Report

This report is an analysis of a chat application and the protocol that supports the application. It encompasses features of the application, the protocol design and the individual client implementations of the application

1. Protocol Design and Functionality

The protocol's design was developed with the following rules in mind: these are rules that all client implementations and communications must follow.

Protocol Rules:

- **Message Encoding**: Exchanging of messages between clients and servers should be encoded in ASCII.
  All queries to the server must start with "/", else the message is processed as normal or not at all.
- **Message Length:** A maximum of 1024 bytes can be exchanged between clients and the server.
- **Connection Management:** All clients must be actively connected to the server while running.
  To close the connection to the server – clients must terminate their programs.

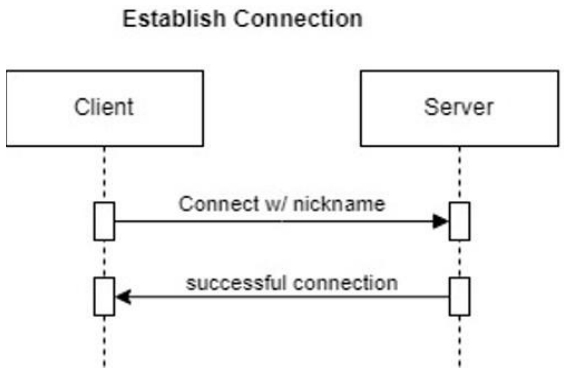
  ConnectionResetError is handled by removing the client from the list of active clients.

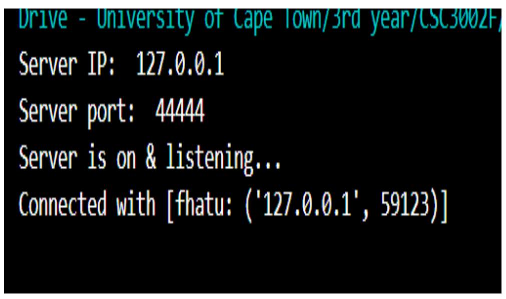
1.1 Client connection:

When a client connects to the server, the client sends their nickname to the server.

The server acknowledges this connection and adds the client to the list of active clients that are connected to the server

(see sequence diagram below + screenshot of server output)





```
Drive - University of Cape Town/3rd year/CSC3002F/
Server IP:  127.0.0.1
Server port:  44444
Server is on & listening...
Connected with [fhatu: ('127.0.0.1', 59123)]
```

1.2. Message Exchange:

Clients can send numerous messages to the server.

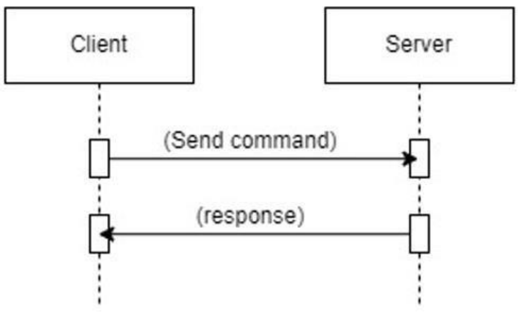
The server processes these messages and responds accordingly.

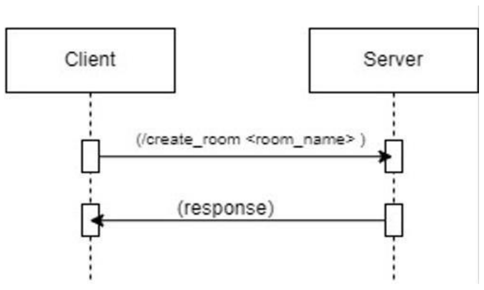
Valid commands are:

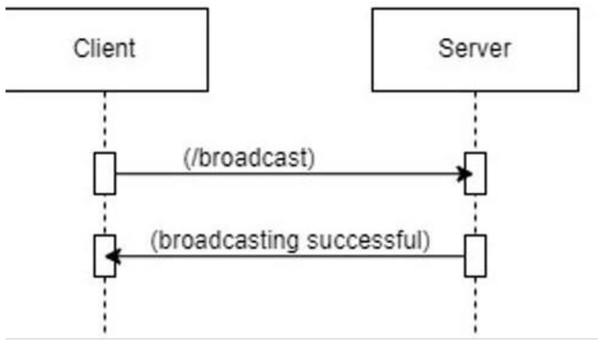
- /members : to obtain a list of active members
- /hide : to hide the client’s nickname from list of active members
- /reveal : to show the client's from list of active members
- /broadcast : to broadcast a message to all active members
- /create_room : to create a new chatroom
- /join : to join a chatroom
- /room : to send a message to a chatroom
- /get_rooms : to obain a list of chatrooms
- /leave : to exit a chatroom

(see sequence diagrams)

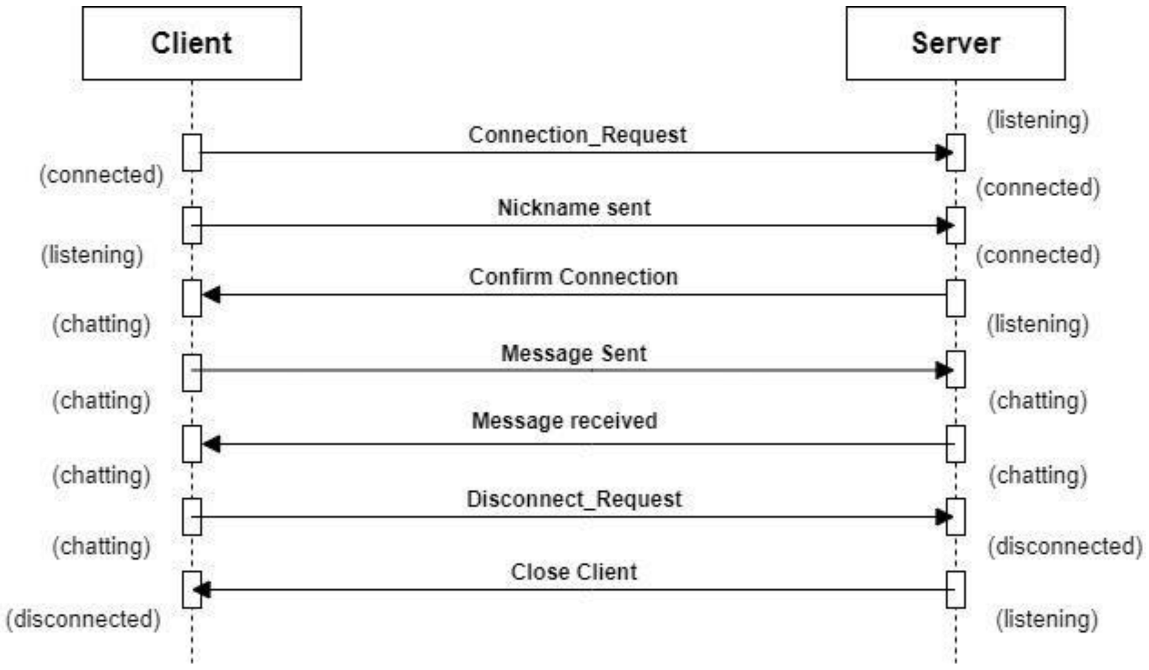
Send/Receive Message





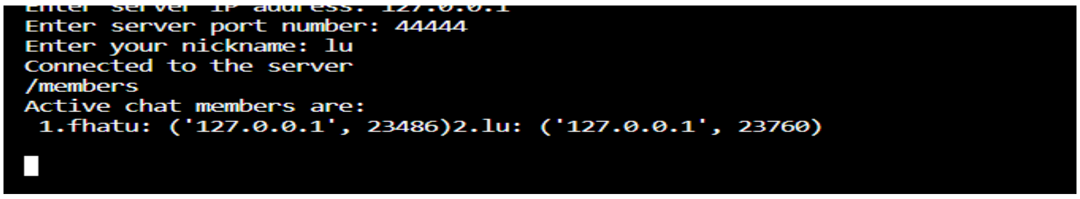


Below is a sequence diagram indicating the different states in which the server and client are in and the events that prompt these events:



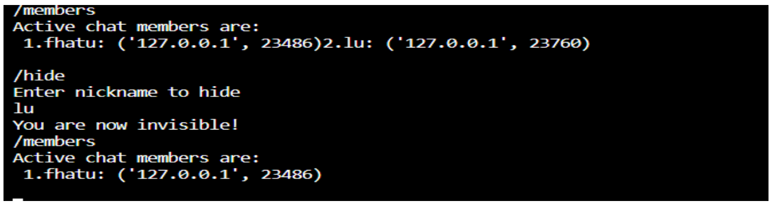
APPLICATION FEATURES

Available users: list of available users, with their ip address and port at which they're listening at.
Reasoning: a user might need to know who is available for communication if they want to broadcast a message or know who they can communicate with privately. This way, if the user they want to communicate with is not available, clients will not waste time trying to connect to an unavailable user. Including ip address and port number means any client who wants to connect to another active may get their details to request connection.
How it works: If the message the server receives is "/members", the server will return a list of the currently connected and visible users with their IP address and their port number.

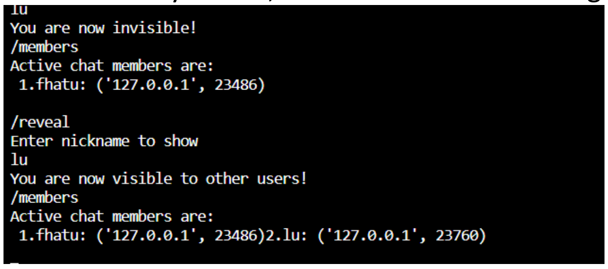


```
Enter server port number: 44444
Enter your nickname: lu
Connected to the server
/members
Active chat members are:
 1.fhatu: ('127.0.0.1', 23486)2.lu: ('127.0.0.1', 23760)
```

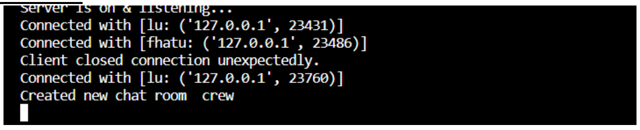
Visibility: connected users can choose to be visible to/hidden from other users
Reasoning: Sometimes a person might be connected but busy or not wanting any connections, or to receive broadcast messages. This feature gives the users the option to not appear as online, while staying connected. They can then choose to appear to other users when they please. This ensures have some level of control of when other users can communicate with them, and their privacy.
How it works:
If the message received by the server starts with "/hide", the server prompts the client to send the nickname to be hidden, then removes that nickname from the list of available users. Updated list of members will not contain this client's nickname.


```
/members
Active chat members are:
 1.fhatu: ('127.0.0.1', 23486)2.lu: ('127.0.0.1', 23760)

/hide
Enter nickname to hide
lu
You are now invisible!
/members
Active chat members are:
 1.fhatu: ('127.0.0.1', 23486)
```

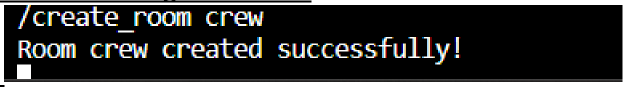
If the message starts with "/reveal", then the program will first check that user doesn't appear on the current list of active clients, then add them to it if that's the case. If the user is already visible, the server will do nothing.


```
lu
You are now invisible!
/members
Active chat members are:
 1.fhatu: ('127.0.0.1', 23486)

/reveal
Enter nickname to show
lu
You are now visible to other users!
/members
Active chat members are:
 1.fhatu: ('127.0.0.1', 23486)2.lu: ('127.0.0.1', 23760)
```

Creating new chatroom: this feature allows connected users to create a new chatroom where multiple clients can communicate with each other.
Reasoning: When users want to discuss something once off with people of like interests, they can just do so all at once. Instead of individual conversations where the same information would have to be shared repeatedly. Chatrooms also allow people to view announcements from others and others' thoughts without having to be responsive. They are great avenues to sharing knowledge to people who care about it. This is better than broadcasting messages to all connected users because users would just get flooded with irrelevant messages that way.
How it works**:** the client must send the "/create_room" command accompanied by the name the user wants to give the chatroom. When the server receives "/create_room" command, it checks if the chat room name is unique, if the name already exists, the user will be notified then the user can choose to change the chat room name or join that existing chatroom. The server sends a message to notify the client of the created chatroom. The creator is automatically a part of the chatroom.
Server:

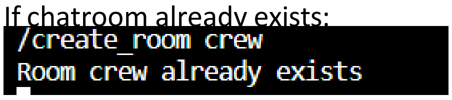

```
Server is on & listening...
Connected with [lu: ('127.0.0.1', 23431)]
Connected with [fhatu: ('127.0.0.1', 23486)]
Client closed connection unexpectedly.
Connected with [lu: ('127.0.0.1', 23760)]
Created new chat room  crew
```

Client creating chatroom:



```
/create_room crew
Room crew created successfully!
```

If chatroom already exists:



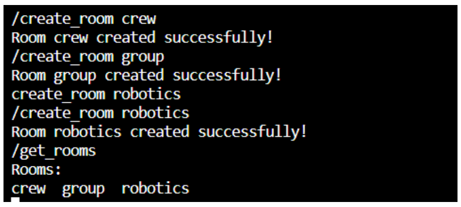
```
/create_room crew
Room crew already exists
```

Joining chatrooms:

Clients can join existing chatrooms.

Reasoning: if a client wants to join a chatroom that's based on a certain topic, they can join an existing one instead of having to create their own. This limits the number of chatrooms created, thus limiting the amount of computer resources required.

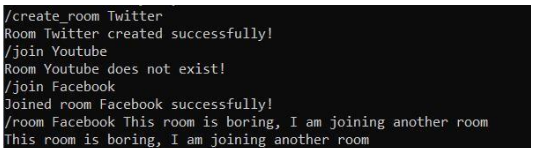
How it works: server receives **"**/join" message, if the client is not already in specified chatroom, client is added. The client is then notified of the chatroom they have joined. The Server also prints "{client} joined {room name} and notifies the other users. If the client is already a member of that chatroom, they are notified of this fact and not re-added. If the chatroom is not on the list of existing chatrooms, the client is told they are attempting to join a non-existent chatroom. **List of existing rooms** allows clients to get a list of chatrooms they can join using "/get_rooms", so they don't have to create several chatrooms for similar discussions.



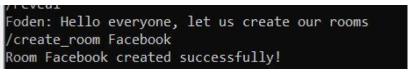
```
/create_room crew
Room crew created successfully!
/create_room group
Room group created successfully!
create_room robotics
/create_room robotics
Room robotics created successfully!
/get_rooms
Rooms:
crew  group  robotics
```

Users can also **send messages in the chatroom**, by sending a message that's starts with "/room", followed by the chat room they want to broadcast to and then the message. If the room does not exist, the user is notified. Clients can **also leave chatrooms** by using the "/leave" command. The client is notified after they are removed from the chatroom. If a user tries to, leave a chatroom they are not a member of, they're notified.

More Chatroom commands in action:

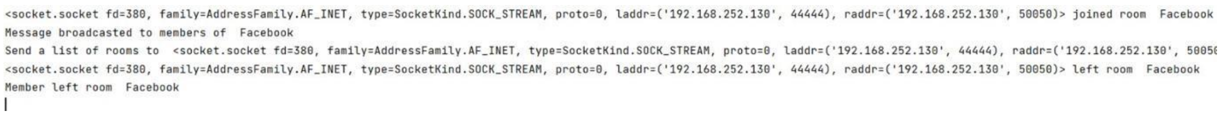


```
/create_room Twitter
Room Twitter created successfully!
/join Youtube
Room Youtube does not exist!
/join Facebook
Joined room Facebook successfully!
/room Facebook This room is boring, I am joining another room
This room is boring, I am joining another room
```



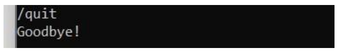
```
Foden: Hello everyone, let us create our rooms
/create_room Facebook
Room Facebook created successfully!
```

Server:



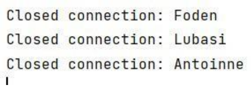
```
<socket.socket fd=380, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.252.130', 44444), raddr=('192.168.252.130', 50050)> joined room  Facebook
Message broadcasted to members of  Facebook
Send a list of rooms to  <socket.socket fd=380, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.252.130', 44444), raddr=('192.168.252.130', 5005
<socket.socket fd=380, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.252.130', 44444), raddr=('192.168.252.130', 50050)> left room  Facebook
Member left room  Facebook
```

When client disconnects:



```
/quit
Goodbye!
```

Server:



```
Closed connection: Foden
Closed connection: Lubasi
Closed connection: Antoinne
```

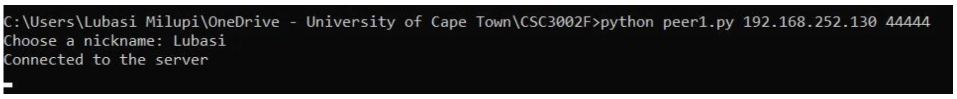
CLIENT IMPLEMENTATION FOR LUBASI MILUPI (MLPLUB001)

As I delved into the implementation of the client-side code for Network Assignment 1, my primary goal was to create a versatile and interactive script capable of handling both TCP and UDP communication. The script is within a Python class named `Client`, designed to facilitate seamless interaction with a server using sockets. The initiation process prompts the user to choose a nickname and provides essential server connection details, including the server's IP address, main communication port, and a supplementary UDP port. The script sets up both TCP and UDP sockets for communication, establishing a connection with the server and binding the UDP socket to the specified IP and port. Threading plays a crucial role in ensuring concurrent execution of various tasks. Three threads are employed: one for receiving messages from the server (`receive_thread`), another for handling private messages via UDP (`udp_recv_thread`), and a third for managing user input and sending messages (`write_thread`). The communication protocol between the client and server is in such a way to exchange critical information. Upon receiving the 'Nickname' prompt from the server, the client responds by sending its chosen nickname. If the server requests the UDP port number ('port'), the client promptly sends it. This ensures a seamless handshake between the client and server. Private messages are a distinct feature, allowing users to initiate private conversations. The user inputs '/private,' specifies the recipient's port, and provides the message. The script then sends this message over UDP, appending the sender's nickname for identification. A 'quit' command terminates the client, closing both the TCP and UDP sockets. Other messages and prompts are sent to the server for execution. In terms of execution, the script accepts command-line arguments for server IP and port, providing flexibility for different users (i.e tutors, students and lecturers) to specify these parameters during runtime. A clear usage prompt is displayed in case of incorrect arguments.

The usage of the client code will be shown below:

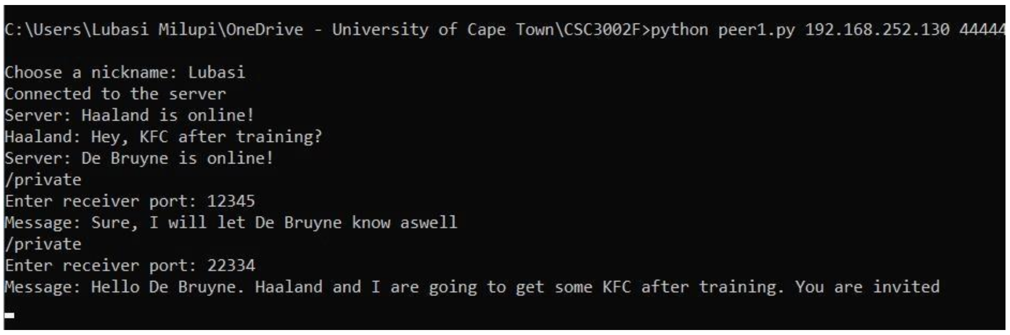
In this example, the udp port numbers were as follows: Lubasi (34567) Haaland (12345) De Bruyne (22334)

Connecting to the server:



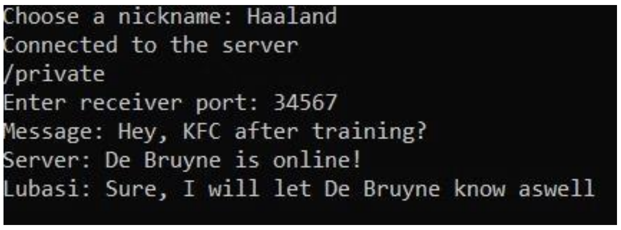
```
C:\Users\Lubasi Milupi\OneDrive - University of Cape Town\CSC3002F>python peer1.py 192.168.252.130 44444
Choose a nickname: Lubasi
Connected to the server
```

Sending messages:



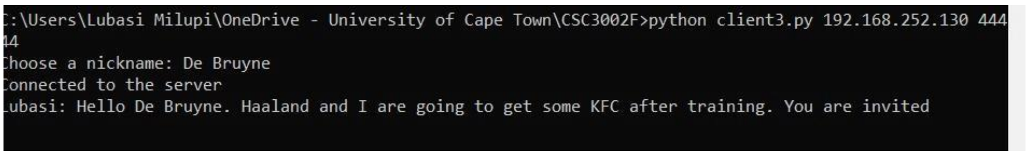
```
C:\Users\Lubasi Milupi\OneDrive - University of Cape Town\CSC3002F>python peer1.py 192.168.252.130 44444

Choose a nickname: Lubasi
Connected to the server
Server: Haaland is online!
Haaland: Hey, KFC after training?
Server: De Bruyne is online!
/private
Enter receiver port: 12345
Message: Sure, I will let De Bruyne know aswell
/private
Enter receiver port: 22334
Message: Hello De Bruyne. Haaland and I are going to get some KFC after training. You are invited
```

Haaland side:



```
Choose a nickname: Haaland
Connected to the server
/private
Enter receiver port: 34567
Message: Hey, KFC after training?
Server: De Bruyne is online!
Lubasi: Sure, I will let De Bruyne know aswell
```

De Bruyne side:



```
:\Users\Lubasi Milupi\OneDrive - University of Cape Town\CSC3002F>python client3.py 192.168.252.130 444
44
hoose a nickname: De Bruyne
onnected to the server
ubasi: Hello De Bruyne. Haaland and I are going to get some KFC after training. You are invited
```

To conclude, this client-side setup provides a solid base for network chatting. The way it's built, the threads it uses, and the fact that it can handle both TCP and UDP make it flexibleand ready for all kinds of network adventures.