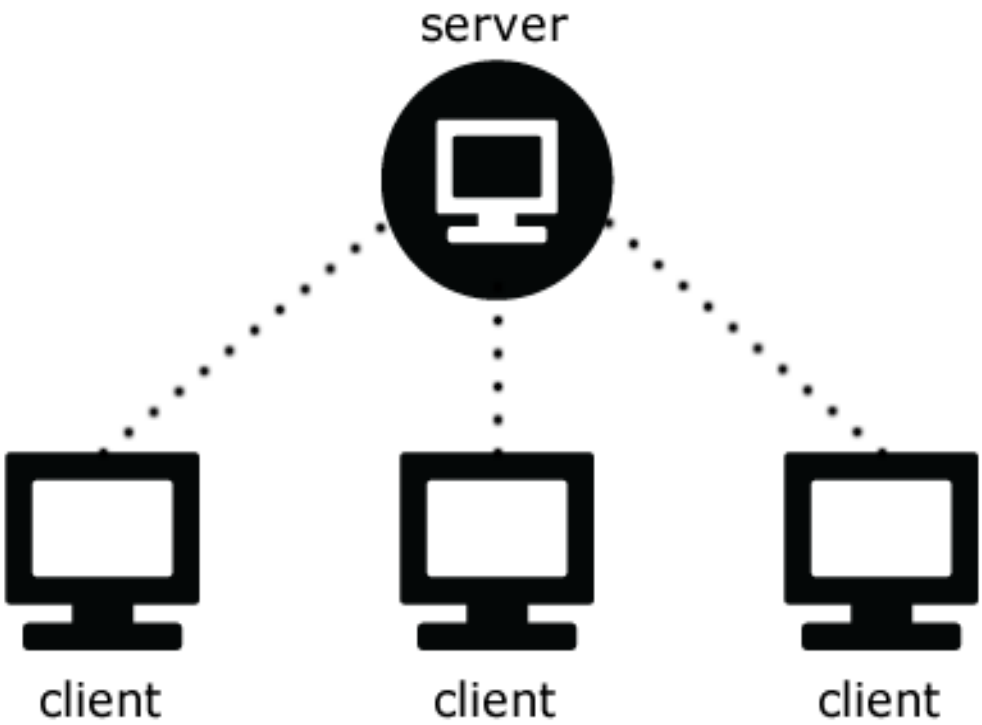
CSC3002F Assignment 1
Networks
Report



Team Members
Botshelo Nokoane - NKNBOT001
Dikatso Moshweunyane - MSHDIK008
Libhongo Mko - MKXLIB001

TABLE OF CONTENTS

Introduction

This assignment is on networked applications where students are required to develop a Python-based client -server chat application. The application design uses UDP at the transport layer that mimics the handshaking protocol that is only available in TCP. It supports the client -server architecture and implements a server that should manage the interaction between clients. It allows multiple pairs/groups of users to exchange messages in real-time. The different clients and servers are able to run on different hosts over a network. The client interface is a CLI with appropriate user menus.

Protocol Requirements and Constraints

Real-time interaction

- The protocol must allow pairs/groups of users to exchange messages in real-time

Loss Detection

- The protocol should confirm that all messages sent are received.

Error Detection

- The protocol should allow messages exchanged to be verified for any errors using a hash function both on the client and server

Reliability

- The protocol should ensure reliability and ensure that no message is lost along its way to the receiving end.
- Different clients and the server should be able to run on different hosts over a network

Protocol Design

Pattern of communication

Client-server communication

- Server responsible for the overall control and coordination of communication between various clients

Message Structure

The client and the server use the same message structure across all functionalities.
This is the general structure that's used to send messages.

```
message_structure = {
  "header": {
    "command": string,
    "message_type": string,
    "recipient_data ": {
      "ip_address": string,
      "port_number ": int
    }
  },
  "body": string or {
    "username": string,
    "password": string,
  }
}
```

The message structure looks more like a dictionary in python and contains fields that are used to make decisions on the server and on the client. The header contains fields that describe the actual data in the message. The command specifies what the clients want to do, which can be establishing a connection with the server, sending a message to a specific or all client(s), listing all online clients, or disconnecting with the server. The messa ge type can be a **command message** which defines the different stages of communication between parties, such as the initiation or termination of a session. A **control message** which manages the dialogue between parties, such as message acknowledgments and retr ansmission requests. And **data transfer messages** which carry the data that is exchanged between parties. The recipient data contains the recipient's address information which compromises the client's IP Address and port number. Depending on the command sent by the client, the body may contain the actual message the client wants to be sent to the server or the credentials of the user like username & password for authentication or

sometimes it can contain the sender and receiver username as well as the text, t hese fields are used when sending a text message to a client.

Here is an example of how the message structure is used in the application,

```
message_struct = {
    "header": {
        "command": "SEND_ALL_MESSAGE",
        "message_type": "MESSAGE_EXCHANGE",
        "recipient_data": (serverName, serverPort),
    },
    "body": message
}      Botshelo Nokoane, 3 days ago • implement group messag
```

Here is what a message will look like when passed around,

```
{'body': {'text': 'Hello there!', 'sender_username': 'Dikatso', 'receiver_username': '
Botshelo'}, 'header': {'message_type': 'MESSAGE_EXCHANGE', 'command': 'SEND_MESSAGE',
'recipient_data': ('botshxlo-Lenovo-V130-15IKB', 12000)}}
```

Description of System Functionality

The chat application was implemented using Python and comprises a CLI (client.py) and a Server (server.py). Both of the applications use UDP at the transportation layer and support client -server architecture.

CLI

This application utilizes threads for sending and receiving messages concurrently and independently from the server. For receiving messages, the **receive** method uses a While loop to continuously listen or receive messages from the server.

Characters are pulled into a string and that string is later transformed into a dictionary to produce a message structure that's used across the application.

```
message, _ = clientSocket.recvfrom(2048)

# transform dictionary in text format to dictionary format (text -> dict)
dictMessage = ast.literal_eval(message.decode())
```

ex: message structure

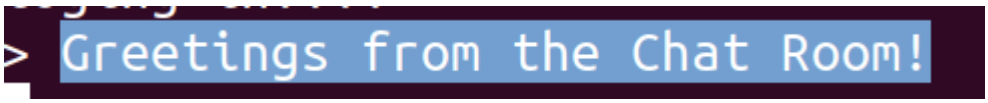
```
message_struct = {
    "header": {
        "command": "SEND_MESSAGE",
        "message_type": "MESSAGE_EXCHANGE",
        "recipient_data": (serverName, serverPort),
    },
    "body": message
}
```

Then certain fields are pulled out of the dictionary to check the commands and type of messages received from the server.

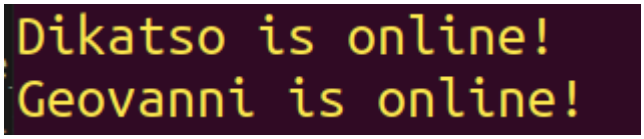
```
header = dictMessage["header"]
message_type = header["message_type"]
command = header["command"]
message = dictMessage["body"]
```

If a command was not specified in the data object (message_struct) then depending on the type of message received from the server, the CLI will print out either

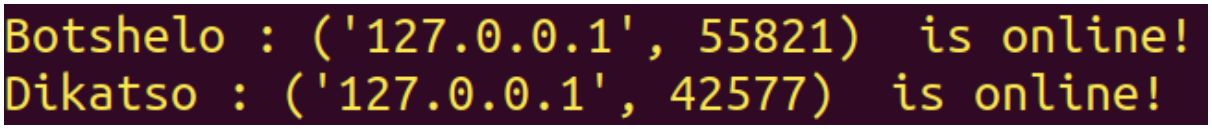
A GREETINGS message



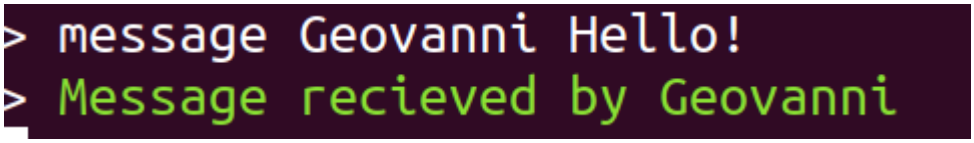
An ALERT message (e.g when a user logs in)



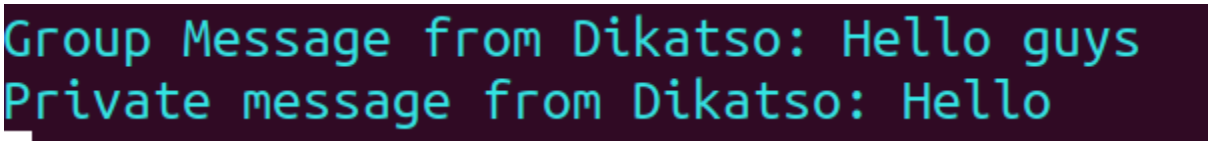
A LIST_CONTACTS message (listing all online users)



An ACKNOWLEDGEMENT message (in green)



A private or group CHAT message



If a client receives a chat message then the client will send an acknowledge message to the identifying sender.

```
# acknowledge message was recieved if type of message is MESSAGE_EXCHANGE (client-to-client)
if message_type == "MESSAGE_EXCHANGE":
    username = message.split(" ")[3].split(":")[0] # username of the client to send acknowledgement message to
    message = {
        "username": username,
        "text": "Message recieved by " + currently_logged_user["username"],
    }

    send_acknowledgement_message(message)
```

Another method that runs within a thread called **establish_connection_and_commands** takes inputs from the user via CLI and performs certain oper ations based on the commands given. The method uses a While loop to constantly listen or receive messages coming from the server. To establish a connection between a client and the server,

- The user types in the **login** command
- The CLI will take in the username and check it against the Database records.
- If a username exists within the Database records then the CLI will continue to take in the password and send the user credentials to the server to establish a connection.
- In the case where a username doesn't exist then, the user will be asked to enter a password for the username they typed and the user credentials will be stored in the Database and locally.
- A message will be sent to the server to indicate that a user establishes a connection and will be stored in a list of all online users.
- Then the server will alert all other online users that a user is online.

```
if command == "login":

    # check if user not logged in already
    if currently_logged_user["username"] != "":
        print("alread logged in... as " + currently_logged_user["username"])
        continue

    username = raw_input("enter username: ")

    # check if username is not already in use

    user = auth.findUser(username)

    if user["success"]:

        # try user password until user loggs in
        while len(currently_logged_user["username"]) == 0:

            # compare password
            password = raw_input("enter password: ")

            if auth.password_equal(password, user["password"]):

                # found user, proceed with login
                userCredentials = {"username": username, "password": auth.hash_password(password)}

                print("loging in....")
                time.sleep(1)
                login_and_connect_client(userCredentials)
```

```
                # store user locally
                currently_logged_user["username"] = username
                currently_logged_user["password"] = auth.hash_password(password)
                break
            else:
                print("Wrong password!")

# user not found
else:
    print("User record doesn't exist")
    answer = raw_input("sign up ? Y/N... ")

    if answer in ["Y", "Yes", "y", "yes"]:
        password = raw_input("enter password: ")

        print("signing up...")
        time.sleep(1)

        # add user to database
        auth.addUser(username, password)

        userCredentials = {"username": username, "password": auth.hash_password(password)}

        print("loging in....")
        time.sleep(1)
        login_and_connect_client(userCredentials)

        # store user locally
        currently_logged_user["username"] = username
        currently_logged_user["password"] = auth.hash_password(password)
```

To log out of a session with the server,

- The user needs to enter the **logout** command,
- Locally stored user credentials will be cleared and a message will be sent to the server to indicate that a user logs out and that user will be removed from a list of all online users (store on the server).

```
if command == "logout":
    print("logging out...")
    time.sleep(1)

    currently_logged_user["username"] = ""
    currently_logged_user["password"] = ""
    logout_and_disconnect_client()
```

To message an online user-private messaging,

- The user will have to input “message <receiver_username> <message>”
- Then the client will send a message to the server, containing the command, message type, receiver username, and sender username.
- Then the server will redirect that message to the receiving user.

```
elif command == "message":
    receiver_username = user_input[1]
    text = " ".join(user_input[2:])      You, 2 days ago • 🔒 SECURITY

    if receiver_username == currently_logged_user["username"]:
        print("messages to oneself not allowed!")
    else:
        message = {
            "sender_username": currently_logged_user["username"],
            "receiver_username": receiver_username,
            "text": text
        }
        message_to_online_user(message)
```

-

To message all online users - group messaging,

- The user will have to input “message* <message>”
- Then the client will send a message to the server, containing the command, message type, and sender username.
- Then the server will redirect that to al l online users.

```
elif command == "message*":
    text = " ".join(user_input[1:])

    message = {
    "sender_username": currently_logged_user["username"],
    "receiver_username": "",
    "text": text
    }
    message_to_all_users(message)
```

Server

The server is the one that handles all the messages from the clients.

When the server receives a **login** command from a client, the server will try to mimic the hand - shaking protocol connection which is not supported in UDP Socket connections.
The server will then add the user to a list of all online users and send back a greetings message to the user.

```
if command == "LOGIN":
    """ mimic handshaking protocol connection
        login user and store user credentials together with client address
    """
    user = dictMessage["body"]
    username = user["username"]
    password = user["password"]
otshelo Nokoane, last week • Update server …
    # store user in user-dictionary
    clients.append({ "username": username, "password": password, "address": clientSenderAddress}) # store client data

    message_struct = {
        "header": {
            "command": "",
            "message_type": "GREETING",
            "recipient_data": clientSenderAddress,
        },
        "body": "Greetings from the Chat Room!",
    }

    toBeSentMessage = str(message_struct).encode()
    serverSocket.sendto(toBeSentMessage, clientSenderAddress)
```

The server will then send a message to all online users to notify them of this user's online status.

```
    # find all users to send online message (xxxxx user is online!)
    for client in clients:
        address = client["address"]
        message_struct = {
            "header": {
                "command": "",
                "message_type": "ALERT",
                "recipient_data": address,
            },
            "body": "{0} {1}".format(username, "is online!")
        }

        # send message to all users except the user (this user) that just joined the chat
        if clientSenderAddress != address:
            toBeSentMessage = str(message_struct).encode()
            serverSocket.sendto(toBeSentMessage, address)
```

When the server receives a **logout** command from a client, the server will search the list of online users and remove that client.

```
# functionality for logging out a user
elif command == "LOGOUT":
    for client in clients:
        if clientSenderAddress == client["address"]:
            clients.remove(client)
```

When the server receives a SEND_MESSAGE (Chat message) command from a client together with the information like the message type, actual text message, receiving username from the sending client it will search for the user that's supposed to receive the message in the list of online users and send a message to that client.

```
# functionality for sending a message to a user
elif (command == "SEND_MESSAGE"):
    message_info = dictMessage["body"]
    sender_username = message_info["sender_username"]
    receiver_username = message_info["receiver_username"]
    text_to_be_sent = message_info["text"]

    # find user who is supposed to receive message
    for client in clients:
        if receiver_username == client["username"]:
            address = client["address"]
            message_struct = {
                    "header": {
                        "command": "",
                        "message_type": "MESSAGE_EXCHANGE",
                        "recipient_data": address,
                        "sender_data":  clientSenderAddress,
                    },
                    "body": "Private message from " + str(sender_username) + ": " + str(text_to_be_sent)
            }

            toBeSentMessage = str(message_struct).encode()
            serverSocket.sendto(toBeSentMessage, address)  # send welcome message to all other clients except the current c
```

When the server receives a SEND_ALL_MESSAGE (group chat message) command from a client, the server will send a text message to all clients in the online user list except for the sending client.

```
elif (command == "SEND_ALL_MESSAGE"):
    message_info = dictMessage["body"]
    sender_username = message_info["sender_username"]
    #receiver_username = message_info["receiver_username"]
    text_to_be_sent = message_info["text"]

    # find all users who are supposed to receive the message
    for client in clients:
        address = client["address"]

        if address != clientSenderAddress:
            message_struct = {
                "header": {
                    "command": "",
                    "message_type": "MESSAGE_EXCHANGE",
                    "recipient_data": address,
                    "sender_data":  clientSenderAddress,
                },
                "body": "Group Message from " + str(sender_username) + ": " + str(text_to_be_sent)
            }

            toBeSentMessage = str(message_struct).encode()
            serverSocket.sendto(toBeSentMessage, address)  # send welcome message to all other
```

When a client requests for all the online users, the server will receive a **contacts** command and will send a list of all online users to the requesting client excluding itself.

```
# functionality for listing all online users
elif command == "CONTACTS":
    for current_client in clients:
        if current_client["address"] != clientSenderAddress:
            message_struct = {
                "header": {
                    "command": "",
                    "message_type": "LIST_CONTACTS",
                },
                "body": "{0} : {1} {2}".format(current_client["username"], current_client["address"], "is online!")
            }

            toBeSentMessage = str(message_struct).encode()
            serverSocket.sendto(toBeSentMessage, clientSenderAddress)
```

Whenever a client receives a message, if the type of message is a CHAT message, then the client will send a message to the server to notify it of an acknowledgm ent message that should be sent to the client that sent the message. The server will search the list of all online users and once it finds the user it will send an acknowledgment message to that user.

```
# functionality for sending acknowledgement messages between users
elif command == "" and message_type == "ACKNOWLEDGEMENT":
    username = dictMessage["body"]["username"]
    text = dictMessage["body"]["text"]

    for client in clients:
        if client["username"] == username:
            print(client)
            print(client["address"])
            message_struct = {
                "header": {
                    "command": "",
                    "message_type": "ACKNOWLEDGEMENT",
                },
                "body": text
            }

            toBeSentMessage = str(message_struct).encode()
            serverSocket.sendto(toBeSentMessage, client["address"])  # send message to all other clients except
```

Features

Authentication

Whenever a user wants to make a connection with the server, the client must first log in, providing a username and password, the application will search for a user with that specific name and if a user already exists in the Database, the provided password will be hashed and compared with the password stored on the Database. In the instance where it's a new user, the provided password will be hashed and stored together with the username in the Database.

The auth.py file implements all the methods required t o do authentication like **hash_password** - for hashing passwords, **findUser** - searching for a user in the database, **insertUser** - for creating a new user. To hash passwords, we used Python's native library called hashlib and used a sha256 algorithm which outpu ts a value that is 256 long.
To store the users we used SQLite and created a User table to store all users.
This feature was included because we wanted to keep track of all the users that use the application. Here's an example of the feature below

```
dikatso@dikatso:~/Downloads/inst-main/instance-1$ python client.py
> login
enter username: MSHDIK008
User record doesn't exist
sign up ? Y/N... Y
enter password: NKNBOT001 - MKXLIB001
signing up...
loging in....
> Greetings from the Chat Room!
```

Group Chat

Group message was made possible by storing all online clients in an in-memory database (list) so whenever a user specifies sending a group chat message, we loop over that list containing the users and send a message to all of them one by one.

On the client-side:

```
elif command == "message*":
    text = " ".join(user_input[1:])

    message = {
    "sender_username": currently_logged_user["username"],
    "receiver_username": "",
    "text": text
    }
    message_to_all_users(message)
```

On the server-side:

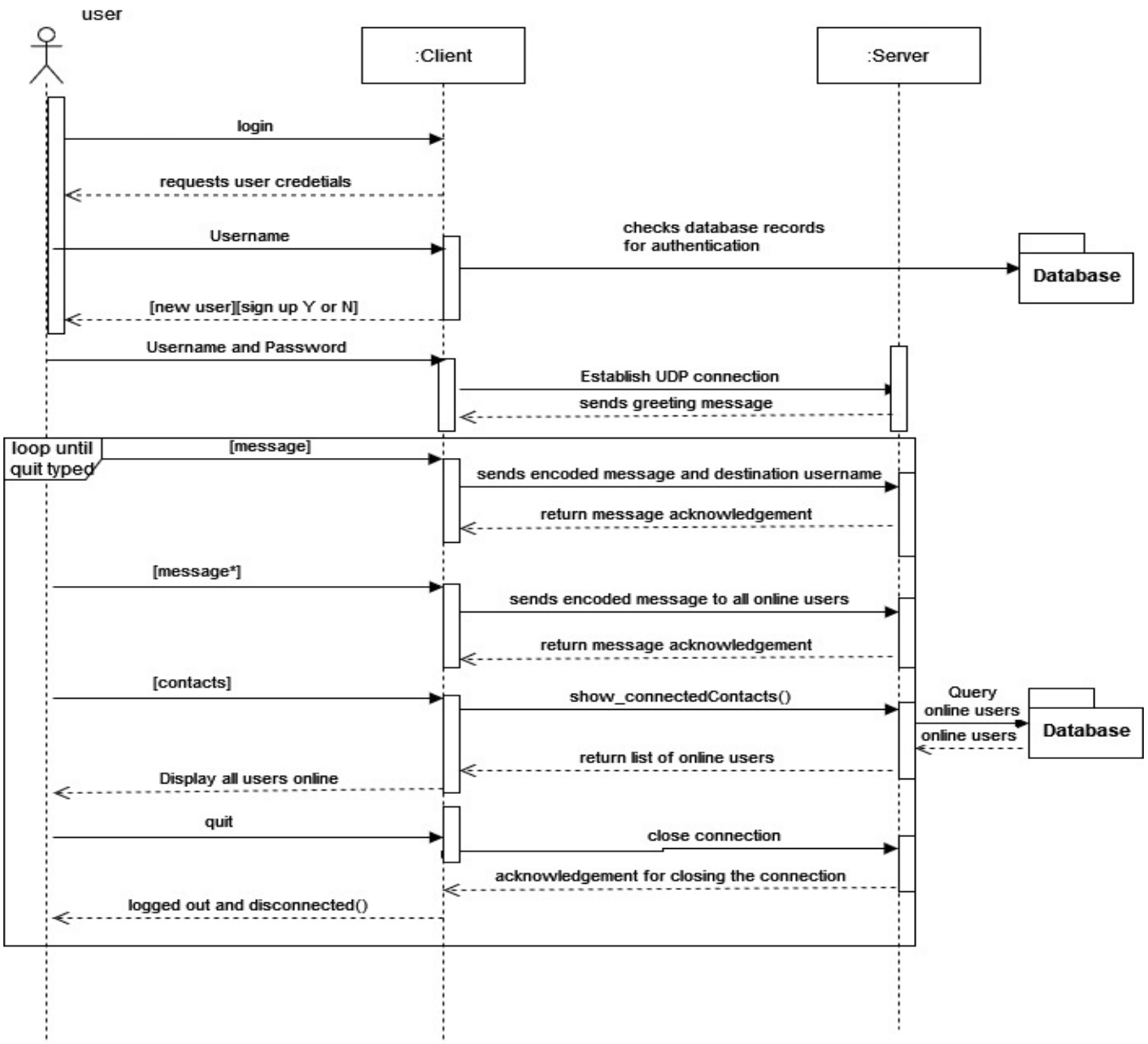
```
elif (command == "SEND_ALL_MESSAGE"):
    message_info = dictMessage["body"]
    sender_username = message_info["sender_username"]
    #receiver_username = message_info["receiver_username"]
    text_to_be_sent = message_info["text"]

    # find all users who are supposed to receive the message
    for client in clients:     Botshelo Nokoane, 3 days ago • implement group messaging …
        address = client["address"]

        if address != clientSenderAddress:
            message_struct = {
                "header": {
                    "command": "",
                    "message_type": "MESSAGE_EXCHANGE",
                    "recipient_data": address,
                    "sender_data":  clientSenderAddress,
                },
                "body": "Group Message from " + str(sender_username) + ": " + str(text_to_be_sent)
            }

            toBeSentMessage = str(message_struct).encode()
            serverSocket.sendto(toBeSentMessage, address)  # send welcome message to all other
```

This feature was included to allow group messaging for the users. Here's an example of the group feature below.

```
> Greetings from the Chat Room!
NKNBOT001 is online!
MKXLIB001 is online!
message* Here's a message from MSHDIK008 to NKNBOT001,MKXLIB001
> Message received by MKXLIB001
Message received by NKNBOT001
```

Sequence Diagram

Below is a diagram that shows the sequence and messages between servers and clients in different states, and reactions to the messages.



Conclusion

The application is working as expected. All the assignment requirements have been met. The UDP client server has been implemented in a manner that is more secure and reliable. All group members contributed equally towards the assignment.