CS2101 – File Sharing Application

170005748

Introduction:

The task was to develop a file sharing application which would allow a user to send and receive files with other users of the application.

Instructions:

The application can be run by navigating to the directory where the jar is and running the command:
java -jar LitTorrent.jar <shared_directory> <download_directory>
This application should be able to interact with other applications implementing the LitTorrent protocol.
The UML diagram is also included in the zip file.
The protocol employs a few basic commands such as send, delete and list files. The help command within the program specifies more information on the commands as well as the included RFC design document as detailed below.

Design:

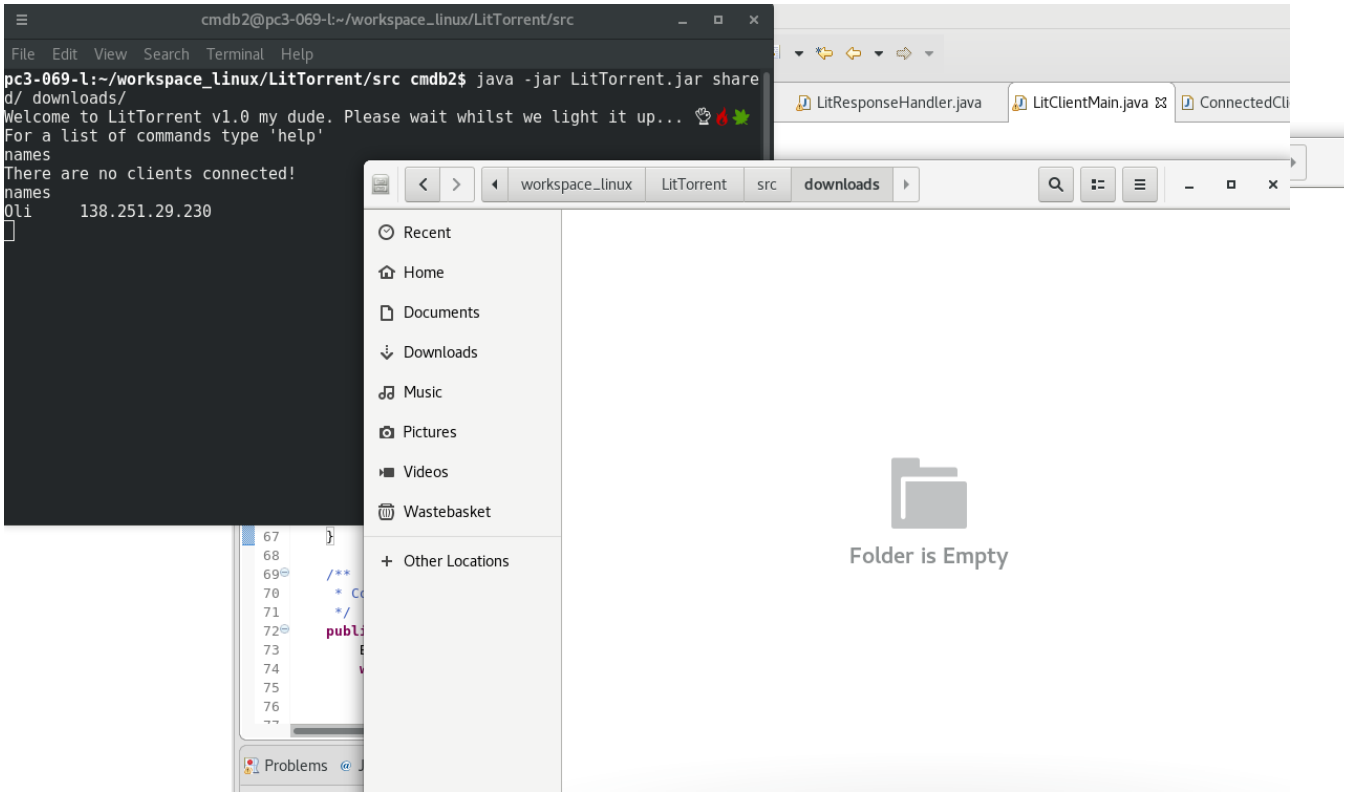
In order to share between other clients, a common protocol was required. A number of us in the class decided to design our own protocol and use this in order to demonstrate interconnectivity between clients. This collaborative design document is included with the zip where greater detail can be found on the design choices made as well as the commands and valid responses. I also decided that the application should not recognise itself – the IP of a connecting client must be different to the client IP.

Implementation:

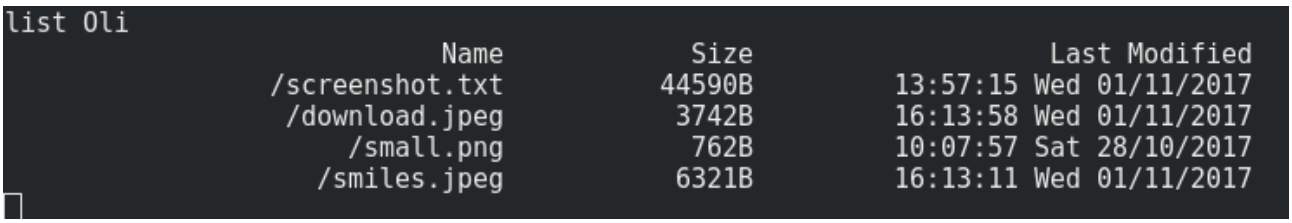
The application makes extensive use of threads to discover new clients connecting on the multicast address, disconnect clients and to handle more than one incoming response at a time. In order to send files of any type, a BufferedReader could not be used and so the header must be processed by a normal input stream. This forced me to write a read line function specifically to read lines of text from an input stream.
To keep a list of connected clients, I have a class which holds the necessary information and this is populated or de-populated as required by the multicast server. For example, it contains the last time the server received a request from the IP address so that it can be disconnected if it hasn't received a request within 10 seconds.
Since the shared and download directories can be user-specified, there is a check that the directory path ends with a forward slash, as this would otherwise break some functionality. The incoming request handler also checks that the incoming requested file path begins with the shared directory, otherwise it appends to ensure that files are not requested, or deleted, from outside of the specified directory.
I made sure that the implementation adhered to the defined protocol as closely as possible so as to ensure connectivity between different clients using the same protocol.
One function that was required was to continuously look into directories to list their files when calling the list command so that the file path was correct when sending it to other users.
On exiting the program, I do attempt to close threads and connections cleanly, however this can take very long, so I had to implement a quick fix where if the while loop runs too many times then the program just calls System.exit(). I don't feel this was the best course of action, but it stops the program from hanging when trying to exit.

Testing

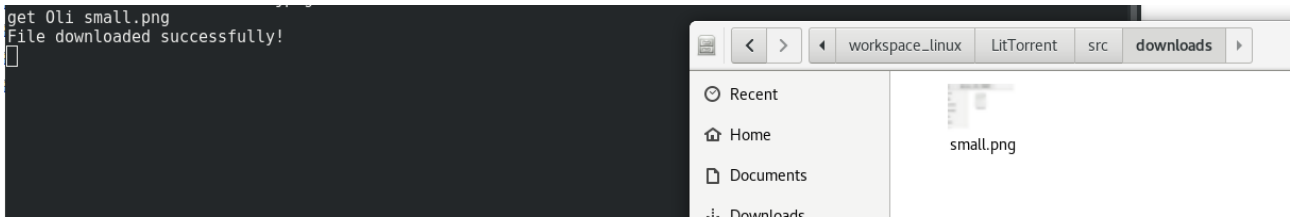
For some testing, I collaborated with a fellow student to test whether my program could send requests properly. These tests are shown below:



This test shows that there are currently no files in the download directory, and that the program displays the list of connected clients correctly.



Here I am testing that the requested list of files is also displayed correctly.



And this shows that the file was received correctly

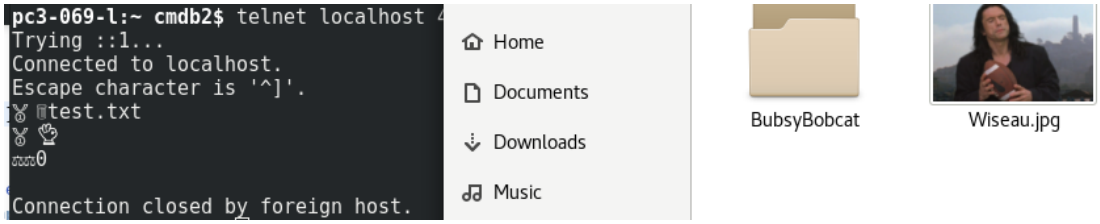
For testing my own server, I used telnet to get image proof and also validation from classmates.

```
pc3-069-l:~ cmdb2$ telnet localhost 42069
Trying ::1...
Connected to localhost.
Escape character is '^]'.
🤘 ✋
🤘 ✊
5

BubsyConnection closed by foreign host.
```

Testing that the identifier “Bubsy” is returned correctly.

```
pc3-069-l:~ cmdb2$ telnet localhost 42069
Trying ::1...
Connected to localhost.
Escape character is '^]'.
🤘 👌
🤘 ✊
124

shared/BubsyBobcat/purrfect.txt471509539460000
shared/Wiseau.jpg38999915071088640000
shared/test.txt221509733506000
Connection closed by foreign host.
```

Test that the list of files is returned correctly.



Test that files are deleted correctly.

I also tested my error checking:

```
names
There are no clients connected!
list NotRealClient
Not a valid name! Type 'names' for a list of names
Unable to resolve host! Make sure the name is spelt correctly, otherwise they may have disconnected!
```

Conclusion:

Overall, I think I achieved at least the base requirements as well as a working timeout for connected clients. Implementing the protocol did bring up a few difficulties with emojis but I think I overcame these successfully. These include the encoding of emojis with some being more than one character and being 4 bytes in length. There was also a last minute issue with spaces in identifier names which caused issues, however, I implemented a quick fix to replace spaces with underscores locally. Given more time, I would have implemented dealing with spaces properly. I also had to return “dummy” when trying to lookup and IP from the friendly name since null or empty would return the localhost address, and list my own files rather than just throwing an error and helpful message. I would also have liked to implement logging for errors which would not be beneficial for the user to see.