A FILE SHARING PROGRAM

October 25, 2019

Jacques Huysamen - 20023669
Gerrit Burger - 21261687
Computer Science 354 - Networking

Contents

1 Project Description

1.1 Introduction

Peer-to-peer file transfer refers to the distribution of digital media using a peer-to-peer network connection. This technology enables users to search for files that other users might have and request to download files. If a client requests to download a file and the file transfer is accepted a connection is established directly between the two clients and the file is transferred over this connection. To use peer-to-peer file transfer software is needed to enable users to locate other users that have the file a client has requested, these clients that have the file are called peers.

1.2 Problem statement

We were instructed to implement a peer-to-peer file transfer system, similar to programs like DC and Vuze. A server was implemented that handles the interaction between clients and the initialization of clients. The server did however not play any role in the transferring of files itself. A client program was implemented to enable users to search for files from other clients and download these files.

2 Project Features

2.1 Client Features

Required features included for the client:

- Ability to search files on the systems of the other clients
- Searching returns exact matches as well as substring matches
- Download and upload files from and to other clients
- Progress bars for the uploading and downloading of files
- The ability to pause and resume uploads and downloads
- The ability to send private messages and broadcast messages

2.2 Server Features

Required features included for the server:

- Enable clients to choose a unique nickname when they connect.
- Sends out search requests to all clients when a client searches for a file.
- Assigns a port number to be used when a file is transferred.
- Alerts a client that is uploading if the downloading client has paused the file transfer.

3 Program Description

3.1 Server

3.1.1 Initialization

When the server program is run and a new instance of a server is created the local machine's IP address is found and set as the server's IP address. A pool of port numbers is then created, ranging from 8000 to 9000, to be used for the peer-to-peer sockets. For this project, we used encrypted sockets to enable secure communication between clients and the server. The client will wait for clients to connect to the server when a new client connects the encrypted socket connection will be accepted and the relative information about the client will be stored. A new thread will then be spawned for each client that connects to the server that will handle all communication between that client and the server.

3.1.2 Client connection

When a client connects to the server a secure socket is set up using Python's SSL library to wrap the normal socket. For the secure sockets, we used the TLSv1.2 cipher suite which uses Elliptic Curve Diffie-Hellman to encrypt the packets sent over the sockets. Before the client allows a user to continue it validates the user's username to see whether the name is invalid or might be in use, should the username be in use the server tells the user to please specify a different username. As soon as the user is then connected they are added to the list of current users connected. The server spawns a thread for each user that connects to it.

3.2 Client

3.2.1 Initialization

When the client program is run the GUI will start and the client will be prompted to enter the IP address and port number of the server the client wishes to connect to. If these values are valid and the client can successfully connect to the server the client will be prompted to enter a username to be used. If the entered username is already in use by the other user connected to the server the server will notify the client and the client will be prompted to enter another username. Once a valid username has been entered the main GUI will open and all the needed sockets connections between the server and client will be initialized. A list of all the users currently connected to the server will be sent to the client and will be shown on the client's GUI.

3.2.2 Handling client searches

When a client wants to search for a file, they specify the file name that they want to see. This filename is sent to the server and is relayed to all of the other clients. Each client then runs through their files to search for the specified file. This search is executed using the library called "fuzzywuzzy" which returns resulting files with a score showing how relevant they are, we picked a relevancy score of 70 percent as a cut off, but the scores were also influenced by a small change we made that sees how many of the characters of the specified file name and the file names on the systems of the other clients match, giving scores to matched character to ensure that substring matches are more accurate. These results are returned to the server and then all put together to be sent to the client who asked for the file.

3.2.3 Security

All socket connections used were encrypted using the python SSL library. The library acts as a socket wrapper that wraps the normal python socket objects to automatically encrypt and decrypt all data sent over the socket. Elliptic-curve Diffie-Hellman key exchanges were used, each client has an elliptic-curve public-private key pair used to establish a connection with other clients. Both clients and servers use the dhparam.pem file to generate the keys to be exchanged. The file contains fields prime p and a generator g, the server uses the formula $B = g^b p$ to calculate parameters to send to a connecting client. The server sends (B,g,p) to the client who computes $A = g^a p$ and $K = B^a p$. A gets sent back to the server and the server computes $K = A^b p$. Because $A^b = g^{a*b} = g^{b*a} = B^a p$

both the server and client will agree on a shared key used to decrypt and encrypt data sent over the socket.
In the case of a peer-to-peer connection between two clients being set up, the above method will be used where one client will take the role of the server when connecting.

3.2.4 Handling client downloads

When the results are returned to the client, the client has the option of choosing one of the files that he wants to download. After choosing the file the client sends a request through the server to the other client asking the client to upload the file. The file is sent straight from one client to the other via the secure sockets.

3.2.5 Handling pausing and resuming

When a client decides to pause a download they send a message to the server which relays the message to the client which is uploading and tells them to stop sending data packets, they then close their sockets and their thread, at the same time the client which is downloading the file stops receiving data and closes its sockets and thread.

As soon as the client then chooses to resume the download it sends a message through the server to the other client which then seeks the partially written file to see where it stopped sending and the find the position of the bytes at which it is supposed to start sending again, at the same time the client which wants to resume re-opens a thread and socket and waits for the other client to send data over the secure socket, the client appends the data received to the partially received file and finishes building the file.

4 Data Structures

Since we used python all of the data structures are in the form of dictionaries because their key-value pairing makes it very easy to work with stored data and retrieving necessary data from them.

4.1 Data Structures

4.1.1 Server Data Structures

On the server-side of the project a variety of data structures were used to keep track of data about users, their sockets, addresses, etc. and an array for ports.

The following is the dictionaries we had and their uses:

- We had a dictionary called client which had a key which was the socket of a user and it had a value which was the nickname of the client.
- We had a dictionary called users that had a key which was the nickname of the client and had a value which was the IP address of the client.
- We had a dictionary called ip_to_name which had a key which was the IP address of the client and value which was the nickname of the client.
- We had a dictionary called ip_to_sock which had a key which was the IP address of the client and value which was the socket of the client.
- We had a dictionary called addresses that had a key which was the socket of the client and value which was the IP address of the client.
- We had a dictionary called nickname which had a key which was the nickname of the client and value which was the socket of the client.

We also used an array used to store generated ports which were allocated to newly created sockets for the uploading and download.

4.1.2 Client Data Structures

As the client program mostly only makes requests to the server, not many data structures were used. Most of the data structures were used for the GUI, here as a few examples of data structures used:

- When a client sends a request to search for a file and the server replies with possible search results a list will be updated and displayed in the client GUI.
- A few booleans were used to communicate between threads to pause and resume the downloading and uploading of files.
- A class instance is created each time a download is initiated. This class will open the socket on which file data will be received, open the file and receive all the file data and write the data to the file.
- A class instance is created each time an upload is initiated. This class will open the socket on which file data will be sent, open the file that must be transferred and read and send the file over the socket.
- Each time a new client connects to the server a dictionary is sent to all clients to update their online user list on the GUI.

5 Experiments

5.1 Searching

5.1.1 Hypothesis

Searching files on other clients' systems will give exact results as well as a substring in the case of not giving the whole name or spelling mistakes, but when the spelling mistakes are too high there are no results.

5.1.2 Method

Firstly exact file names were specified and the results were observed, then only a part of the file names was specified and the results were observed and lastly, some spelling mistakes were specified and the results were observed.

5.1.3 Results

The following files were available on the client's system:

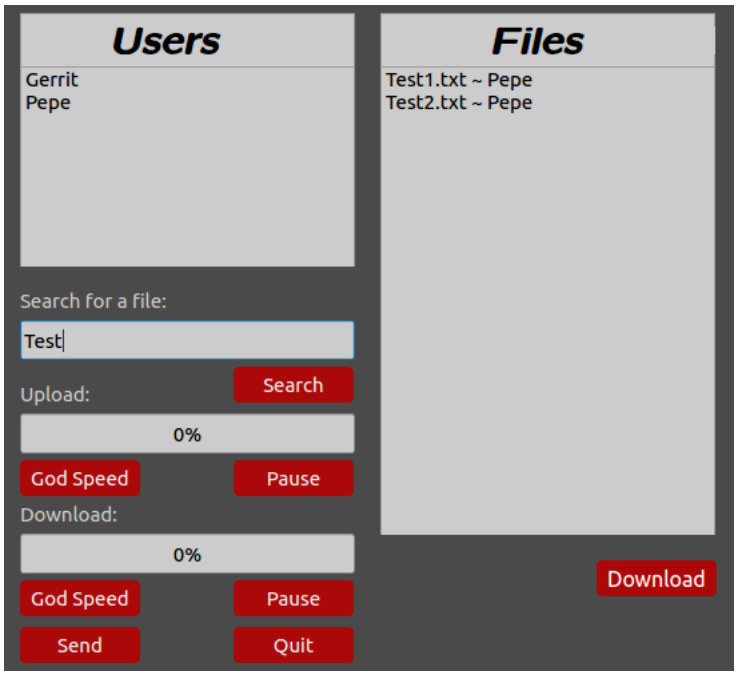

Figure 5.1: Files on the client's system

For the case of searching the exact file name:

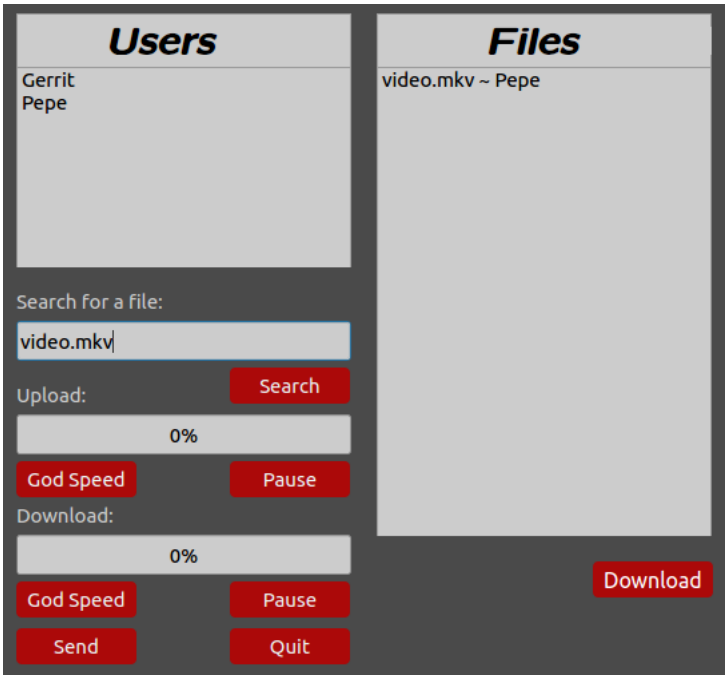

Figure 5.2: Exact file name

For the case of specifying a substring:

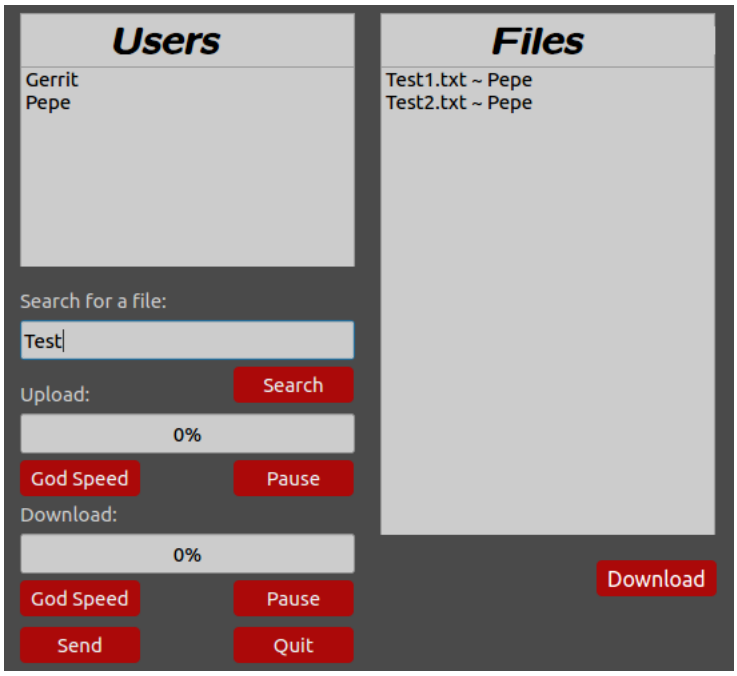

Figure 5.3: Sub string specified

For the case of spelling mistakes:

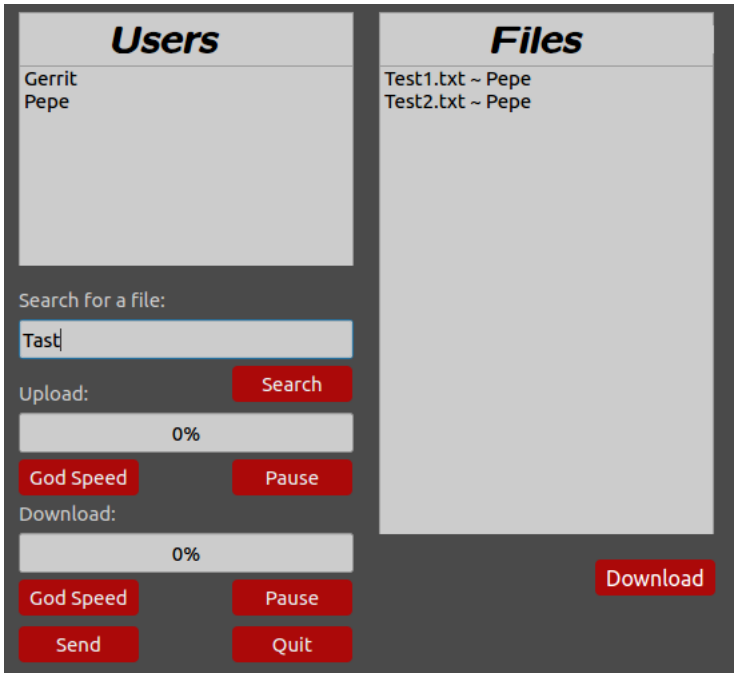

Figure 5.4: Spelling mistakes and the results it yields

For the case of having too many errors:

Figure 5.5: Too many errors were present

5.1.4 Conclusion

From the results, it is easy to see that when exact file names are specified the file is found. It is also clear that when only substrings are present or spelling mistakes were made there are still results, but when the spelling mistakes are too much or the name is to obscure there are no results. Hence the hypothesis can be accepted.

5.2 Single connection stream

5.2.1 Hypothesis

Multiple clients can download at the same time and it will not affect each other.

5.2.2 Method

Since this experiment only entails single connection stream downloading and uploading was run separately, but multiple clients downloaded and uploaded at the same time. Ex. Both user A and user B downloaded at the same time and afterward both A and B uploaded at the same time.

5.2.3 Results

Even when multiple clients downloaded or uploaded at the same time the speed of the download/uploads was not effected and the program remained stable and there were no errors.

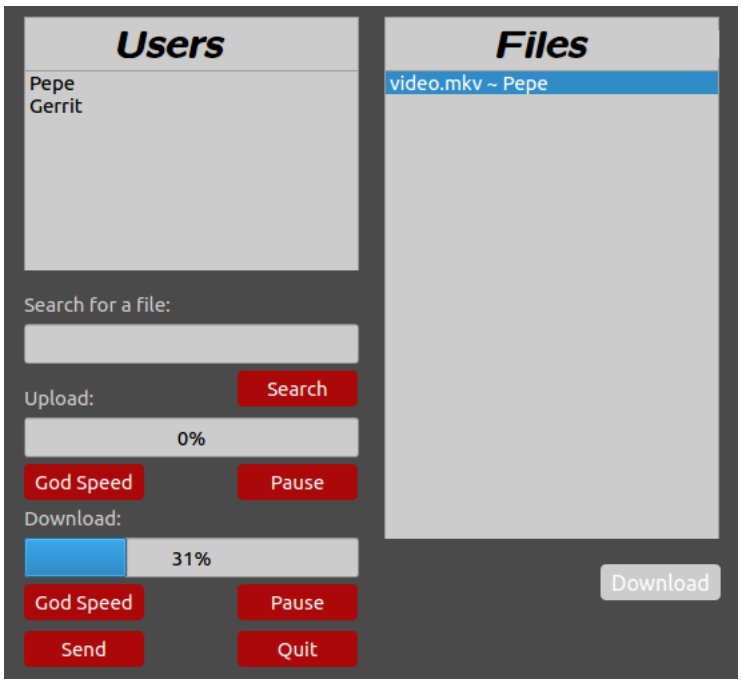

Figure 5.6: User A downloading

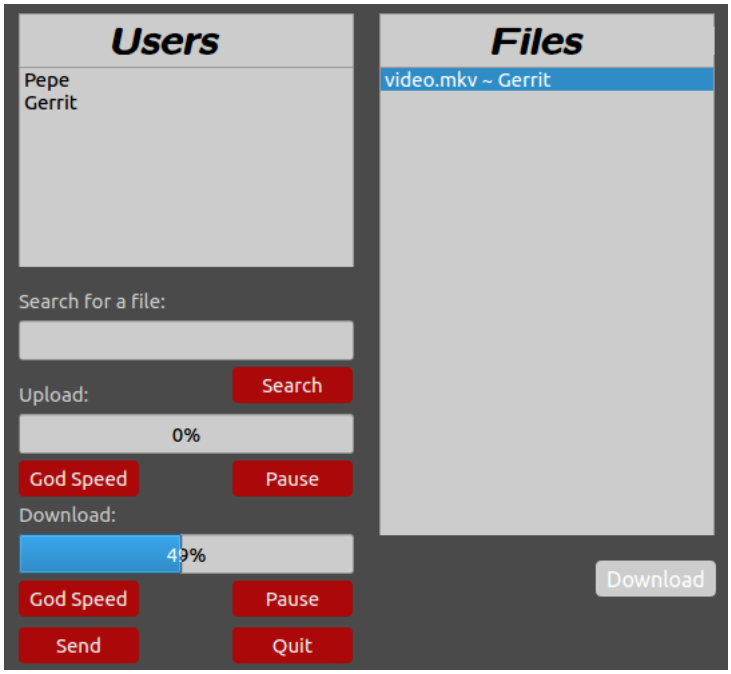

Figure 5.7: User B downloading

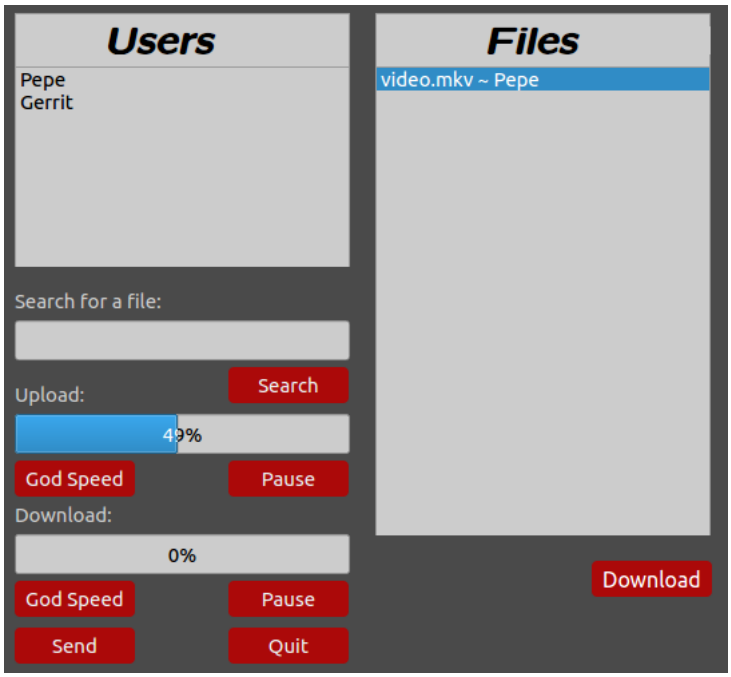

Figure 5.8: User A uploading

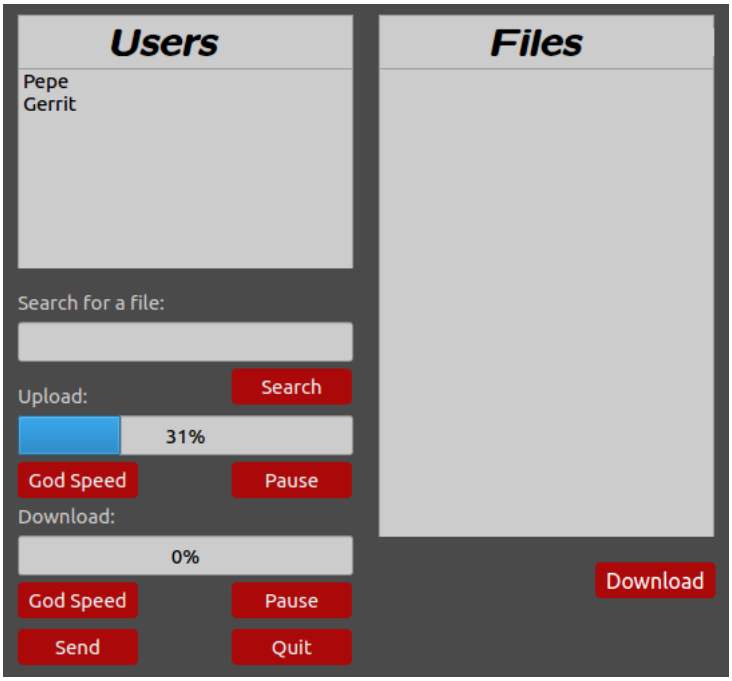

Figure 5.9: User B uploading

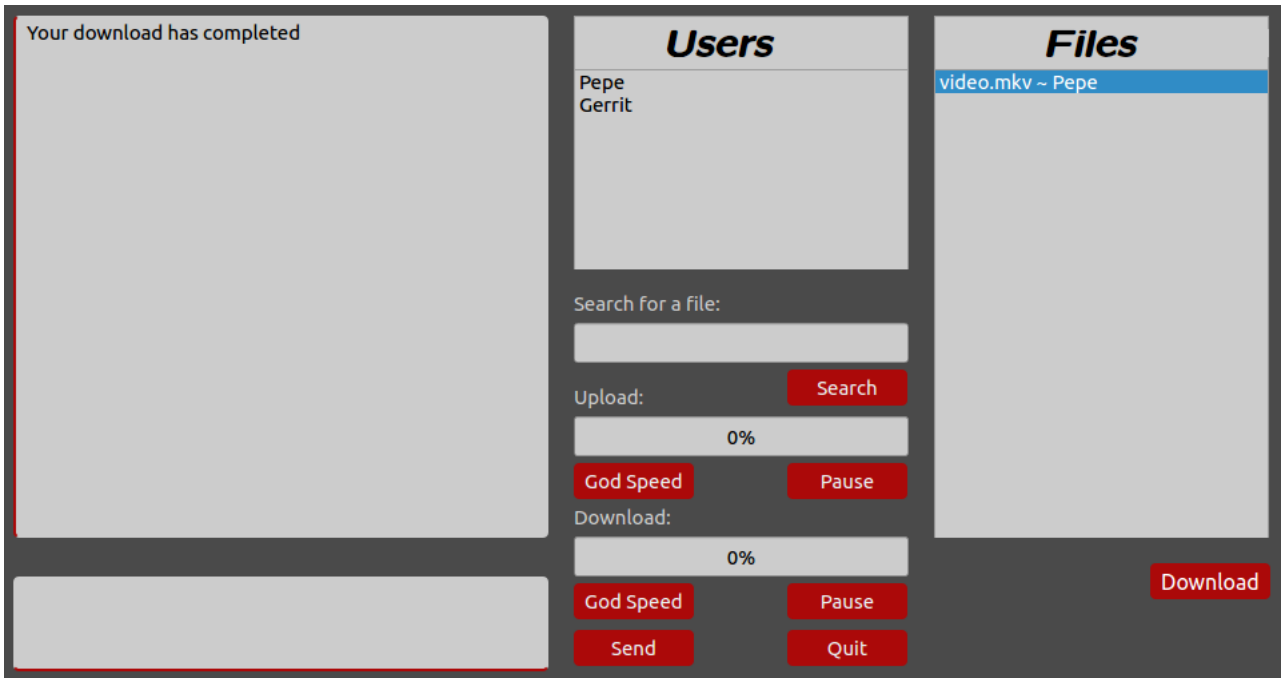

Figure 5.10: Download complete (Gerrit)

Figure 5.11: Upload complete (Pepe)

5.2.4 Conclusion

From the results, it is clear that the hypothesis can be accepted since the program remained stable under the experiment.

5.3 Double connection stream

5.3.1 Hypothesis

Multiple clients can upload and download files concurrently without affecting the speed of the download/upload or the stability of the program

5.3.2 Method

In this experiment, multiple clients both download and uploaded at the same time. Ex. Both user A and B download and upload at the same time.

5.3.3 Results

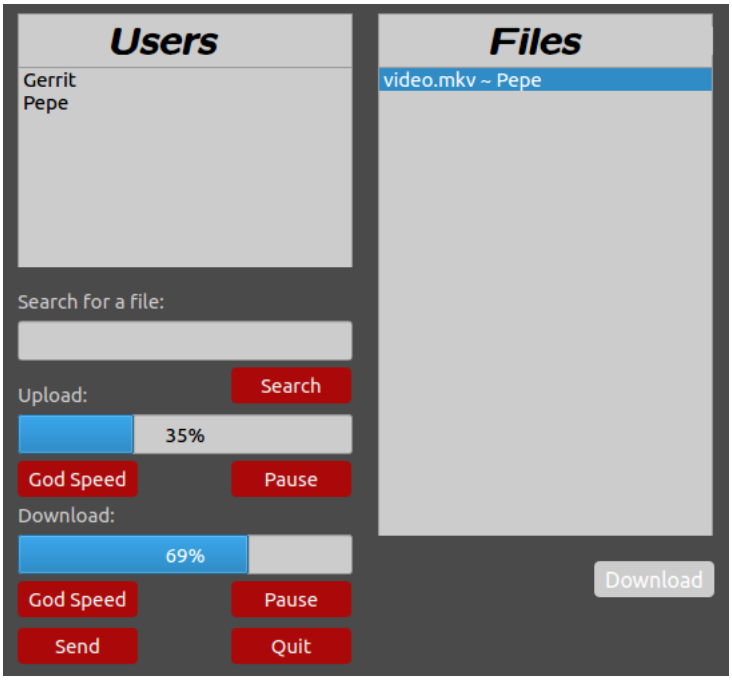

Figure 5.12: User A both download and upload

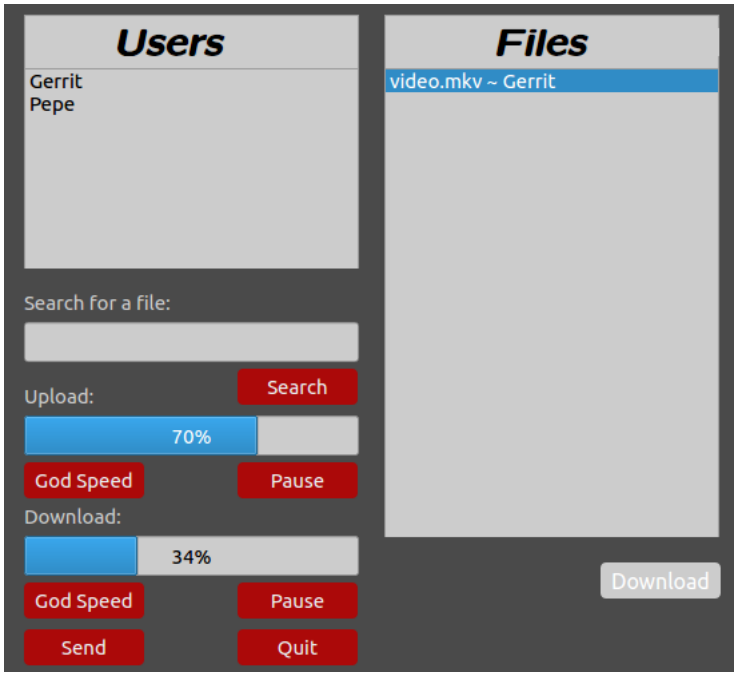

Figure 5.13: User B both download and upload

Figure 5.14: Double connection stream download and upload complete

5.3.4 Conclusion

When multiple users both upload and download at the same time it does not affect the speed of the transfers or the stability of the program. Hence the hypothesis can be accepted.

5.4 Pause and resume functionality

5.4.1 Hypothesis

Consecutively pausing and resuming a download or upload will not affect the stability of the program or corrupt the received file.

5.4.2 Method

For this experiment, the file being downloaded is firstly paused and resumed five times on the downloading side. After doing a sha1sum on the received file and confirming that the file has no corruptions the file is downloaded once more and this time the download is paused and resumed five times on the uploading side and a sha1sum is executed again to see whether the file is corrupted.

5.4.3 Results

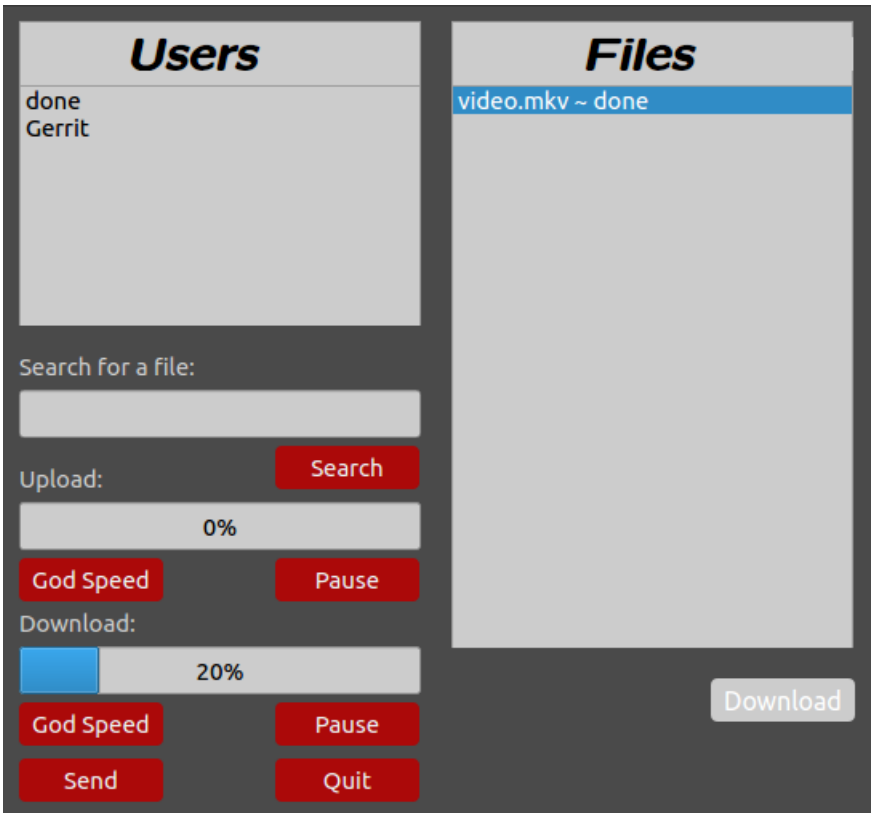

Figure 5.15: The paused file on the downloading side

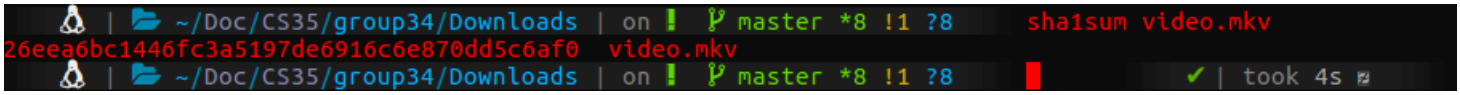

Figure 5.16: sha1sum of the file after pausing consecutively on the downloading side (Gerrit)

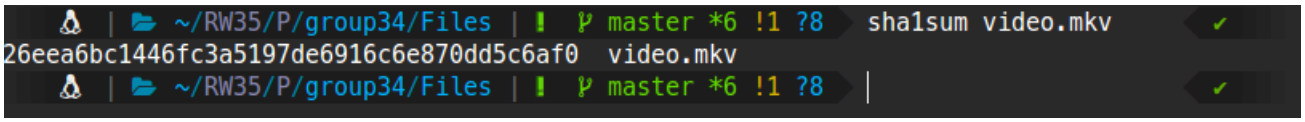

Figure 5.17: sha1sum of the file after pausing consecutively on the downloading side (Jacques)

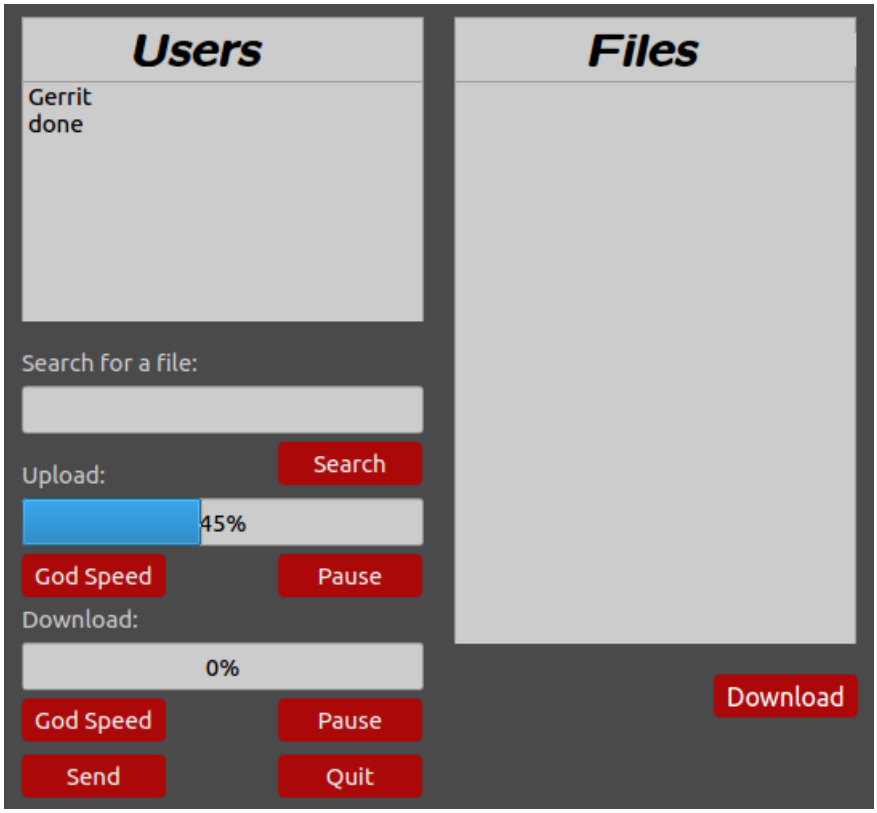

Figure 5.18: The paused file on the uploading side

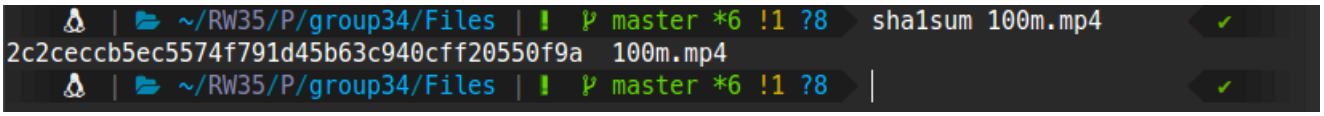

Figure 5.19: sha1sum of the file after pausing consecutively on the uploading side (Jacques)

Figure 5.20: sha1sum of the file after pausing consecutively on the uploading side (Gerrit)

5.4.4 Conclusion

From the results, it is clear that the consecutive pausing and resuming did not affect the stability of the program and the resulting received file is not corrupted. Hence the hypothesis can be accepted.

5.5 The effect of using encrypted sockets on transfer time

5.5.1 Hypothesis

The SSL wrapped sockets will have a slightly longer transfer time for the same file compared to normal sockets.

5.5.2 Method

Four different files with file sizes 1MB, 10MB, 100MB, and 480MB were transferred from one client to another over an SSL encrypted socket and then over a normal socket. Each transfer time of each file was recorded.

5.5.3 Results

Below are the results of the transfer times:

Filesize	Upload time	Download time	Upload time - ssl	Download time - ssl
1MB	0.01302	0.0305	0.0252	0.0412
	0.0240	0.0385	0.0154	0.0328
	0.0179	0.0318	0.0116	0.0386
10MB	0.0884	0.2246	0.1090	0.2673
	0.0878	0.2439	0.1080	0.2446
	0.0943	0.2698	0.1128	0.4119
100MB	0.9594	2.2983	1.1113	2.3102
	0.9735	2.4645	1.133	2.4687
	0.9724	2.3912	1.1001	2.4272
480MB	8.3022	9.3104	8.9040	10.0015
	8.2525	9.7672	9.6877	10.0423
	8.1082	9.2966	10.0952	10.5085

For the normal socket the average upload times are as follows: 1MB-0.0183, 10MB-0.0901, 100MB-0.9718, 480MB-8.2209 and the download times: 1MB-0.0336, 10MB-0.2461, 100MB-2.3847, 480MB-9.4581. For the encrypted socket the average upload times are as follows: 1MB-0.0174, 10MB-0.1099, 100MB-1.1154, 480MB-9.5623 and the download times: 1MB-0.0375, 10MB-0.3079, 100MB-2.4020, 480MB-10.1841.
As seen from the results above the SSL sockets have a slightly longer download and upload time for every file size.

5.5.4 Conclusion

Looking at the results above we can conclude that there is slight overhead when using the SSL sockets. This overhead can comes from encrypting and decrypting from the data received and sent. We can accept our hypothesis.

6 Conclusion

From the experiments performed the following was clear:

- Searching a file with the exact name, substring or spelling mistakes gives the correct results
- Single connection stream does not affect the receiving of the files or the stability of the program
- Double connection stream does not affect the receiving of the files or the stability of the program
- Consecutively pausing and resuming a download or upload will not affect the stability of the program or corrupt the received file
- Secure sockets have a slightly higher download time than normal sockets.