Distributed System Project 2

Group Name: **Not Safe For Workshop**
Group Members:

Member Name	Login Name	University Email
Yujun Zhang	yujuzhang	yujuzhang@student.unimelb.edu.au
Tong Su	tsu2	tsu2@student.unimelb.edu.au
Zhenyu Wang	zhenyuw6	zhenyuw6@student.unimelb.edu.au
Yu Bai	ybbai1	ybbai1@student.unimelb.edu.au

1. Introduction

This report aims to state the detailed communication between Server and Client, the encryption and decryption method and comparison between TCP and UDP.
The goal of the project is to let the BitBox Client communicate with a BitBox Peer, then let this Peer to interact with other BitBox Peers. The communication commands between Clients and Server(Peers) are sending securely using RSA public/private key and AES cryptography. Peers themselves could also choose to communicate with other BitBox Peers. The transport protocol between Peers can switch to TCP or UDP.

2. Server (Peer) and Client communication

2.1. Relationship Between Client and Server (Peer)

In this project, the connection between Peer and Peer is the same as Project 1, but the transport protocol can switch to TPC or UDP. All the Peer are initialized by running the terminal command:

```
java -cp bitbox.jar
unimelb.bitbox.Peer
```

There is a new role Client added in this project. For the Clients, Peers stand for the role of Server. The transport protocol between them is TCP. Each Client can connect with one Peer, and require this Server Peer to connect with many other Peers. Each Peer can be connected by many Clients at the same time. The graph below shows a simple situation, connections are separated by different colours:

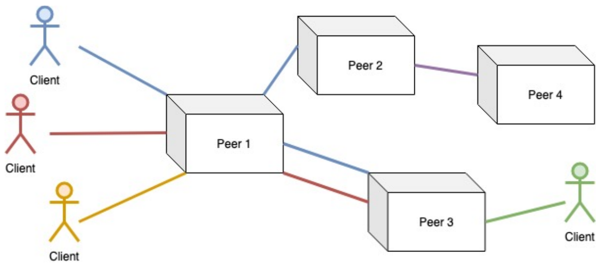


Figure.1 Clients and Peers

2.2. Clients' Command Illustration

The Client will initialize the connection with Server Peer by executing commands. All the information will be stored and sent step by step. The format of commands is like:

```
Java -cp bitbox.jar
unimelb.bitbox.Client -c
connect_Peer -s server.com:3000 -p
target.com:8500 -i
Giovanna@Giovanna
```

The string after "-c" stands for the operation which the Client will ask the Server Peer to do. There are three operations can be executed in this project, "list_peers", "connect_peer" and "disconnect_peer".
The address after "-s" stands for the Server Peer. And the address after "-p" stands for the target connection Peer.
The string after "-i" is the identity name of the Client. In this project, the sample Client identity name is "Giovanna@Giovanna".

2.3. Communication Description

After running the command in Terminal, the process between Client and Peer can see in the graph below:

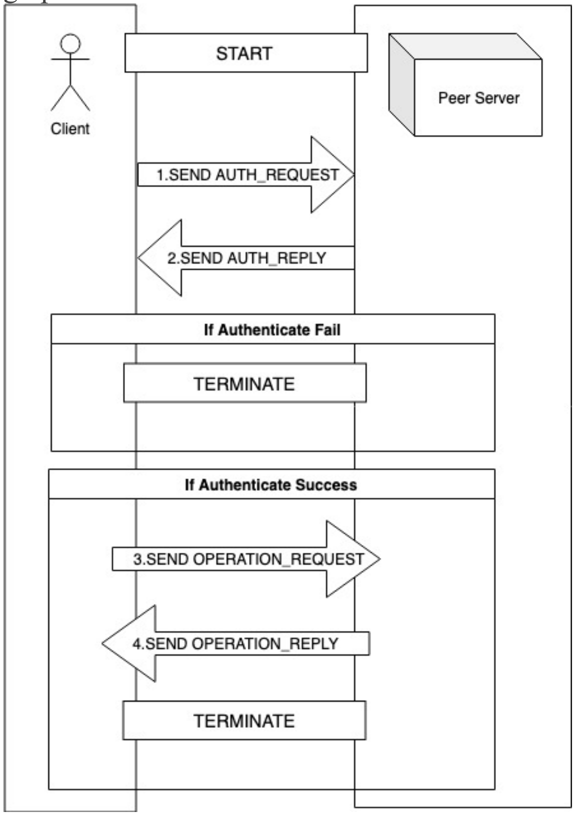


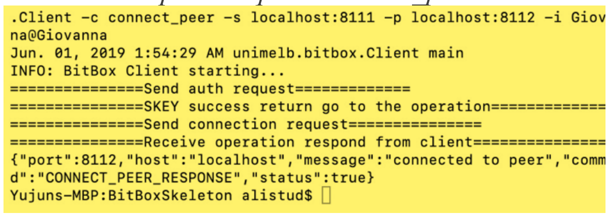
Figure.2 Client to Peer process

In step 1, the Client sends AUTH_REQUEST to the Server with it identify name. The Server will check whether this identity name have a corresponding public key stored in its public key list. If the Server finds the public key, it will generate a secret Key. Otherwise, a fail authorization reply will be sent.
In step 2, the server uses the Client's public key to encrypt the secret Key in

AUTH_REPLY and send it back. Once the Client read the reply, it can use its private key to get the shared secret key.
Then according to the operation, the Client will send corresponding OPERATION_REQUEST to the server. And the Server will reply corresponding OPERATION_REPLY. Both of these messages are encrypted and decrypted by the common secret key. Then the sending message process is ending. If the Client wants to send another operation, he or she needs to send a new command in the same process.
Once the "connect_Peer" initialization successfully, the Client can always modify files in the Peer to Peer share file fold until Peer closed or Sending the "disconnect_Peer" operation. Then the connection between Server Peer and Target Peer will be cancelled.
The interact messages snapshots are shown below:

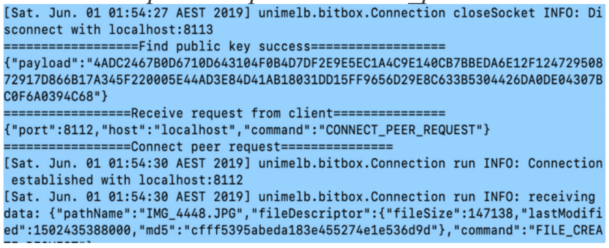
Situation 1: connect to the target Peer

Client part: request connect_peer



```
.Client -c connect_peer -s localhost:8111 -p localhost:8112 -i Giov
na@Giovanna
Jun. 01, 2019 1:54:29 AM unimelb.bitbox.Client main
INFO: BitBox Client starting...
===============Send auth request============
===============SKEY success return go to the operation============
===============Send connection request===============
===============Receive operation respond from client==============
{"port":8112,"host":"localhost","message":"connected to peer","comm
d":"CONNECT_PEER_RESPONSE","status":true}
Yujuns-MBP:BitBoxSkeleton alistud$ 
```

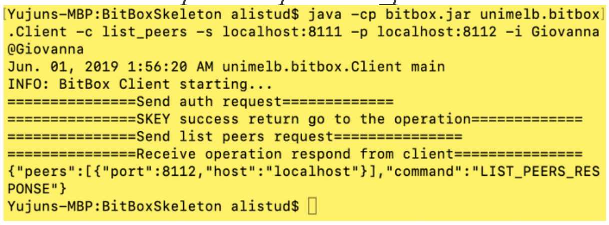
Peer part: respond connect_peer



```
[Sat. Jun. 01 01:54:27 AEST 2019] unimelb.bitbox.Connection closeSocket INFO: Di
sconnect with localhost:8113
=================Find public key success=================
{"payload":"4ADC2467B0D6710D643104F0B4D7DF2E9E5EC1A4C9E140CB7BBEDA6E12F124729508
72917D866B17A345F220005E44AD3E84D41AB18031DD15FF9656D29E8C633B5304426DA0DE04307B
C0F6A0394C68"}
================Receive request from client==============
{"port":8112,"host":"localhost","command":"CONNECT_PEER_REQUEST"}
================Connect peer request==============
[Sat. Jun. 01 01:54:30 AEST 2019] unimelb.bitbox.Connection run INFO: Connection
 established with localhost:8112
[Sat. Jun. 01 01:54:30 AEST 2019] unimelb.bitbox.Connection run INFO: receiving
data: {"pathName":"IMG_4448.JPG","fileDescriptor":{"fileSize":147138,"lastModifi
ed":1502435388000,"md5":"cfff5395abeda183e455274e1e536d9d"},"command":"FILE_CREA
```

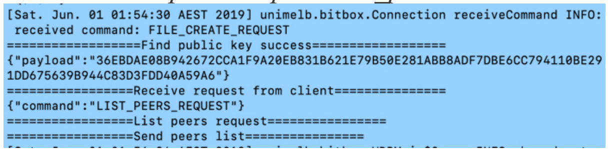
Situation 2: list all the connect Peers in the Server Peer

Client part: request list_peers



```
Yujuns-MBP:BitBoxSkeleton alistud$ java -cp bitbox.jar unimelb.bitbox
.Client -c list_peers -s localhost:8111 -p localhost:8112 -i Giovanna
@Giovanna
Jun. 01, 2019 1:56:20 AM unimelb.bitbox.Client main
INFO: BitBox Client starting...
===============Send auth request============
===============SKEY success return go to the operation============
===============Send list peers request===============
===============Receive operation respond from client==============
{"peers":[{"port":8112,"host":"localhost"}],"command":"LIST_PEERS_RES
PONSE"}
Yujuns-MBP:BitBoxSkeleton alistud$ 
```

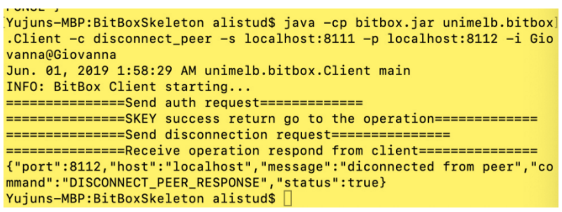
Peer part: respond list_peers



```
[Sat. Jun. 01 01:54:30 AEST 2019] unimelb.bitbox.Connection receiveCommand INFO:
 received command: FILE_CREATE_REQUEST
=================Find public key success=================
{"payload":"36EBDAE08B942672CCA1F9A20EB831B621E79B50E281ABB8ADF7DBE6CC794110BE29
1DD675639B944C83D3FDD40A59A6"}
================Receive request from client==============
{"command":"LIST_PEERS_REQUEST"}
================List peers request===============
================Send peers list===============
```

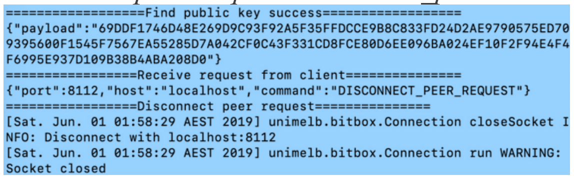
Situation 3: disconnect to the target Peer

Client part: request disconnect_peer



```
Yujuns-MBP:BitBoxSkeleton alistud$ java -cp bitbox.jar unimelb.bitbox
.Client -c disconnect_peer -s localhost:8111 -p localhost:8112 -i Gio
vanna@Giovanna
Jun. 01, 2019 1:58:29 AM unimelb.bitbox.Client main
INFO: BitBox Client starting...
===============Send auth request============
===============SKEY success return go to the operation============
===============Send disconnection request===============
===============Receive operation respond from client==============
{"port":8112,"host":"localhost","message":"diconnected from peer","co
mmand":"DISCONNECT_PEER_RESPONSE","status":true}
Yujuns-MBP:BitBoxSkeleton alistud$ 
```

Peer part: request disconnect_peer



```
=================Find public key success=================
{"payload":"69DDF1746D48E269D9C93F92A5F35FFDCCE9B8C833FD24D2AE9790575ED70
9395600F1545F7567EA55285D7A042CF0C43F331CD8FCE80D6EE096BA024EF10F2F94E4F4
F6995E937D109B38B4ABA208D0"}
================Receive request from client==============
{"port":8112,"host":"localhost","command":"DISCONNECT_PEER_REQUEST"}
================Disconnect peer request==============
[Sat. Jun. 01 01:58:29 AEST 2019] unimelb.bitbox.Connection closeSocket I
NFO: Disconnect with localhost:8112
[Sat. Jun. 01 01:58:29 AEST 2019] unimelb.bitbox.Connection run WARNING:
Socket closed
```

Because the Secret Key is randomly generated every time, it only works inside one command progress. Even the secret key is filched by attackers, this secret key cannot use for decrypting the messages in the later processes. It's safe to encrypt and decrypt the requests and responds between Clients and Peers.

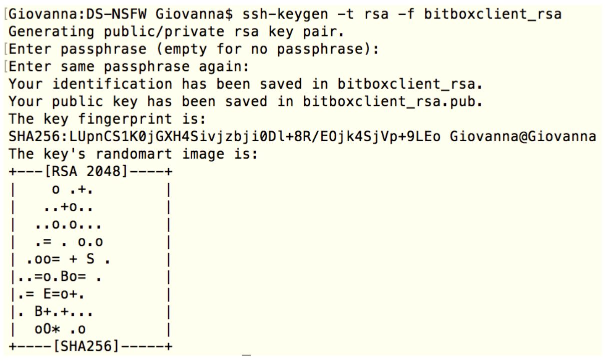
2.4.Limitation

Once the Client successfully connects with the Server Peer, he or she can have the right to modify all the files in the shared file until sending the disconnecting. Meanwhile, an active Server Peer cannot choose to disconnect with a specific Client. This might cause some inconvenient for the Server Peer part. Besides, files in Server Peer's folder are easy to be modified. If there are many Clients connect at the same time, it might cause messy in the folder. In the future improvement, the file access authority of Clients might need to build, for example, only can read files, cannot delete or update and so on.

3. Public/private key and Encryption Technique

3.1.Assumption

For this project's purpose, we will assume that the Client will generate the public and private key pair under the same directory with 'bitbox.jar' file. This key pair is generated by the command "ssh-keygen -t rsa -f bitboxClient_rsa" which is shown in Figure 3. For simplicity, there is no passphrase when generating these keys. Also, we assume that the Client will use "bitboxClient_rsa" for the private key and the public key will be "bitboxClient_rsa.pub".



```
Giovanna:DS-NSFW Giovanna$ ssh-keygen -t rsa -f bitboxclient_rsa
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in bitboxclient_rsa.
Your public key has been saved in bitboxclient_rsa.pub.
The key fingerprint is:
SHA256:LUpnCS1K0jGXH4Sivjzbji0Dl+8R/EOjk4SjVp+9LEo Giovanna@Giovanna
The key's randomart image is:
+---[RSA 2048]----+
|    o .+.        |
|   ..+o..        |
|  ..o.o...       |
|  .= . o.o       |
| .oo= + S .      |
|..=o.Bo= .       |
|.= E=o+.         |
|. B+.+...        |
|  oO* .o         |
+----[SHA256]-----+
```

Figure 3. Generate the public and private key

3.2.How to use

The public and private key technique used in this project 2 is RSA which is generated by “ssh-keygen”. Clients need to generate their own RSA key pair before sending a connection request to Peers. RSA is an asymmetric algorithm for encryption and decryption. It is asymmetric because it needs two different keys to separately encrypt and decrypt the message. For this to work securely, the public key is given to a trusted server or people and private key has to be stored safely and no one else should know it except the owner who generated this key pair. So that the person with the private key could communicate with people with public key secretly.
AES algorithm is also used in this project. This is a symmetric algorithm so only one key is required for both encryption and decryption which is more efficient than RSA. In order to transfer this secret key to the Client safely and securely, we need to use the Client’s RSA public and private key pair.
For this project, the Client will generate this RSA key pair and give his public key to the Peer that the Client wishes to connect with. Then the Client could send a connection request to the Peer who has his public key. This Peer will need to check whether his configuration.properties' file contains this Client’s public key. If this Client’s public key exists, then the Peer could use AES algorithm to generate a secret key and use the Client’s public key to encrypt this secret key. So only this Client with his private key could decrypt this message and get the secret key.

3.3.How it works

The public and private key generated on local files are in PKCS#1 PEM format which is not compatible to use in Java. In order for Client to read their private keys in a local file, first need to read the whole key string into java, delete “-----BEGIN RSA PRIVATE KEY-----” and corresponding footer parts in PKCS#1 format, decode it using Base64 and convert it to PKCS#8 key format which Java could understand. Then convert from PKCS#8 key format to Java PrivateKey format for “javax.crypto.Cipher” library to encrypt and decrypt.

3.4.Limitations

For now, the public key is being sent to Peer manually which is not convenient and is not secure enough. A malicious attacker or eavesdropper may get the public key so that their communication is not secure now, as other people could use the public key to decrypt Clients’ message. For further enhancement, we could use trusted CA to securely send the public key by assigning digital certificates. Also, in the current project, reply attacking may still occur as other people could get the encrypted message, even they may not be able to decrypt it, they can keep sending interrupt things to Peers as a fake Client which may cause the server to crash.

4. Comparison between TCP and UDP

4.1.UDP Implementation

UDP has been implemented in project 2. The class “UDPMain” is created for listening to incoming connections. It provides ports for new connections. The new connection established uses a different port from the listening port because UDP generates port at random. If there is a data packet loss when transmitting a file using UDP, the system will resend the file at most three times. It will lead to failure in transmission if three transmissions are all unsuccessful and then the part has already been transmitted will be deleted.

4.2.Comparison

UDP and TCP are both network protocols work based on IP (Internet Protocol) that are used to send data packets. Also, they both transmit data packets using ports. TCP is connection-oriented and UDP is connectionless. This means there should be a connection established between the server and the Client before sending TCP packets. For instance, a three-way handshake is used to establish a connection in Project 1. On the other hand, UDP does not have such a connection. The packet in UDP is sent individually and directly from the sender to the receiver without a stable connection. As a result, TCP is more stable when transmitting the data packet while there is a high chance that UDP will lose data packet when transmitting the file as we did in project 2. In addition, a sequence number is in the data packet when we are implementing TCP in Project 1 so that TCP has the ability to know the order of the packet. However, UDP does not add a sequence number so it is impossible for the receiver to know the sequence of the packet. What's more, TCP has error detection and correction methods. For example, the sender needs to resend the packet if it does not receive an acknowledgement in TCP in project1. In UDP, it has the ability to detect the error but cannot correct the error. As mentioned above, TCP needs more requirements than UDP so that the speed of UDP is faster than TCP. Conversely, TCP is more reliable than UDP because it provides data packet sequencing, acknowledgments, error detection and error correction. In conclusion, TCP may be more suitable for reliability is more important, (e.g. file transfer). UDP is more suitable for speed is more important (e.g. live streaming).