
2. One big file for stress test
3. Interrupt and resume test

4. Partial update test

Part 1:

Part 1 uses 20 pdf files, which is 1.37MB in total. For three round test, I will put folder in different VM. The time and speed is take each file's speed to calculate the average. The result is not accurate for true test since the start of threading also need time.

VM name	File size (bytes)	Transfer time (s)	Transfer speed (Mbps)
VMA	1437707	0.04896	228
VMB	1437707	0.03769	291
VMC	1437707	0.06855	160

Part 2

Part 2 uses a 1.08GB txt file, which start with 1KB random number and fill with "0" in rest part. For three round test, I will put file in different VM.

VM name	File size (bytes)	Transfer time (s)	Transfer speed (Mbps)
VMA	1162944224	10.95	810
VMB	1162944224	11.10	799
VMC	1162944224	10.48	846

Part 3

Part 3 uses a zip file which contains a video, For three round test, I will put folder in different VM, after about 2 seconds, kill one machine and after another is received, restart it, when all file is received, I will decompress the file to check whether it's broken.

VM name	File size (bytes)	Transfer time (s)	Transfer speed (Mbps)	Resume
VMA	510828408	3.3281	1171	True
VMB	510828408	3.0049	1297	True
VMC	510828408	3.6664	1063	True

Part 4

Part 4 use a 10KB txt file consist with zero. For three round test, I will put folder in different VM, when other received files, change the front from “000000” to “111111”, and check the file in other virtual machine. The transfer speed is for sending 10KB txt.

VM name	File size (bytes)	Transfer time (s)	Transfer speed (Mbps)	Partial update
VMA	10000	0.0048	16	True
VMB	10000	0.0016	43	True
VMC	10000	0.0013	55	True

5. Conclusion

To sum up, the program meets the “LEFT” requirements, although with some difficulties like “sticky package” and “memory exhausted” to resolve, the program achieve stable and considerable results after well-designed testing. Development methods like iterate on a prototype and unit test helps a lot. However, due to the time limitation, advanced functions like resume from the breakpoint, file slicing algorithm, synchronized delete, and real partial update is not complete, also, the encryption method is not complete though parameters are added to program already. The project still need further develop to solve these problems.

6. Reference

[1] Microsoft, 2020 “OneDrive Feature”, https://www.microsoft.com/en-us/microsoft-365/onedrive/free-online-cloud-storage/ [Accessed: 12/22/2020]

[2] BitTorrent, 2020 “Resilio usecase”, https://www.resilio.com/usecases/ [Accessed: 12/22/2020]