

Computer Science 354
Assignment 5 – File Sharing Program
By Group 16

Project collaborators:

Wihan Uys (21553491) - 21553491@sun.ac.za

Nikita Smal (20720661) - 20720661@sun.ac.za

Contents:

List of Figures:	3
List of Tables:	3
Introduction:.....	4
Features included in the solution:	4
Features not included in the solution:	7
Extra features included:	7
Description of files:.....	7
Description of the program:	8
Issues encountered:	8
Compilation:	8
Execution:	8
Note:.....	9
Libraries:	9
Experiments:	9
Methodology:.....	9
User connection and disconnection.....	10
Username selection	10
The chat functionality	10
Searching for a file.....	10
Downloading different file types	10
Changing the chunk size for sharing files	10
Results:.....	11
User connection and disconnection.....	11
Username selection	11
The chat functionality	12
Searching for a file.....	14
Downloading different file types	14
Changing the chunk size for sharing files	15
Conclusion:	16

List of Figures:

Figure 1: Initialisation of the Server in terminal.....	4
Figure 2: Sign In Popup.	5
Figure 3: File Sharing Popup.	6
Figure 4: Client GUI.	6
Figure 5: File Download Popup.	7
Figure 6: Server Response Results from the User Connection Test.	11
Figure 7: Server Response Results from the Username Selection Test.	11
Figure 8: Messages Received by the Three Connected Clients.	13
Figure 9: Results shown for a Search for "1".....	14
Figure 10: Results shown for a Search of "Test4.txt".....	14
Figure 11: Time Taken and Packet Loss for Different Chunk Sizes.	15

List of Tables:

Table 1: Client File Lists compared to the List of Files Returned from Searches.	14
Table 2: Comparison between Original and Sent Files.....	15
Table 3: Results of Varying Chunk Size of Packets.	15

Introduction:

The purpose of this report was to implement a file sharing program that was able to share files in a peer-to-peer (P2P) manner. This program was coded in python and used the library PySimpleGUI to build the GUI.

This report will explain the features of the implemented file sharing program as well as features that were unable to be implemented. There will also be a description of the files used in the program as well as issues encountered while creating the program. Furthermore, the report will show the experiments conducted to test functionality of the program and ensure the requirements were met. Finally, the report will conclude with the results of the experiment as well as give recommendations based on the findings.

Features included in the solution:

This section described the features of the program and discussed the importance of each.

The server was only there to handle the communication between the clients and the user was unable to interact directly with it. Figure 1, showed the server, which had the following features:

- The server displayed all information, like connections, downloads, searches, and disconnects, in terminal.
- The server only stored the client information and distributed each client's unique username to all clients. No two clients had the same username, and the server prevented any duplicates.
- The server did not track or store the files shared by clients.
- All communication between clients was handled by the server, but file transfer occurs through P2P.
- Clients first connected to the server before they could talk to each other.



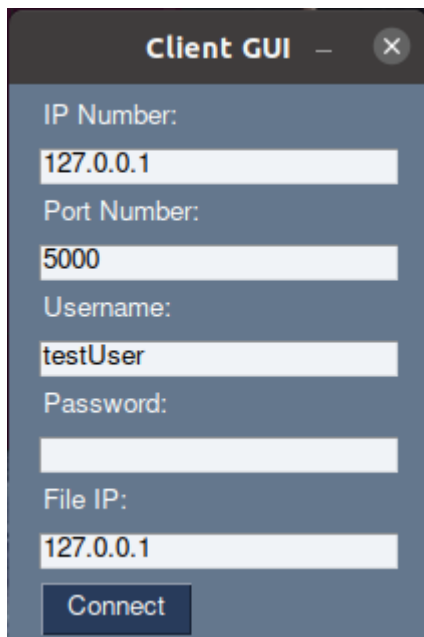
```
nikitasmal@nikitasmal-PS42-Modern-8RC:~/group-16-project-5$ python3 server.py
[server] run
[server] current settings:
ip:          127.0.0.1
tcp port:    5000
name:        A Python Server
password:
max clients: 32
header size: 64
encoding:    utf-8
Accept settings? [y]Yes [n]No
y
[server] binding on 127.0.0.1:5000
[server] starting...
[server] waiting for connections...
[server] --uptime=0 seconds--
```

Figure 1: Initialisation of the Server in terminal.

The user interacted with the program through the client user interface. The clients that connected to the server had the following features:

1. A client was able to chat to all other clients through a broadcast or to a single client through a whisper.

2. The clients could see a list of all online users.
3. Each client chose the settings of the server as well as their unique username at the start of the program as shown in Figure 2.
4. Multiple clients could connect and disconnect without interrupting the running of the program.
5. When connecting to the server, clients chose which files they wanted to share by specifying the folder location of the files in Figure 3.
6. Once connected to the server, the clients were able to search for a file listed by other clients by any letter in the filename as shown in Figure 4. The server handled the search by requesting the results from every other client.
7. If the file was found, the user could select to download the file from Figure 5 by giving the port number and the filename.
8. The client was able to pause and resume the download at any time.



The image shows a 'Client GUI' window with a dark blue header and a light blue body. It contains several input fields for user configuration. The fields are labeled 'IP Number:', 'Port Number:', 'Username:', 'Password:', and 'File IP:'. The 'IP Number' and 'File IP' fields are pre-filled with '127.0.0.1'. The 'Port Number' field is pre-filled with '5000'. The 'Username' field is pre-filled with 'testUser'. The 'Password' field is empty. At the bottom of the form is a dark blue button labeled 'Connect'.

Field	Value
IP Number:	127.0.0.1
Port Number:	5000
Username:	testUser
Password:	
File IP:	127.0.0.1

Connect

Figure 2: Sign In Popup.

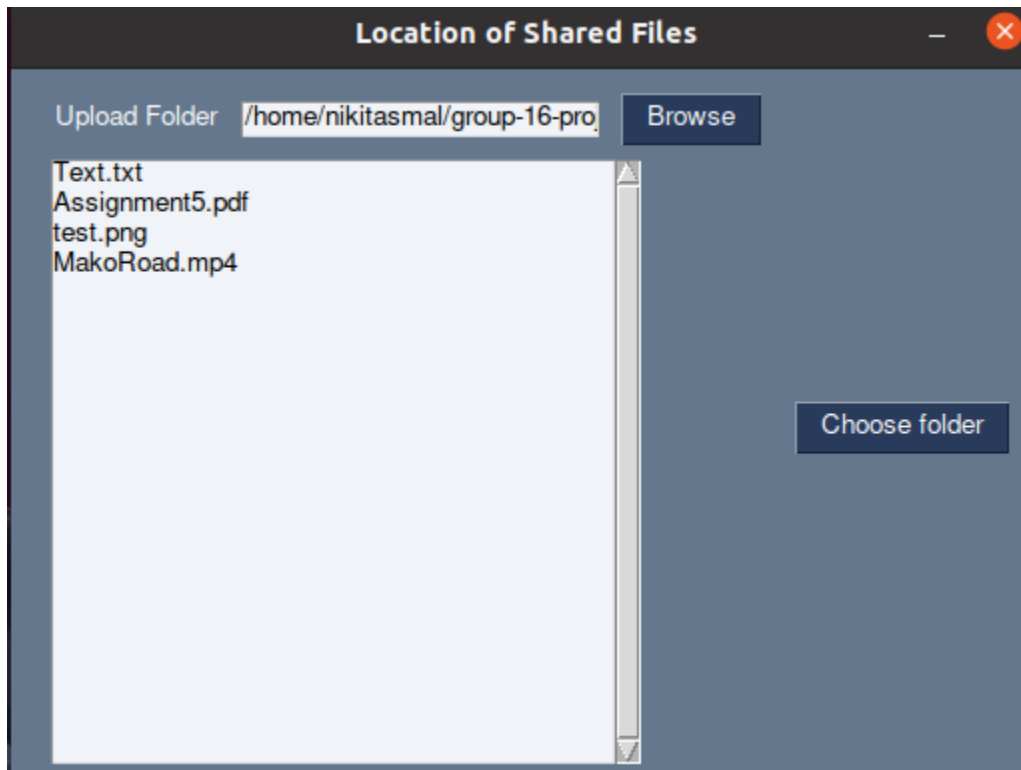


Figure 3: File Sharing Popup.

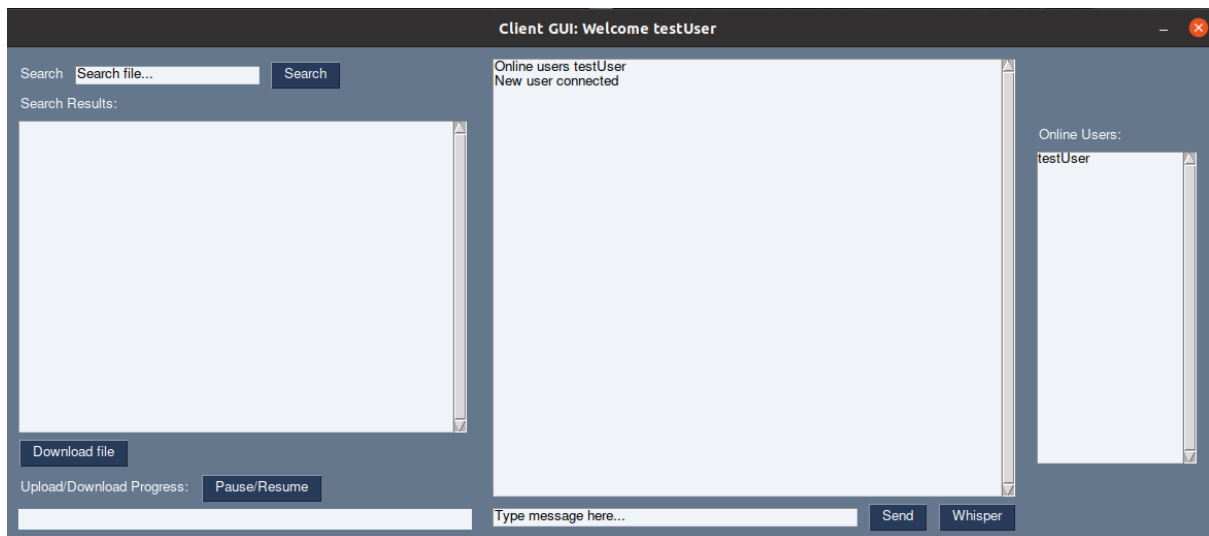


Figure 4: Client GUI.



Figure 5: File Download Popup.

Features not included in the solution:

All features listed in the specification document of the assignment were included.

Extra features included:

As an extra feature the program gave the clients the ability to chat with each other as well as the application was implemented on a GUI.

Description of files:

The root folder included the following files:

1. server.py
 - a. Contained the code to run the Server.
 - b. The server stored the list of clients as well as the connection to each.
 - c. All clients connected to the server, and it handled each connection and communication.
2. client_GUI.py
 - a. Contained the code to run the client GUI as well as the relevant popups.
 - b. Each client started with the sign in popup, then the upload folder popup and then after filling those in, the main GUI was shown.
 - c. The client GUI handled both the sending of messages to other clients through the server as well as the displaying of incoming messages from other clients.
 - d. When a user wanted to download a file, the client GUI ran the download popup.
 - e. It also displayed the progress of a download.
3. sign_in_popup.py
 - a. Handled the fields required to sign onto the server.
 - b. Returned a client dictionary with the required values.
4. upload_folder_popup.py
 - a. Contained the user interface to select a folder on the computer to be shared.
 - b. Returned the file path of the folder containing the shared files.
5. download_popup.py
 - a. The download popup handled the input from the user to be able to download a file.
 - b. Returned a dictionary of the port and filename to the client GUI.

Description of the program:

With the detailed descriptions of each file and feature mentioned above, the running of the program is as follows.

First the server was started up and requested the user to confirm or input the settings of the server. Once completed, the server set up the sockets and binds to the given settings. The server started the program timer and began to listen for a new client.

At this point the client GUI started by requesting the user details and the settings of the server to connect too. The client GUI also requested the folder location of the files to share. Thereafter, the client sent a request to the server to connect and waited for confirmation.

The server received the client request and set up this new client on a new thread. The new client thread checked if the client was allowed to join. If the client was allowed, then it stored the client on the server as well as created another thread for the client listener. Otherwise, the client was rejected, and the server continued.

If the client was allowed to connect, it received a response from the server and set up the server listener to receive any further communication from the server on a new thread. Thereafter, the client GUI was created, and the user could begin to use the features described above.

Each command of the program was sent using a header followed by a message ([HEADER] [MESSAGE]) structure. The header would be identified using a '!'. For example, '!download' was used when a client requested a download.

Issues encountered:

This section described the issues that were encountered and how they were overcome:

1. When a user pressed a button like the pause resume, a command was needed to be sent to a different thread to stop the packets from being sent. This had to be done using the thread event method provided by Threading. This event was set to be true or false, therefore when the user selected the button the event would be true to pause and this would be shared across threads thereby, using a conditional loop, pausing the download or upload.

Compilation:

No compilation was required before execution.

Execution:

To execute the code in terminal, the following commands were used:

- `$ git clone git@git.cs.sun.ac.za:Computer-Science/rw354/2021/project-5/group-16-project-5.git`
- `$ cd group-16-project-5`

To run the Server, the following command was run from the root folder:

- `$ python3 server.py`

This started the Server in the Command Line Interface (CLI) which was used to interact with the program.

To run the client, the following command was run from the root folder in a new terminal:

- `$ python3 client_GUI.py`

This started the client GUI which was used to interact with the program.

Note:

Both the Server and the client GUI start with the default IP address 127.0.0.1, and the default port number 5000. These could be changed to any IP address or port number that the user wished to use.

Libraries:

There were many libraries used to complete this program and add functionality to python. The important libraries used for this program were:

1. The socket library was used to create TCP sockets and send packets.
2. Threading was used to split the running of the program onto multiple threads so that separate processes could run in parallel.
3. Dataclasses provides a decorator and functions for automatically adding generated special methods such as `__init__()`.
4. To make the terminal easier to navigate, Colorama was used to change the text colour of certain words.
5. PySimpleGUI was used to create the GUI.

Experiments:

The purpose of the experiments was to validate whether the features described above were working correctly. All experiments contained two or three clients connected and the program was hosted over Hamachi. The following features were tested:

- User connection and disconnection,
- Username selection,
- The chat functionality,
- Searching for a file,
- Downloading different file types, and
- Changing the chunk sizes for sharing files.

These features were chosen because they were the fundamental features of the program. Without these features the program would not be able to meet the needs of the user.

Methodology:

For each of the features above, the program was set up in the following way.

1. Start the server and confirm the settings

User connection and disconnection

2. Connect two clients
3. Disconnect one
4. Reconnect another with the same username

The purpose of this was to ensure that client connections and disconnections were handled correctly and do not cause the server or other clients to have errors.

Username selection

2. Connect one client with username “default”
3. Connect another with the same username
4. Connect again with a unique username

The purpose of this test was to verify that the server only accepted unique usernames, and the connection of other clients was not interrupted.

The chat functionality

2. Connect three clients
3. Send a message a message to all clients
4. Send a message to a specific online user

This ensured that when a normal message was sent, all clients received it as well as when a whisper was sent, only the receiver could see the message.

Searching for a file

2. Connect three clients
3. Select folders for all clients that contained many files that are different for each client.
4. Search for a letter that many filenames contain
5. Search for a complete filename
6. Search for a file type

This test was used to determine whether the user was able to search for partial matches as well as whether the results would be the combination of all available files from multiple users.

Downloading different file types

2. Connect two clients
3. Select different file types; png, jpeg, txt and pdf
4. Download each file
5. Open the original and the sent files to compare for any byte loss

The program may have been better at sending one file type over another therefore it was important to test many different types. For example, the text file that was sent may be missing letters of words and the sent png may have missing pixels, but each would have had a varying degree of packet loss depending on their size and type. The purpose of this experiment was to determine which file types were sent the most effectively.

Changing the chunk size for sharing files

1. Change the chunk size value that clients use to share files
 - a. 8192 bytes

- b. 4096 bytes
 - c. 1024 bytes
 - d. 128 bytes
2. A 50 MB test file was created using `$ xfs_mkfile 50m TestFile`
 3. Connect two clients with different usernames
 4. Start a file transfer by requesting a download of the same 50MB file.
 5. Compare packet loss vs. time taken for the download

Results:

The following were the results of the experiments conducted.

User connection and disconnection

Two users were successfully connected and when one disconnected, the other remained unaffected. Furthermore, the client was able to reconnect successfully without any errors. Figure 6 shows that multiple clients can connect and disconnect from the server without any issues.

```
[server] --uptime=270 seconds--
[server] --uptime=315 seconds--
[server] --uptime=360 seconds--
[server] new client [127.0.0.1] connected, setting up thread
[server] new thread created successfully
[server] --> [127.0.0.1] Welcome to A Python Server!
[server] creating thread for listening to [user1]
[server] new client [127.0.0.1] connected, setting up thread
[server] new thread created successfully
[server] --> [127.0.0.1] Welcome to A Python Server!
[server] creating thread for listening to [user2]
[server] purging client [user2]
broadcast new clients
[server] --uptime=405 seconds--
[server] new client [127.0.0.1] connected, setting up thread
[server] new thread created successfully
[server] --> [127.0.0.1] Welcome to A Python Server!
[server] creating thread for listening to [user2]
```

Figure 6: Server Response Results from the User Connection Test.

Username selection

The first client “user1” was able to connect as normal, but when the second client tried to connect to the server, the server gave the output shown in Figure 7. Thereafter, the client was able to connect with a unique username.

```
[server] --uptime=405 seconds--
[server] new client [127.0.0.1] connected, setting up thread
[server] new thread created successfully
[server] --> [127.0.0.1] Welcome to A Python Server!
[server] creating thread for listening to [user2]
[server] --uptime=450 seconds--
[server] purging client [user2]
broadcast new clients
[server] new client [127.0.0.1] connected, setting up thread
[server] new thread created successfully
[server] --> [127.0.0.1] username already in use
```

Figure 7: Server Response Results from the Username Selection Test.

The chat functionality

All three clients 1, 2, and 3 were connected when client 1 sent a message to everyone. This message was received and was the same for each client. Thereafter, client 1 sent a message to client 2 and client 3 was the only client that did not see the message. The results shown in Figure 8 show that the messaging and whisper work correctly.

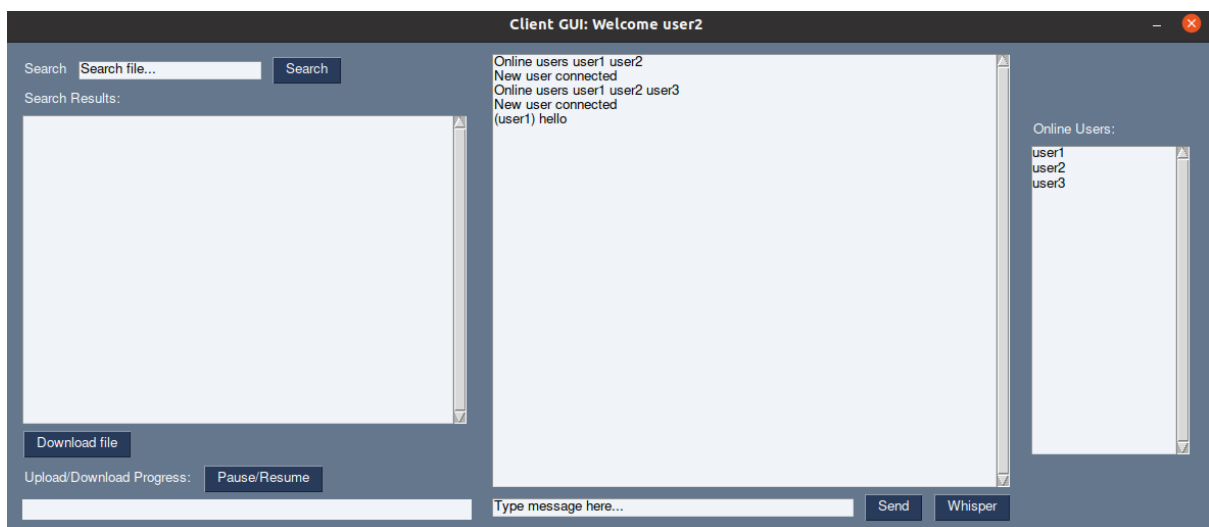
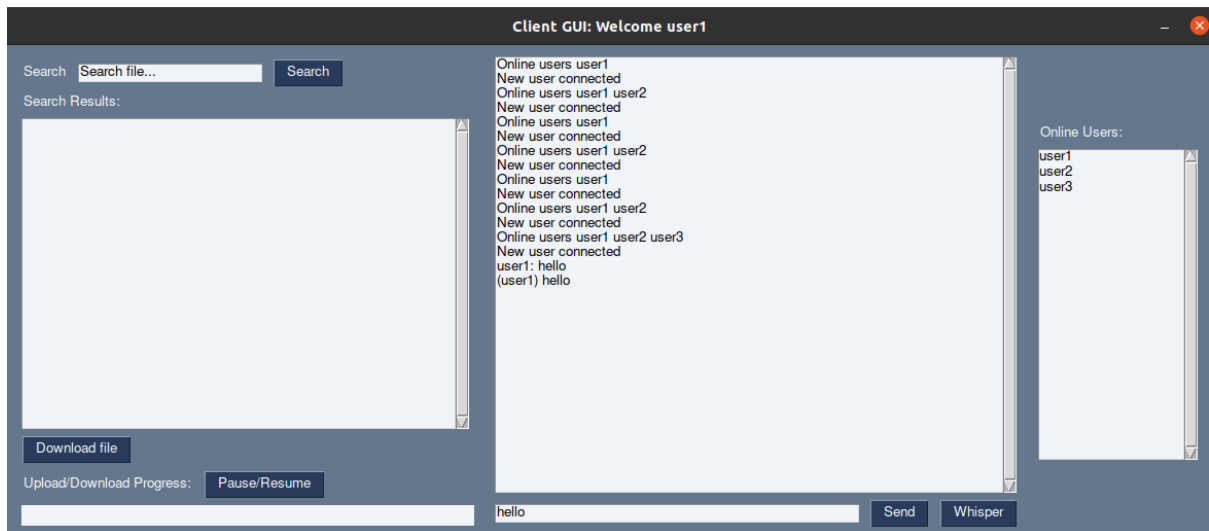


Figure 8: Messages Received by the Three Connected Clients.

Searching for a file

With the three clients 1, 2, 3 connected, two searches were made. The first was for a single character “1” and the other was for a complete file name “Test4.txt”, which are shown in Figure 9 and Figure 10, respectively. The files shared by each client and the results of each search are shown in Table 1.

Table 1: Client File Lists compared to the List of Files Returned from Searches.

Client A Files	Client B Files	Client C Files	Result For “1”	Result For “Test4.txt”
Test1.txt	Test11.txt	Test111.txt	Test1.txt	Test4.txt
Test2.txt	Test4.txt	Test6.txt	Test11.txt	
Test3.txt	Test5.txt	Test7.txt		



Figure 9: Results shown for a Search for "1".

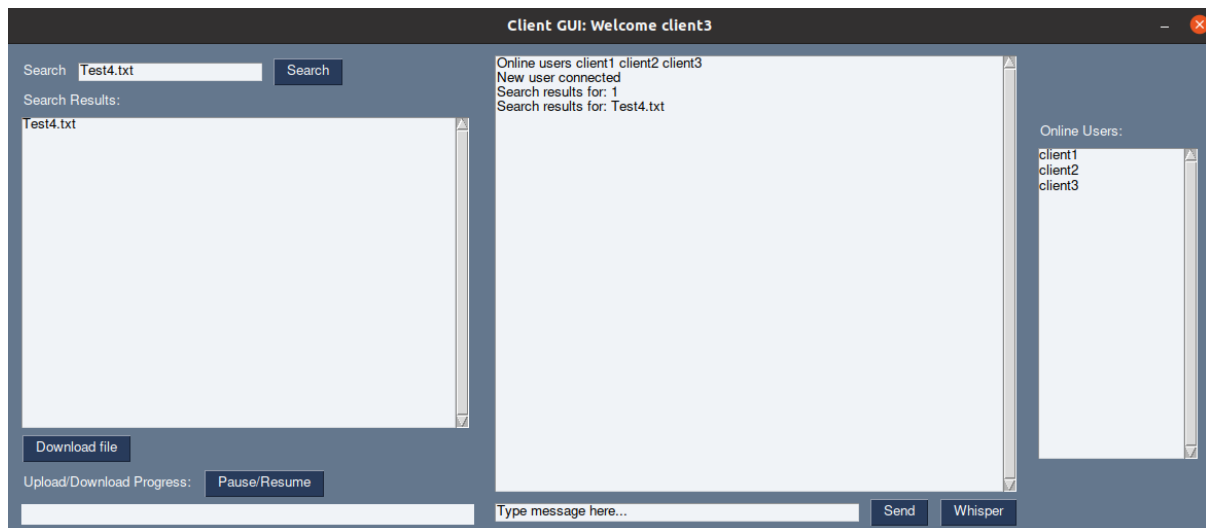


Figure 10: Results shown for a Search of "Test4.txt".

Downloading different file types

Table 2 shows how certain files experience a different percentage of bytes loss. PDF experienced the most bytes lost even though it wasn't the largest file. The png, which was the largest file, experienced the least bytes lost. This is likely due to the different file formats storing their bytes in different ways

making some easier to convert between bytes and their respective formats. It is clear that all files cannot be treated exactly the same when converting them to bytes.

Table 2: Comparison between Original and Sent Files.

File Types Sent	Size Before Sending	Size After Sending	% Bytes Lost
png	1453482	1450618	0.20%
jpeg	29145	28633	1.76%
txt	4675	4619	1.20%
pdf	119842	116554	2.74%

Changing the chunk size for sharing files

Table 3: Results of Varying Chunk Size of Packets.

Chunk Size	Packet loss %	Time taken (seconds)
8192 bytes	1.6	62
4096 bytes	0.53	124
1024 bytes	0.09	653
512 bytes	0.00	856
128 bytes	0.00	1486

From the results of this experiment shown in Table 3, it was concluded that the most efficient chunk size for sharing files was 512 bytes, depending on the sending and receiving computer hardware. Figure 11 shows how the packet loss and speed increase with chunk size.

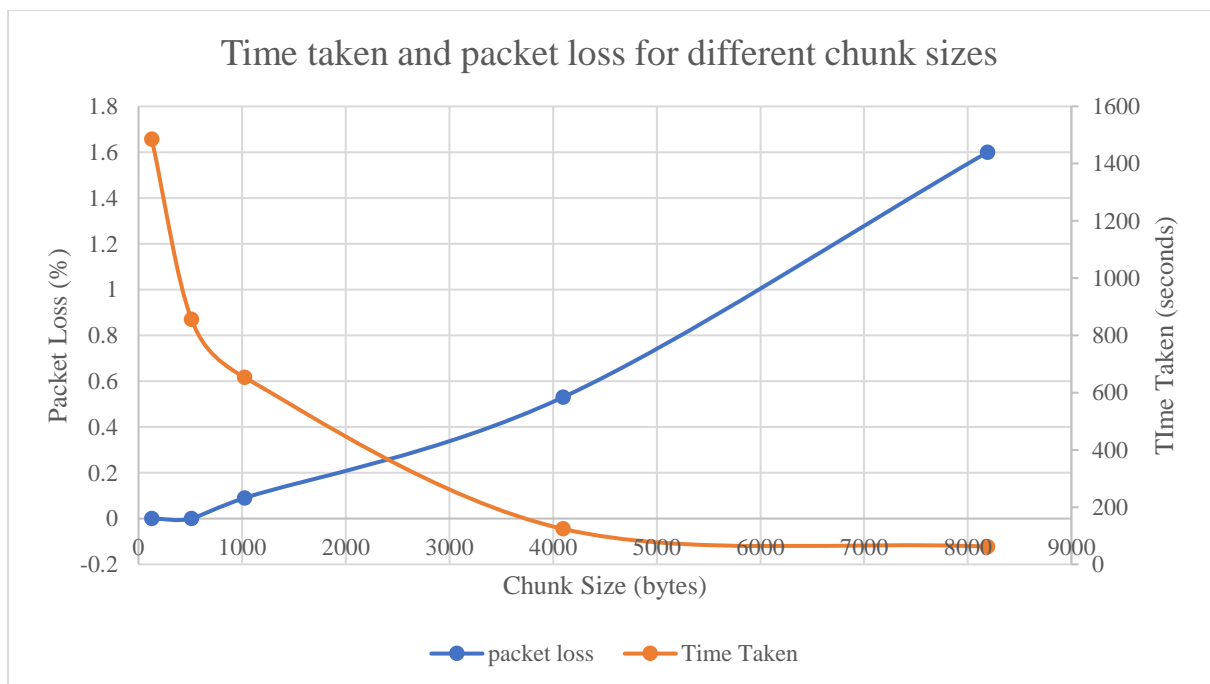


Figure 11: Time Taken and Packet Loss for Different Chunk Sizes.

Conclusion:

The purpose of this report was to create a file sharing program. This report showed how this was done by explaining the features of the server and client as well as features that were unable to be implemented. There was also a description of the files used in the program as well as issues encountered while creating the program.

Furthermore, the report used experiments to test the functionality of the program and ensure the requirements were met. These tests showed how multiple clients were able to connect to the server and send messages to each other. Clients were able to search for files on other clients' computers without knowing the exact filename as well as download them. The tested download formats were txt, jpeg, png and PDF which were all successful. It was found that the number of bytes lost was not due to the file size but rather the file format. This means that different file formats should be treated differently.

In order to correct the errors in byte loss and optimise the program, the chunk size was varied. It was found the larger chunk size had a faster data transfer rate, but the packet loss was more noticeable. The smaller packet size of 512 bytes was found to have the best performance with no packet loss.