# Group 14 Project 5: P2P File Sharing

Dylan Callaghan: 21831599@sun.ac.za
Stephen Cochrane: 21748209@sun.ac.za

November 2, 2020

## 1 Introduction

In this project, the requirement was to implement a p2p file sharing program, that offered anonymity but also security for the sharing of files. The framework was to consist of one server with many clients, where all the clients were connected to the server. The server's role was to manage interactions between the clients by means of message passing (both text and commands). The clients, once introduced by the server, should then be able to transfer files p2p, from one client to the other without server interference. The client should have at least the functionality to support one concurrent upload and download each. The client would search for files with a server command which will return the results from each client connected.This client could then choose which file to download and begin after some security measures. Optional features were to implement multiple uploads and downloads (more than one each), and a text based chat.

## 2 Unimplemented Features

Of all of the features outlined (mostly in the introduction above), all were implemented.

## 3 Additional features

For this project, we implemented three extra features, namely client text messages, multiple concurrent uploads and downloads, and upload forking and restarting. We also exceeded the minimum requirement for security, implementing a very secure framework including multiple technologies. The security framework, however, will be described in section 9.1

### 3.1 Multiple uploads/downloads

For our implementation, we decided on a very general approach for most parts of the system. The Gui, and functionality for uploads and downloads is one

of these parts, and this allowed us to easily extend our program to support multiple uploads and downloads. The implementation works by assigning each upload/download it's own thread. Whilst this transfer (upload/download) is busy, it will continue to operate without disrupting the system. All of these transferring threads are collected together in a data structure called ActionList (see section 8.1), which allows the client application to manage each transfer, and retrieve information. This data structure has built in thread safety, as well as various features to allow for all the features of the client.

### 3.2 Text messages

The client, along with being able to search for files through the server, also has the ability to send text messages to the group of p2p clients. This additional functionality provides clients the ability to talk to the rest of the p2p group, and which adds value to the program. This additional feature was possible due to the general design of the server. The server implementation deals only with server messages, with certain tags. Most of these tags are for the transferring of files, however a new "message" tag was easily implemented and framework set up on the client to deal with these messages.

### 3.3 Upload forking and restarting

For our implementation, we wanted the application to be as similar as possible to an actual implementation of p2p found on the internet today. For this reason, we think of our uploads as "seeded files". Seeding is a concept in most running p2p file sharing programs that describes files hosted on a peer's computer, which he/she is willing to share. These files can be shared multiple times and with anyone on the p2p network. In our implementation, we support uploads as seeding by ensuring that uploads can be shared by multiple users (through forking), and will continue to share after one user has downloaded (by restarting). Forking happens when a peer is uploading a file that another user is already downloading, and in the event that another user wants to download the same file. Since there is only one file being hosted, in this situation, normally the second user would have to wait. However, our application deals with this by forking the active upload to allow for a new upload thread of the same file. This thread can then be used to download this file for the second user simultaneously.

## 4 Description of files

- Client Holds all logic for the communication between the separate client functionalities. This file contains methods that are called by other components of the client, and calls components in these methods, providing a bridge between the different aspects.
- ClientFuncPass The abstract interface that the Client uses to facilitate two-way communication between it's components.

- User Class for the User object. This object is a simplistic structure mainly used for Gui displaying and for the transformation between nickname and IP address.
- Gui The Gui component of the client. This is a modular Gui which communicates with the client when receiving or sending updates. The Gui handles only drawing of the client graphically and does not have any logic for the client.
- Transfer This class is modified class from project 2, it allows for setting up a peer to peer transfer between two clients.
- Connection Abstract class used by Transfer. Represents a transfer method.
- TCP : extends Connection. This is again a modified class from project 2, although instead of sending raw byte arrays, it sends chunks wrapped in an encrypted Datagram packet.
- Message The `Message` class acts as a wrapper for byte arrays allowing for easy creation of messages (or reading of) for the `ServLink`. This class has 3 fields, in order they are:
  - tag The tag specifies the type of message.
  - len The length of the payload.
  - payload The payload being sent.
- InitTransfer : Wrapped inside a Message packet. This class allows for transferring the data required for establishing a transfer between two clients. All the data is encrypted using RSA.
- Datagram : Wrapped inside a Message packet. The datagram class allows for easy transfer of packets that are intended to be encrypted. And also allows for reading in a byte stream and then decrypting it. The encryption and decryption is optional. And the method used is Transpose Encoding. When encrypting and decrypting the same key must be used, or else the data will be unpacked incorrectly. Due to the nature of transpose encryption adding padding, there is a field `actual_len` which can be used to determine which bytes are actually meant to be read.
- Transpose This class has two static methods, namely encode and decode that both use a key to either encrypt a byte array or decrypt a byte array. This class is used by the Datagram class.
- RSA This class holds the RSA public/private key pair. For more description on this object, see 8.3.
- Server This class is used to run the server, when started it blocks on accepting new client connections. When a client gets accepted a handler process gets forked (and the server will continue blocking until a new client attempts to connect), and all this process does is forward messages

- ServLink This class provides a link to the server, acting as a form of structured message passing, using the `send` and `recv` functions. ServLink communication has a guarantee of a send, and recv is blocking. Servlink just sends and receives streams of bytes, and hence makes use of the helper class `Message` to assist in this. From one client to other connected clients.
- optargs Simple argument parsing, makes handling command line arguments trivial. Very similar to `getopt` in C.

# 5 Program description

For the client, we implemented a mediator design pattern to facilitate communication between separate modular components of the system. The mediator in our case was the general client class, which had components:

- Server link
- Gui
- ActionList object

The client had a general interface which it used for two-way communication between each component. Each component was therefore responsible for it's own functionality, and passed "messages" to other components when they had finished their tasks. This allowed for a very simplistic design, which allowed us to abstract away the full functionality of the system, and just consider the individual modular components. Each component was responsible for interacting with it's part of the system, and in a sense communicating with it's object. In the case of the Gui, this was the graphical display, the ActionList communicated with each ongoing upload/download, the Server link with the server. We used a very simplistic server, that just facilitated communication and sent messages. The server only receives messages from clients and passes them on to other clients. These messages can either be text messages, file searching messages (search request and hit), or file request handshake messages. In this way, the server was used as a look-up and facilitator to the connection between clients. It deals with all set up between clients whether it be for transfers or for connecting to the server. For client connection, the server forks a thread that will set up a connection with the client and maintain this. The server also notifies all other clients of the new user. For transfers, the transfer handshake messages are sent through the server, as well as the pausing and resuming messages for the transfer. For a more in depth description of each component in the system, see sections 4 and 8.

# 6 Experiments

## 6.1 Security with transfers

For this series of experiments, we conducted multiple tests, both with individual classes, and the full application to determine whether the security framework we had was secure.

### 6.1.1 Expectations

We expected RSA to be a secure encryption method which we could use for sending keys to the recipient encrypted and safely. We also expected the transpose method to encrypt the data so as to be non-recoverable without the key used.

### 6.1.2 Findings

In our series of experiments, we found mostly what we expected. The RSA protocol was very secure, however we found errors trying to encrypt a longer key. We also found issues vulnerabilities in our framework, including the lack of security without the digital handshake (as anyone could pretend to be the sender). We also found that the transposition cipher was secure, but lost it's security after using the key multiple times.

### 6.1.3 Conclusion

From these results, we deduced firstly that the keys sent by RSA needed to be shorter than the length specified by the RSA protocol according to the public key used. We therefore had to adjust our framework slightly. We also added the digital handshake (three-way message handshake), which solved the issue of malicious senders. For the transposition cipher, we concluded that one-time pads needed to be used in order to make the cipher secure. To do this, we added key sending in each message, so that each message was encoded with a different OTP.

## 6.2 Local file transfer

In this experiment, we tested in the controlled environment of one local computer, sending from one application to another on this computer.

### 6.2.1 Expectations

We expected the system to work as intended, and for the upload and download speed to be almost instantaneous.

#### 6.2.2 Findings

The results were that the system worked as intended (after solving some bugs related to opening and saving of files). The transfer rates were almost instantaneous for small files, but for large files they started growing bigger.

#### 6.2.3 Conclusion

The conclusion from these results was that this was expected behaviour of the system. This use of the system was essentially providing a file moving application within the computer. Like any given file moving application, the transfer rates for larger files will increase, whereas smaller files would be almost instantaneous. Thus we concluded that our application was running as it should locally.

### 6.3 Hamachi file transfer

In this experiment, we sent files over hamachi to test file transfers over a network.

#### 6.3.1 Expectations

Here we expected significantly lower transfer rates than in the local file transfers. We expected some delays and inconsistencies, but other than that we expected a stable file transfer with the full file sending properly.

#### 6.3.2 Findings

We found that files sent fully as expected with no issues in the files contents, but there were significant delays in some situations when sending over hamachi. These delays were spontaneous, and sometimes occurred in the middle of transferring files, where the beginning of the transfer was faster.

#### 6.3.3 Conclusion

Our conclusion from these results was that these fluctuations are caused by the hamachi network itself, as they are irregular, and not dependent on any of our applications functions. Therefore, we concluded, that given a stable network to operate on, our application was functioning correctly

## 7 Issues encountered

The only issues we encountered was that of hamachi being hamachi (that is insane packet loss and or connection issues).

# 8 Significant Data Structures

## 8.1 ActionList

This data structure stores a list of all active uploads/downloads. It's use is to provide a global list of all active uploads and downloads, for the use of the entire client system. Because of this (and the fact that this is a multithreaded implementation), the list is implemented to be thread-safe. It ensures that any thread accessing the list is the only thread accessing it at that given time. Beyond this, it allows every upload and download to be maintained and removed, and interfaces with the Gui to remove the upload or download respectively. The data structure contains a list of action objects, which are threads that can be started to begin their containing upload/download. This provides an easy way for any part of the application to start/pause/resume downloads, by just going through this data structure.

## 8.2 User

This data structure is used in many places in the program, and stores a single user, and related data. The user data structure comprises of:

- A unique nickname
- An IP address-port pair (also unique)
- A colour (for the Gui)
- An RSA key-pair (either public and private, or just public)

Each time a user needs to be used in the application, a User object from the above is constructed. This ensures that when dealing with a user, all information (or useful information at the time) is associated with the object. Some fields (such as RSA key pair, and nickname), can be left blank if not needed.

## 8.3 RSA

The RSA key pair is a sub data structure which contains the public (and private) keys of a certain user. The structure can be used to create a new public private key pair, after which only the public key can be retrieved, or can be made given a public key, in which case the object can only encrypt. The data structure provides methods to encrypt (using the public key), and decrypt (using the hidden private key).

# 9 Design

## 9.1 Security

### 9.1.1 RSA

The first security layer is a digital handshake protocol that facilitates the sharing of keys securely. This layer uses an RSA encryption to maintain high levels of security. The protocol uses a three-way handshake between the user requesting the download (the receiver) and the user hosting the upload (the sender). The three-way handshake (including preliminary steps) is described here.

- To initiate the whole process, the receiver first searches for the file through server.
- The server will return with a list of files matching (or substring matching) the search. These messages will contain the actual result's file name, the nickname of the person hosting the file (the packet framework used also contains the IP address but this is hidden from the clients), and the sender's public RSA key to be used for the handshake.
- The receiver will then initiate the handshake by sending the first of a set of request messages making up the three-way handshake. This first message we call i1. The i1 message contains a number ($i$) encrypted using the sender's public key, and then the receiver's public key appended to this. The sender and only the sender will receive this message and can decrypt it, extracting the number $i$.
- The sender will then respond with an i2 message, which is the second handshake message. This message contains a number, which is $i + 1$, encrypted with the receiver's public key (from the i1 message). The receiver and only the receiver can then receive this message and decrypt it, extracting $i + 1$.
- The receiver then checks that the received $i + 1$ is equal to the original $i$ plus 1. If this is not the case, it will merely drop the message, preventing any attacks. If it is the case, it will send an Init (i) message, which contains the first key (see 9.1.2) for the file sharing encrypted using the sender's public key, as well as the file name of the file the receiver wants to download.

This process above ensures that the sender and receiver are authenticated before any transfer has started, and that the key for the sending of the file is sent securely to the correct peer.

### 9.1.2 Transpose

We make use of the transpose algorithm with a one time pad on each datagram. When a datagram is encrypted it generates a new 96 byte key, which is

concatenated with the packet data and then encrypted using the previous key, to be sent. When we encrypt the next packet we will then use this previously generated key, so that each key is only used once, and each packet contains the next packets encryption key. This process continues until all packets have been sent. The receiver will decrypt the first packet using the initial key (that was received during the handshake), from the decrypted packet we then read the next key and use that key to decrypt the next packet, This process continues until all packets are decrypted. Thank you TCP for in-order delivery.

This allows us to have a one time pad on each packet, basically forming a daisy chain. Since we make use of a handshake (explained above in RSA), this provides two things,

- Our own form of "SSL", which ensures we are protected against a man in the middle attack.
- RSA encryption of the initial 96 byte key to be used by the transpose encryption.

This means that the only way to decrypt the data would be to obtain the initial key, the only way this can be achieved is either a brute force decryption of the RSA encrypted message which contained the initial key. Or by guessing the Initial key. Both of these options require a large amount of computational power, as well as taking a long (as in really long) amount of time. Another bonus is since we include the next key inside the data payload and then encrypt, is that the key is mixed in with the data, making it is impossible to determine the difference between the two, and hence impossible (or highly improbable) to determine the next key.

# 10 Usage

When connecting to the server, to host a file, you click on the plus, and follow the gui prompts. To remove a hosted file, you must simply click the little "x". To search for a file simply type `/s <substring search>`, All search hits will be shown, to download a search simply click on the desired file.

# 11 Compilation

```
$ make
```

# 12 Execution

## 12.1 Server

### 12.1.1 Running

```
$ ./Server [options]
```

#### 12.1.2 Help

```
$ ./Server -h
```

### 12.2 Client

#### 12.2.1 Running

```
$ ./Client [options]
```

#### 12.2.2 Help

```
$ ./Client -h
```

## 13 Libraries used

- java.awt.*;
- java.io.*;
- java.lang.String;
- java.net.*;
- java.nio.ByteBuffer;
- java.util.ArrayList;
- java.util.HashMap;
- java.util.Hashtable;
- java.util.Iterator;
- java.util.LinkedList;
- java.util.Objects;
- java.util.Scanner;
- javax.swing.*;