Mthokozeleni Sithole STHMTH002 (Individual Work) Report

Description of functionality and features

App Architecture

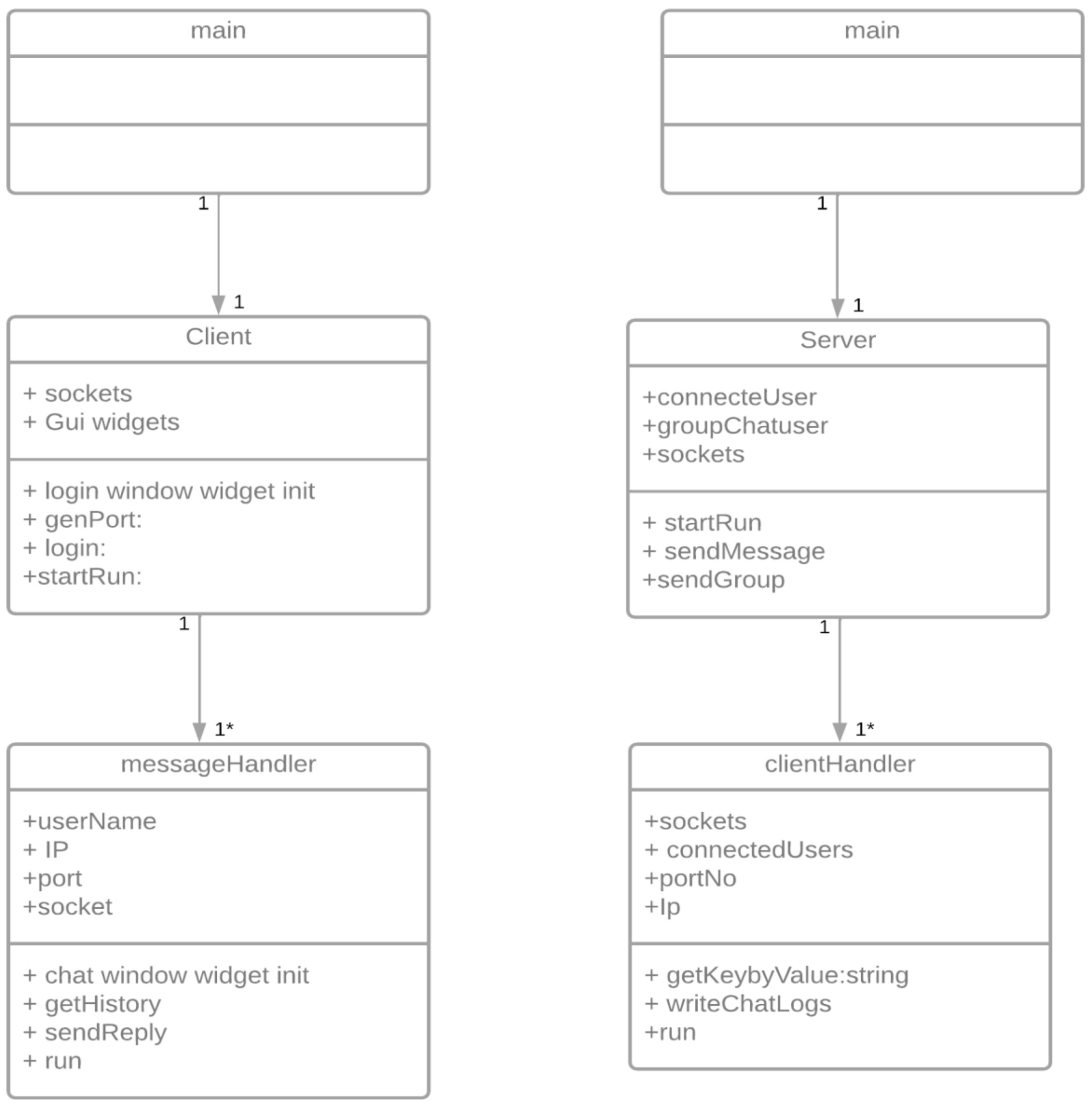
The app consist of 4 classes split into 2 python script files. There is a client class `class Client:` which launches and controls the login GUI and initiates the sockets responsible for sending and receiving messages. The client class has 4 function including the constructor, the methods are genPort `def genPort(self):`, login `def login(self, name, recip):` and startRun `def startRun(self):`. genPort is used to generate a random port number to be used by the client. The login method is called when the “LOGIN” or “GROUP” buttons are pressed, it collects the username and the recipient’s name from the entry widgets and sends it to the server. And finally startRun creates an instance of sockets are used to communicate with the server, creates the login GUI and invokes message handler threads which are used during the chat session to send and receive messages and listen for GUI events.

The messageHandler class `class messageHandler(threading.Thread):` inherits from the threading class and overrides the run method. The run method constantly listens to receive messages the from the server sockets and display them on the text field whilst listening for events on the GUI. There are getters and setters method used to communicate with other methods and the parent Client class, other significant methods are the display method `def display(self,Msg):` used to display messages on the text widget once they are received from the server. The getHistory `def getHistory(self):` method is used to retrieve the users chat history including messages sent while user is offline. The chat history is retrieved by reading a text file created by the server to keep chat logs. Lastly there is the sendRply method `def sendRply(self ):` used to fetch text from entry widget and send to the server.

On the server side, apart from a couple of getters there is the startRun `def startRun(self):`, sendMsg `def sendMsg(self, Msg, sender,recip):` and groupMsg `def groupMsg(self, Msg, sender):` on the Server class `class Server:`. startRun Initiates sockets and bind then listens for login details then updates local variables. After that the method attempts to read from chat logs file if it already exist if there are any recent messages to be sent to the client that has just logged in. client handler threads are then spawned which handle receiving messages from clients and sending them to appropriate recipients. Client handler threads are created according to the type of recipient the user select, then command messages are to notify the users of any changes. Below are the sendMsg and groupMsg used to send to messages to private chat and group chats respectfully. The message is first concatenated meta data required then sent.

The clientHandler class `class clientHandler(threading.Thread):` also inherits from the threading class. Apart from getters from setters the class has the writeChatLogs method `def writeChatLogs(self, Msg):` which appends chat logs or create a new file if it does not exist then it write chat logs. The run method continuously receives and sends messages to the stipulated recipient in the meta data and write or update chat logs. All the features of the App, login window/GUI, chatting window/GUI ,private and group chat and retrieving chat history are implemented through the above classes and methods. All libraries used are built in python libraries including Tkinter which is used to create the graphical user interfaces-GUI.

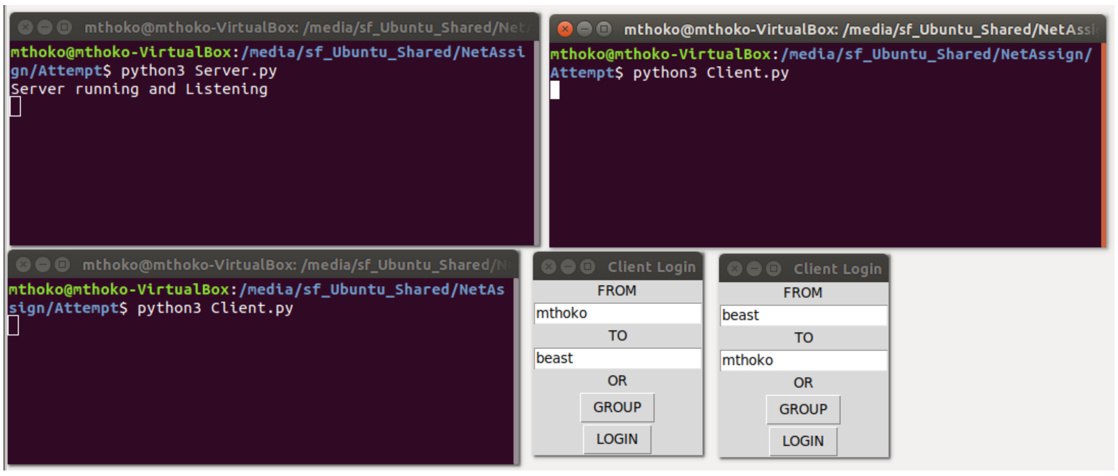
Class analysis diagram



Features

Once the scripts are running the client receive a login screen where they can login at different times if one user logs in and sends messages to an offline user they are stored in the history and will be retrieved by the recipient when they login. To login clients/user enter details, click login then close the login window. To login into a group clients enter their user name then click on the group button then close the login window. Clients in a private chat are separated from group chat activity and cannot send or receive group messages. The get history function ensure reliability by writing chat logs that can be accessed if messages are not sent. While chat once the user press send the entry is cleared to improve the user experience.

Private Chat login



```
mthoko@mthoko-VirtualBox:/media/sf_Ubuntu_Shared/NetAssi
gn/Attempt$ python3 Server.py
Server running and Listening
```

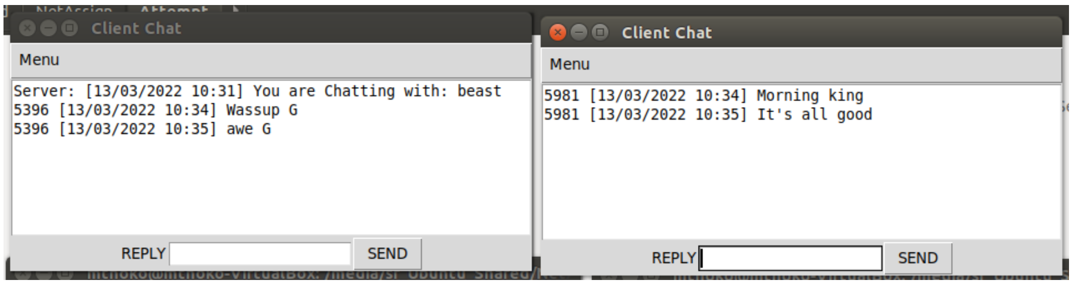
```
mthoko@mthoko-VirtualBox:/media/sf_Ubuntu_Shared/NetAssign/
Attempt$ python3 Client.py
```

```
mthoko@mthoko-VirtualBox:/media/sf_Ubuntu_Shared/NetAs
sign/Attempt$ python3 Client.py
```

Client Login
FROM: mthoko
TO: beast

Client Login
FROM: beast
TO: mthoko

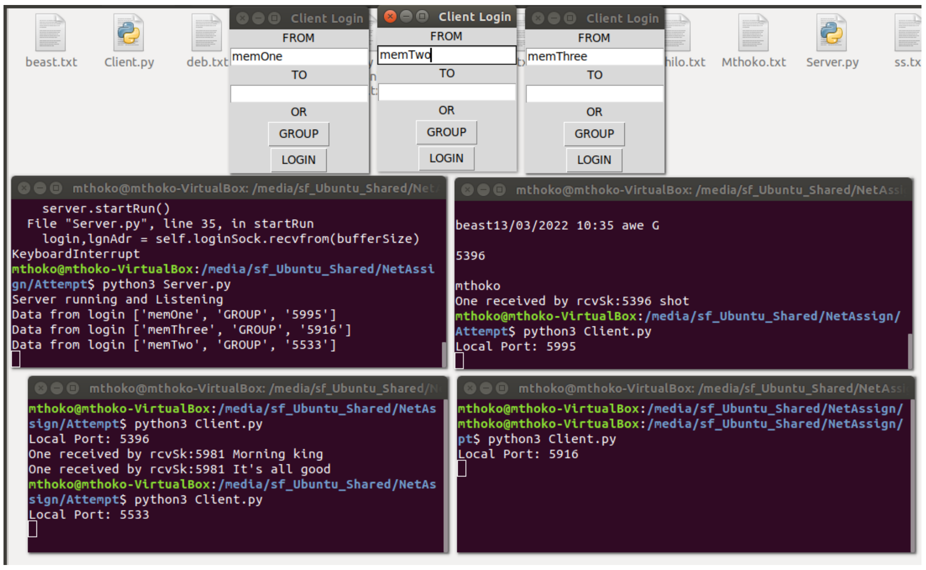
Chat windows



```
Server: [13/03/2022 10:31] You are Chatting with: beast
5396 [13/03/2022 10:34] Wassup G
5396 [13/03/2022 10:35] awe G
```

```
5981 [13/03/2022 10:34] Morning king
5981 [13/03/2022 10:35] It's all good
```

Group Chat



Client Login
FROM: memOne

Client Login
FROM: memTwo

Client Login
FROM: memThree

```
    server.startRun()
  File "Server.py", line 35, in startRun
    login,lgnAdr = self.loginSock.recvfrom(bufferSize)
KeyboardInterrupt
mthoko@mthoko-VirtualBox:/media/sf_Ubuntu_Shared/NetAssi
gn/Attempt$ python3 Server.py
Server running and Listening
Data from login ['memOne', 'GROUP', '5995']
Data from login ['memThree', 'GROUP', '5916']
Data from login ['memTwo', 'GROUP', '5533']
```

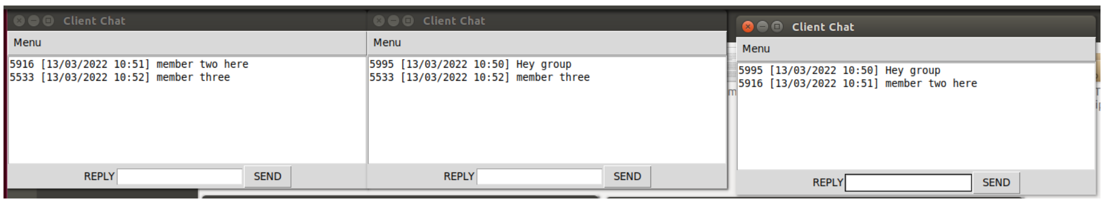
```
beast13/03/2022 10:35 awe G

5396

mthoko
One received by rcvSk:5396 shot
mthoko@mthoko-VirtualBox:/media/sf_Ubuntu_Shared/NetAssign/
Attempt$ python3 Client.py
Local Port: 5995
```

```
mthoko@mthoko-VirtualBox:/media/sf_Ubuntu_Shared/NetAs
sign/Attempt$ python3 Client.py
Local Port: 5396
One received by rcvSk:5981 Morning king
One received by rcvSk:5981 It's all good
mthoko@mthoko-VirtualBox:/media/sf_Ubuntu_Shared/NetAs
sign/Attempt$ python3 Client.py
Local Port: 5533
```

```
mthoko@mthoko-VirtualBox:/media/sf_Ubuntu_Shared/NetAssign/
mthoko@mthoko-VirtualBox:/media/sf_Ubuntu_Shared/NetAssign/
pt$ python3 Client.py
Local Port: 5916
```



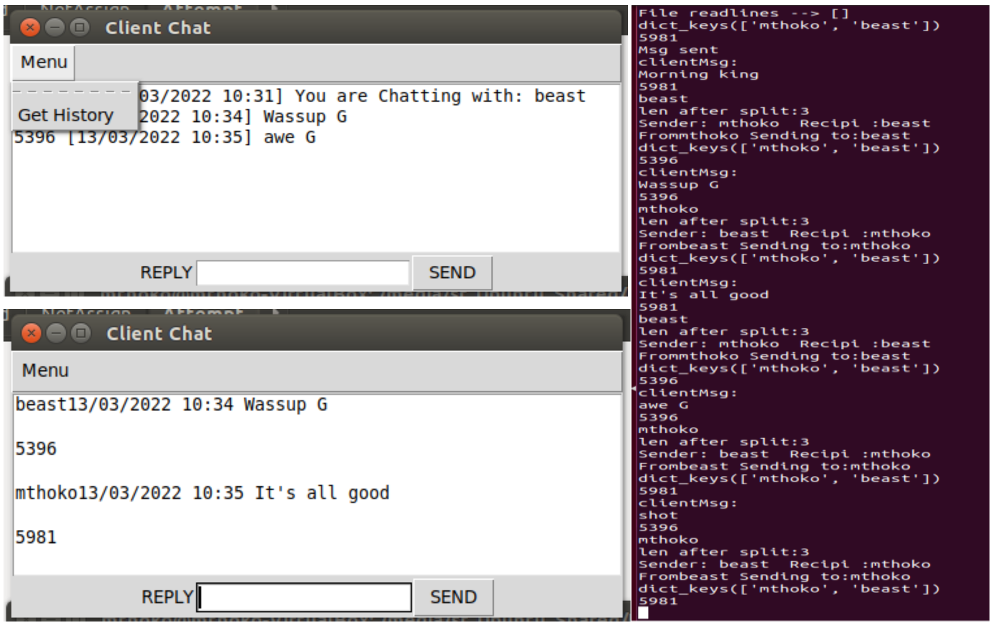
```
5916 [13/03/2022 10:51] member two here
5533 [13/03/2022 10:52] member three
```

```
5995 [13/03/2022 10:50] Hey group
5533 [13/03/2022 10:52] member three
```

```
5995 [13/03/2022 10:50] Hey group
5916 [13/03/2022 10:51] member two here
```

Messages are sent to every member of the group chat except the sender. A member can log off and login later then retrieve the chat history to access the missed messages.

Get History and Server monitoring activity



```
03/2022 10:31] You are Chatting with: beast
2022 10:34] Wassup G
5396 [13/03/2022 10:35] awe G
```

```
beast13/03/2022 10:34 Wassup G

5396

mthoko13/03/2022 10:35 It's all good

5981
```

```
File readlines --> []
dict_keys(['mthoko', 'beast'])
5981
Msg sent
clientMsg:
Morning king
5981
beast
len after split:3
Sender: mthoko  Recipi :beast
Frommthoko Sending to:beast
dict_keys(['mthoko', 'beast'])
5396
clientMsg:
Wassup G
5396
mthoko
len after split:3
Sender: beast  Recipi :mthoko
Frombeast Sending to:mthoko
dict_keys(['mthoko', 'beast'])
5981
clientMsg:
It's all good
5981
beast
len after split:3
Sender: mthoko  Recipi :beast
Frommthoko Sending to:beast
dict_keys(['mthoko', 'beast'])
5396
clientMsg:
awe G
5396
mthoko
len after split:3
Sender: beast  Recipi :mthoko
Frombeast Sending to:mthoko
dict_keys(['mthoko', 'beast'])
5981
clientMsg:
shot
5396
mthoko
len after split:3
Sender: beast  Recipi :mthoko
Frombeast Sending to:mthoko
dict_keys(['mthoko', 'beast'])
5981
```

Specification (Protocol design & specification (sequence diagrams & message formats/structure))

Message Protocol

Login

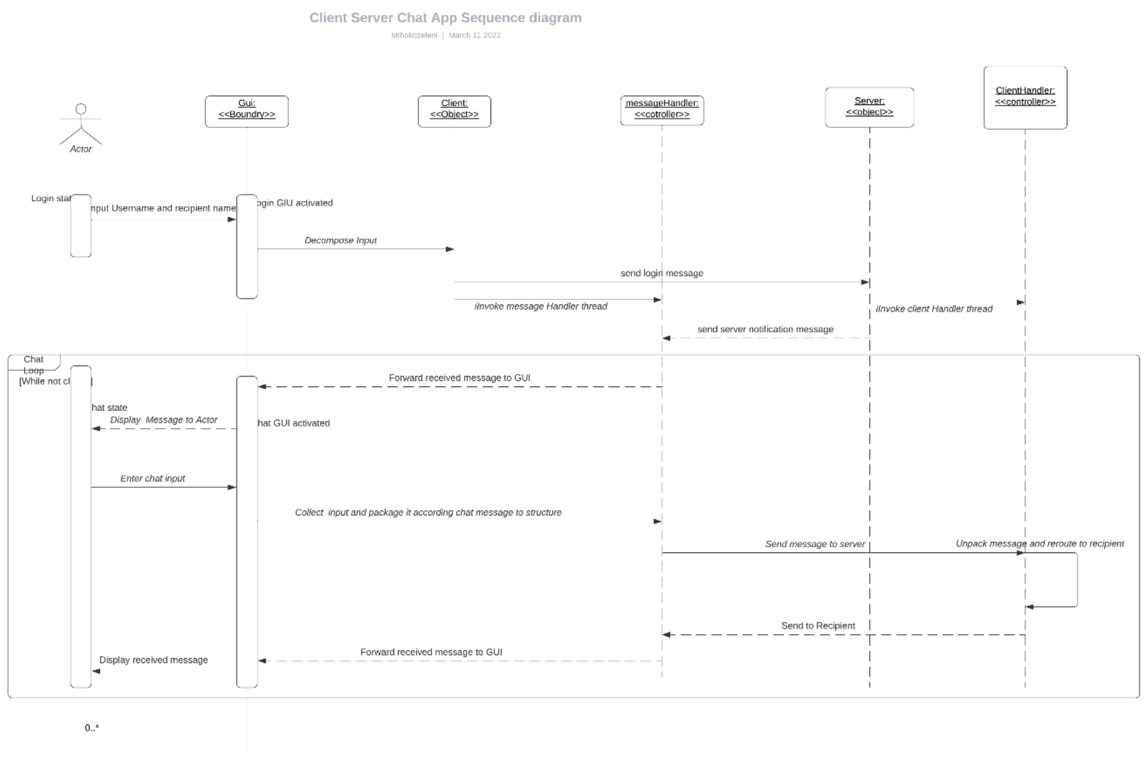
Source Port	Destination Port
length	Checksum
Body: login message= Username +\n+	Recipient +\n+ Sender port number

Chat

Source Port	Destination Port
length	checksum
Body: Message +\n+ sender Port number +\n+	Port number +\n+ Recipient username

The header of the UDP protocol includes the source and destination port, length and checksum which are required by the sockets and the body of the carries the message encoded into bytes. The format of the message helps the server implement our manual protocol, the server utilizes the username, recipient name and sender port number to create a connection from the login packet/data. Once in the chat state the app uses the format of the message to identify the appropriate recipient and a port number to maintain connection to respond and for the identify whom the message comes from.

Sequence Diagram(If not visible PNG folder is attached in submission)


Client Server Chat App Sequence diagram
Mthokozeleni | March 11 2022
Actor
Gui: <<Boundry>>
Client: <<Object>>
messageHandler: <<cotroller>>
Server: <<object>>
ClientHandler: <<controller>>
Login stat
nput Username and recipient name
ogin GIU activated
Decompose Input
send login message
iInvoke message Handler thread
iInvoke client Handler thread
send server notification message
Chat Loop
[While not cl
Forward received message to GUI
hat state
Display Message to Actor
hat GUI activated
Enter chat input
Collect input and package it according chat message to structure
Send message to server
Unpack message and reroute to recipient
Send to Recipient
Forward received message to GUI
Display received message
0..*

The sequence diagram explains the flow of user input from the GUI into the server and to the recipient. And the change of states from login state to the chat state throughout the process.

Conclusion

The are many challenges that I faced while trying to implement the chat app, the major challenges include communicating and updating variables in the main thread as they are manipulated by the client or message handler threads. The Tkinter GUI library can only be accessed or manipulated by the main thread on which is created and the mainloop method is called, this lead to the creation of a separate login and chat GUI. Simulation errors was also a challenge as I do not how that can be implemented. Whilst logging in control messages sent by server are not received until the login window is closed. Beyond that the chat app is quite robust and can handle different situations such linked logins where one user is in a private chat then a third user logs in to chat with one the active users.

References

docs.python.org, www.guru99.com, stackoverflow.com , www.simplilearn.com, www.w3schools.com, www.tutorialspoint.com, superuser.com, www.adamsmith.haus, realpython.com, www.geeksforgeeks.org, python.tutorialink.com, www.freecodecamp.org, clouds.eos.ubc.ca, pythontic.com, www.reddit.com, github.com , help.socketlabs.com, careerkarma.com, cppsecrets.com, www.techwithtim.net, medium.com, developer-shubham-rasal.medium.com, python.plainenglish.io, www.linkedin.com, pythonprogramming.net, python-docs.readthedocs.io, www.programiz.com, www.techwalla.com, www.delftstack.com, www.studytonight.com, www.programcreek.com, codefather.tech, www.learndatasci.com, www.onooks.com, python-list.python.narkive.com, www.codespeedy.com, www.youtube.com,