# COMP28112 Lab 2 Report

Jeremy B. Roe

## 1 A More Robust Messaging System for Healthcare Professionals

### How to run my code

#### Run server

```
python3 ./myserver.py localhost 8090
```

#### Run client

This will create a GUI. To send messages, type into input bar and press button (or press enter) to send request to server.

```
python3 ./myclient.py localhost 8090
```

### Task 2.1

```python
class MyServer(Server):

    def onStart(self):
        print("Server has started")

    def onStop(self):
        print("server ended")

    def onConnect(self, socket):
        print("user connected")

    def onMessage(self, socket, message):
        print(f"message received: {message}")
        message = message.upper().encode()
        socket.send(message)
        return True

    def onDisconnect(self, socket):
        print("\ndisconnecting users...")
```

myServer.py

## Task 2.2

```python
def __init__(self):
    super(MyServer, self).__init__()
    self.userCount = 0

def onDisconnect(self, socket):
    self.userCount -= 1
    print(f"disconnecting user. "
    f"Current number of users: {self.userCount}")
```

myserver.py − > changes

## Task 2.3

```python
def onMessage(self, socket, message):
    (command, sep, parameter) = message.strip().partition(' ')
    print(f"command: {command}")
    print(f"Message: {parameter}")
```

myserver.py − > changes

## Task 2.4

My protocol is designed as follows:

The user must register by typing in the command *SET_NAME <username>* in order to send messages. The username must not contain any spaces and the username cannot already be used. If the user is not registered, they can only use the following commands: *SET_NAME, HELP, and CLOSE.* Once registered, user can use all commands.

```
SET_NAME <username>
HELP
CLOSE
MESSAGE <message>
DM <user> <message>
```

If the user does not type the command and its corresponding parameters in the correct format, they are told that it is not in the correct parameter and should try again.

The code in the server should be structured as follows:

```
DEF onMessage(self, socket, message):
    Split message into variables Command and Parameters.

    CASE Command{
        "HELP":
            Output the list of all commands and their required parameters
            BREAK

        "CLOSE":
            Close the connection between socket and server
            BREAK

        "LIST":
            Sends user a list of all connected users
            BREAK

        "SET_NAME":
            Add username and socket to a dictionary if username is
            valid & not already in the dictionary
            BREAK

        "MESSAGE":
            IF username is set
                Send message if there is more than 1 user in dictionary
                and parameters is not an empty string (meaning there is
                someone to send messages to). Else, say appropriate error
                message
            ELSE
                Tell user to register
            BREAK

        "DM":
            IF username is set
                Split Parameters in to sendUser and Message.
                Send message to person if sendUser exists, is not
                the socket, and message is not an empty string
            ELSE
                Tell user to register
            BREAK

        "":
            BREAK

        DEFAULT:
            Tell socket that the command is not valid
    }
```

When the user wants to close the connection, we pop them from the list and update the count for the number of users:

```
def onDisconnect(self, socket):
    if socket in self.users.values():
        print(
            f"disconnecting"
            f"{list(self.users.keys())
            [list(self.users.values()).index(socket)]}."
            f" Current number of users: {self.userCount-1}")
        self.users.pop(
            list(self.users.keys())
            [list(self.users.values()).index(socket)])
    else:
        print(f"disconnecting user. "
        f"Current number of users: {self.userCount-1}")
    self.userCount -= 1
```

onDisconnect function in myserver.py

## Task 2.5

*See **myserver.py** for implementation*

## Task 2.6

I decided to write the client with a GUI (Graphical User Interface). The module used is Tkinter and it is included in all standard Python distributions.

*See **myclient.py** for implementation*

## Task 2.7

### Q1:

The protocol I created is not stateless. A stateless protocol is when no information is remembered about the client from previous requests. Stateless protocols are simple in design and handles transactions very quickly. An example of this is lab1. Lab1 was stateless as, although information was being stored in the server, the information was not linked to the actual user itself. The server was also simple as all it does is store/edit/delete what the user wants in a dictionary. It did not store any information about the client itself.

A stateful protocol is when the server retains information about the session details and the status information of the client. These types of protocols are a lot more complex in design and as a result, transactions are handled much slower than stateless. The sever kept information about the client by storing the client information in a dictionary, *self.users*. This then means that the server has information about all clients and can send messages to them when necessary.

**Q2:**

One way of improving the implementation would be creating a dictionary of all groups where the key is the group name, and the value is a list of the sockets of all users that are in that group. Once the user has set their username, they will only be able to use the commands *SET_NAME, HELP, and CLOSE*, as well as the following new commands *NEW_GROUP, LIST_GROUP, and JOIN_GROUP*. These new commands will have the following formats:

```
NEW_GROUP <group_name>
LIST_GROUP
JOIN_GROUP <group_name>
```

list of commands for groups

The protocol will be edited to include these cases:

```
"NEW_GROUP":
    IF groupname valid (no spaces), username is set,
    and does not exist:
        create group and add user to the test
    ELSE
        output appropriate error message
    BREAK

"LIST_GROUP":
    IF username is set:
        create group and add user to the test
    ELSE
        Tell user to sign in
    BREAK

"JOIN_GROUP":
    IF username is set, group exists and not
    in group already:
        exit current group if in a group,
        then join new group
    ELSE
        Tell user appropriate error
    BREAK
}
```

Psuedocode of roughly how group commands will be implemented

List will then show the people in their group, followed by all users (or not depending on if you want to see people not in your group). They will only be able to send messages/DMs to people within their group and cannot message those outside their group unless they change groups.