MARMARA
UNIVERSITY

CSE4074 - Computer Networks

Fall 2021 - Project Report
January 11, 2022

Team Members:
150119870 - Ali Reza Ibrahimzada
150117018 - Ahmet Önkol
150117030 - Beyza Aydoğan

Instructor:
Dr. Ömer Korçak

Contents

Project Summary **2**

Methodology and Approach **2**

Challenges Encountered and Solutions **7**

Unresolved Problems **7**

Usage Explanation **8**

Protocol Description **9**

Developer Notes and Future Work **14**

Project Summary

In this project, we implemented a peer-to-peer (P2P) centralized chatting application. The project is composed of two main components. First, we created a centralized registry with a TCP and UDP socket. The main purpose of the registry is to store the clients information, provide services to clients and ease the communication between different users. Second, we also created a client module which can be run by the users in order to request services from the registry and to communicate with other online users. That is, for group chatting purposes, the client messages are routed through the server, but for private chatting, the clients message is routed to other peers directly through their TCP server. The project requirements demand the implementation of three main services in the chatting application, namely (i) Join: a service which allows new clients to join the chatting application, (ii) Search: a service which allows clients to search for other fellow clients, and (iii) Chat: a service which provides chatting through text between clients. We have defined and determined our own protocols in this project in order to accomplish the required tasks. Moreover, our implementation also **provides group chatting** between multiple clients. Although group chatting is not mandatory for the purpose of this chatting application, we decided to do it for the bonus of the project. Furthermore, in order to improve the user experience and decrease the waiting times between messages, our application leverages multithreading both on server side and on client side. The details of each running thread will be given in subsequent sections of the report. Moreover, we have used Python as our main programming language in this project.

Methodology and Approach

Our approach for building the chatting application involves two main modules. As discussed earlier, the server module is responsible for providing services to clients, and to forward their group messages. Next, we also created a client module which can be executed multiple times by different users. Moreover, we used Python's *socket* library for creating sockets both on the server and clients sides. For instance, in order to create a TCP socket using this library, we simply created an instance like the one below:

```
tcp_socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
```

Here, we have determined the Address Family (AF) of the socket as *AF_INET* (Address Family Internet), and its type as *SOCK_STREAM* which simply means the TCP socket. Moreover, in order to avoid unexpected exceptions when running the program, we decided to use the following:

```
tcp_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Here, the *setsockopt* method will set new options for the *tcp_socket*. The first argument is *SOL_SOCKET*, which is the socket layer itself. It is used for options that are protocol independent. The second argument is the option name we want to set and the third one is the value for that option, which is *SO_REUSEADDR=1*. This flag tells the kernel to reuse a local socket in *TIME_WAIT* state, without waiting for its natural timeout to expire. This approach avoids runtime exceptions like address/port not available. Later on, we use the *bind* method

and connect the socket with its corresponding IP address and port number. Moreover, we also invoke the *listen* method of the TCP socket so it could start listening for incoming packets. The *MAX_CONNECTIONS* is an integer which determines the maximum number of clients this socket can serve at one time. At this point, all necessary initializations have been finished and the server is ready to provide services. Please note that UDP socket is initialized in the same way and it has been avoided in the report to remove duplicates.

```
tcp_socket.bind((IP, TCP_PORT))

tcp_socket.listen(MAX_CONNECTIONS)
```

The server module holds a series of data structures (i.e. lists, dictionaries, etc.) to store client related data at runtime. All information stored on the server is volatile and the data will be removed once the server process dies. On the other hand, we have also written some commonly used routines in the server module which are being used several times throughout the program. For instance, a routine called *get_socket(username)* will iterate over all available clients and look for a match with the given username. It will return the socket instance of the corresponding username if there is any client with that username, else it will return *None*. From here on, the responsibilities of the server module is split into three distinct threads so that the users could get concurrent services and improved experience. Figure 1 shows the overall structure of the server module in a block diagram.

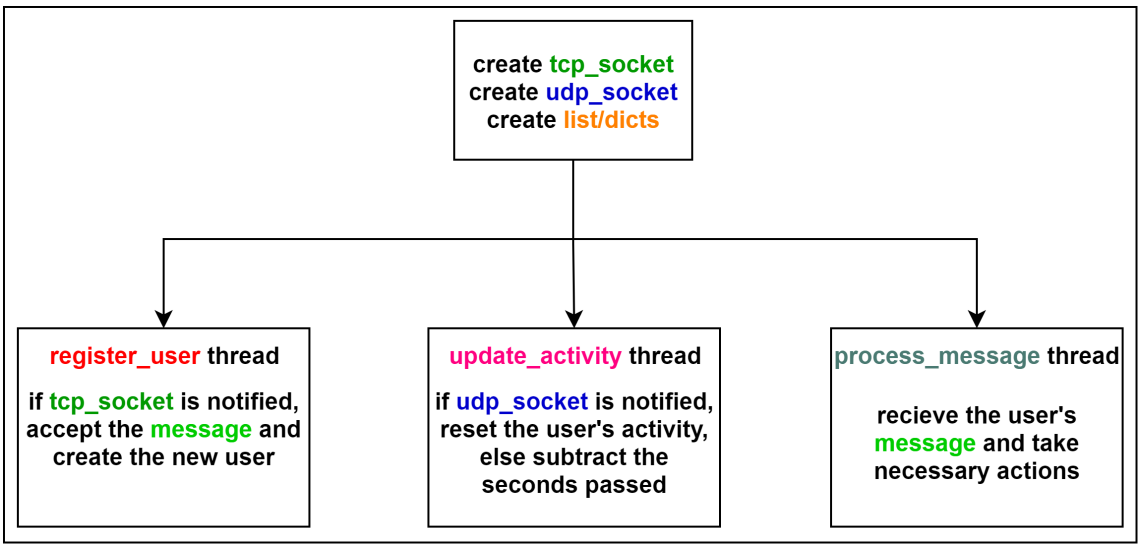

Figure 1: A block diagram explaining the *Server* process

For creating threads in Python, we have used the *threading* library which is included in Python's standard library. This library allows us to create multiple threads and execute them in parallel. Please find a sample thread initialization below:

```
update_activity_thread = threading.Thread(target=update_activity)

update_activity_thread.start()
```

Here we give a target function to the thread so it could only execute that until the process (mother thread) dies. After a thread is initialized, we invoke the *start* method so it could start

running. As it can be seen from Figure 1, there are three main threads running in our server process. Next, we will discuss the characteristics of each thread.

1. **Register User Thread:** In this thread, we use the *select* library from the Python's standard library to gain OS level IO access. That is, it takes a list of file descriptors and it waits until it is ready to be read, or in other words, if any packets have sent/arrived to those lists, then the select method will return those sockets only. Please refer to the following line below:

   ```
   read_sockets, useless, exceptional_sockets = select.select(sockets,
   [], sockets, 0)
   ```

   In the above method call, the first argument is the list of all sockets available in our program which we want to be notified if something came through, the second one (empty list in our case) is the list of objects we want to write, the third one is the list of exceptional sockets (i.e. unexpectedly closed, etc.), and the final argument is the amount of time in seconds the method will wait until at least one socket sent/received something. The reason we set the waiting time to 0 is that we don't want that method call to block the whole thread. Please also note that the same usage of *select* library has been implemented in all other threads.

   After the TCP socket has something to be read, we retrieve the message sent to this socket and extract the new user data from it. It can be simply done by calling the *accept* method of the TCP socket on the server side. The returned value is a tuple, which contains the client socket and its address/port number.

   ```
   client_socket, client_address = tcp_socket.accept()
   ```

   Then, the details of the client are stored into relevant lists and dictionaries. This whole process is repeated in an infinite loop which makes it possible for new clients to enter the chatting application at any time.

2. **Update Activity Thread:** The update activity thread is very similar to the previous thread, but the only difference is that we use a UDP socket for the purpose of updating the user's activity. More specifically, if the UDP socket is not in the list of returned sockets by the *select* library, we iterate over each client and reduce its activity by the amount of time passed since it was last updated. After updating, if the activity time of a client is <= 0, we remove that client from all relevant data structures on the server side. On the other hand, if the UDP socket is in the list of returned sockets, we use the following line to retrieve the message sent to the UDP socket.

   ```
   message = udp_socket.recvfrom(256)
   ```

   The *recvfrom* gets a buffer size and reads that many bytes from the incoming message. Since the HELLO message along with its HEADER and client username cannot exceed 256 bytes, we decided to give 256 for our buffer size. The server will use the client username sent together with a HELLO message in order to reset the activity

time of that client. Please note that a HELLO message is sent by each client through the UDP socket every 60 seconds. Moreover, if the server did not receive a HELLO message from its client for at most 200 seconds, it will preempt them from the system.

3. **Process Message Thread:** This thread also runs an infinite loop, and waits for at least one socket to be returned by the *select* library. If a socket is returned, it retrieves the message sent by that socket and then decodes it. Based on the content of the message, the thread takes a necessary action. Please refer to Protocol Description for a detailed explanation on different types of messages.

   Moreover, we would like to discuss our approach for handling messaging and message forwarding. Generally speaking, we have implemented a pool analogy for solving the group chat problem. That is, if three or more clients are having a group chat, then we put all of them in a pool and forward the messages sent by any client to all clients in that specific pool. We used dictionaries to store pools, such that pool id is the key and the value is a list of all participant socket instances available in that pool.

Next, we would like to discuss the client side of our chatting application. It is important to note that all initialization steps are very similar to that of the server. Therefore, we will jump right into the structure of the client module. Figure 2 represents the overall structure of the client module in our application.

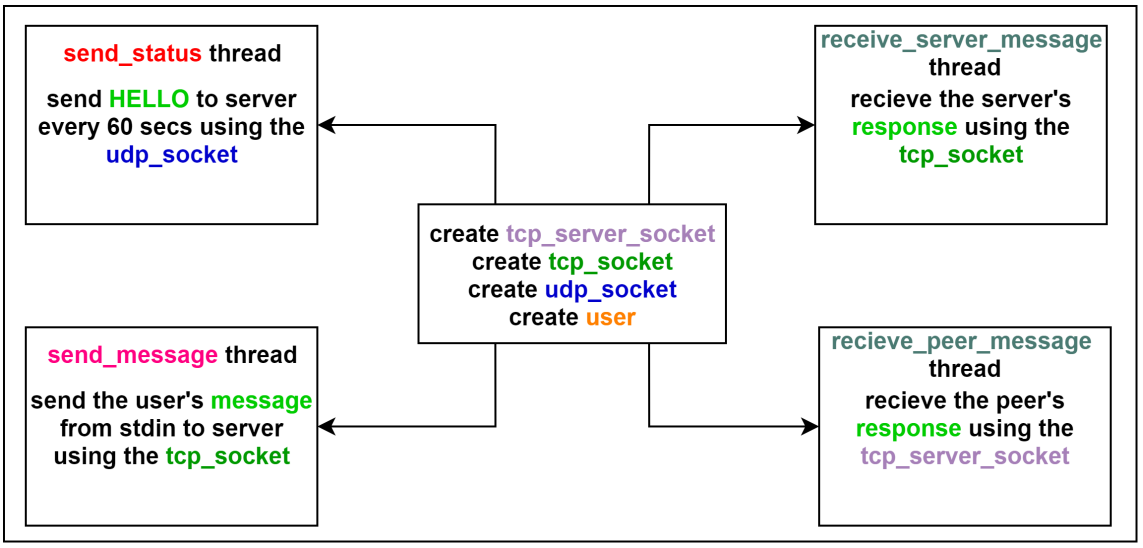

Figure 2: A block diagram explaining the *Client* process

1. **Send Status Thread:** The send status thread is one of the independent tasks a client process should run as long as the user is online. To begin with, we first create a time instance, and then in an infinite loop we keep creating another time instance and consistently keep checking the difference between time instances. Then, if the difference is >= 60, we use the UDP socket and send a HELLO message along with the username of the client to the server. The following line in our code sends the message to the UDP socket of the server.

```
udp_client_socket.sendto(message_header + client_message, (IP,
UDP_PORT))
```

Here, the message header contains the message length, and the client message contains the message HELLO along with the username. The *sendto* method takes the address and port number as its second argument.

2. **Send Message Thread:** The send message thread is mainly responsible for getting inputs from the user using the Python's *input()* instance. The thread will wait until the user presses the ENTER button, and then decide on the message content based on the user's input. For instance, if the user has written SEARCH or CHAT REQUEST, then the thread will make sure that the correct protocol is used and the relevant message is sent to the server. Please refer to the Protocol Description section for a detailed explanation on each protocol used by the client module. Please remember that the input taking process runs in an infinite loop, and it will only stop when the user logs out from the system. Moreover, there can be two candidate recipients for a user message. First, a user can send a normal message in a private chat session to another client's TCP server. The purpose of this is to make sure that a P2P approach has been implemented and that the private messages are not sent through the registry. Second, another possible recipient for a client message is the registry. For instance, messages like SEARCH, CHAT REQUEST, and group chat messages are routed through the server.

3. **Receive Server Message Thread:** The receive server message thread is mainly responsible for receiving the server's response to a query a client already sent to the server. Such server responses include LOGOUT SUCCESS, CLIENT OK, CLIENT EXIT, etc. For instance, the LOGOUT SUCCESS message is sent from the server to the client who already sent the server the LOGOUT message. The client will then take necessary actions based on the server response.

4. **Receive Peer Message Thread:** The receive peer message thread is created for receiving and printing private text messages which are sent by another peer. The TCP server socket of a client consistently listens for new peer messages, and it will print them once a valid client sends a message.

Please note that some threads in the client module make use of a lock to synchronize reading/writing from/to a specific data structure. Without using a lock, there is a possibility that OS changes between threads and an inconsistency can appear as a consequence of this action.

Next, we would like to discuss the format of our messages which are sent between clients and the server. Figure 3 shows an overall message format we have adapted for all our protocols in this project.

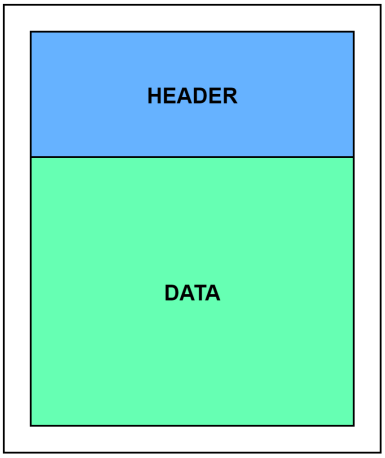

Figure 3: A typical message format used in this project

As shown in Figure 3, the HEADER section of our messages carries the length of the DATA section in terms of number of characters. Such an approach is necessary in making the sockets receive a limited chunk of data (buffer size), and do not waste CPU cycles waiting for something which might not come at all. Please refer to the Protocol Description section for a detailed explanation on how HEADER and DATA sections of a message are combined.

Challenges Encountered and Solutions

In this section, we will discuss the problems we encountered in this project and the solutions we come up with to address those problems. Please find the list of challenges and their solutions below:

- **Blocking Input:** Initially, we wrote the client module without any threading. We were mainly doing three distinct tasks on the client side, namely (i) update user status, (ii) send messages, and (iii) receive messages. The problem arises when the user wants to send a message. That is, when writing a message, the process stops until the user presses the ENTER button. For instance, if a user were to receive a message while s/he is writing, they would not be able to see it until they are done writing. Such an issue brings a lot of inconsistencies and decreases the user experience. We addressed this problem by creating three threads in order to serve the three distinct tasks in the client module. This way, the sending message thread would no longer block receiving messages and updating status routines.

- **Synchronization:** As our threads are running concurrently and we have some variables which are read/written by two or more threads, there is a chance for inconsistency in data at runtime. We have used mutual exclusion (mutex) locks in order to find a solution for this problem.

Unresolved Problems

Our implementation has addressed all of the necessary requirements explained in the project. To the best of our knowledge, we do not think there are any unresolved problems in our

project. We have rigorously tested our project and handled all exceptions we have witnessed. Nevertheless, there might be some edge cases where exceptions could still happen. Below are some of the edge cases we would like to mention in our report:

- If a user with a specific username does not exist in the chatting application, sending a CHAT REQUEST or adding them to a group chat session using GROUP CHAT will cause a problem. The user should first make sure that a specific user exists by using the SEARCH command.

Usage Explanation

In this section, we shall explain the most important and major usages of our chatting application. Please refer to image captions for a better understanding of the image. In the images, the top left terminal is always the server process.

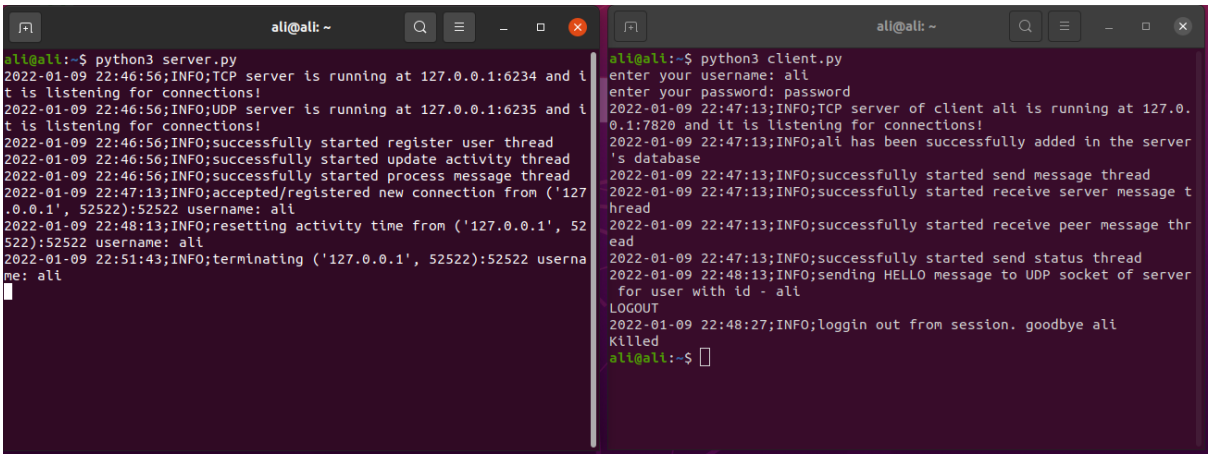

Figure 4: A demo for JOIN and LOGOUT

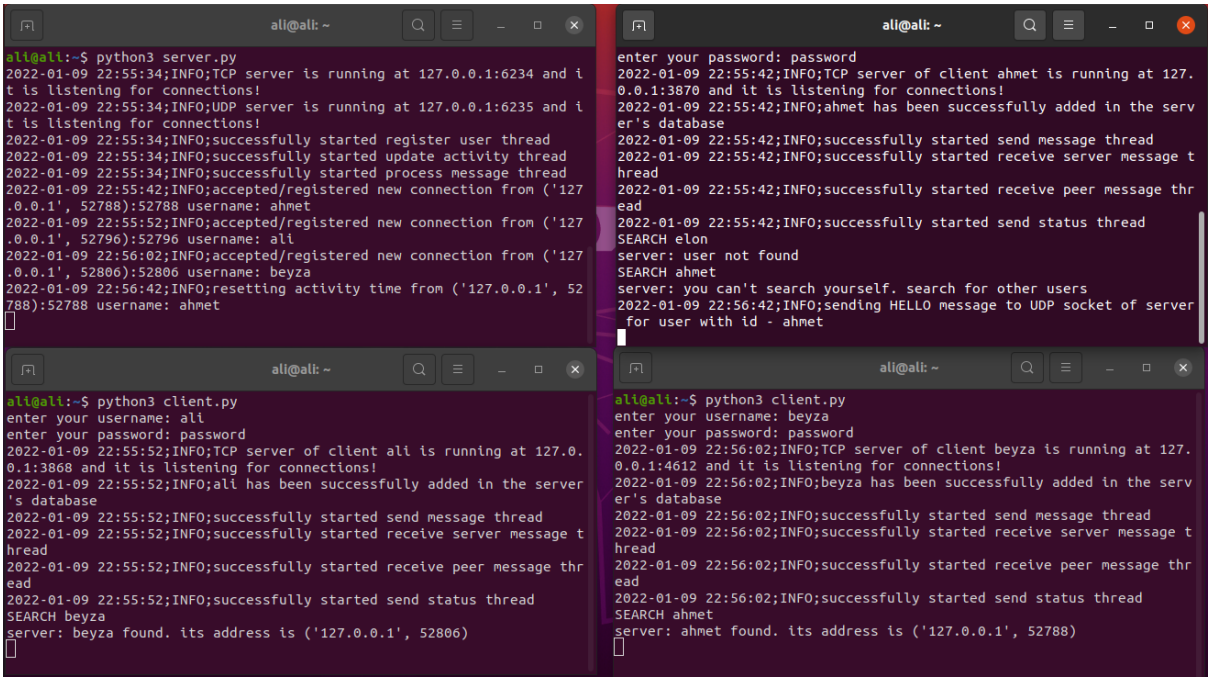

Figure 5: A demo for SEARCH

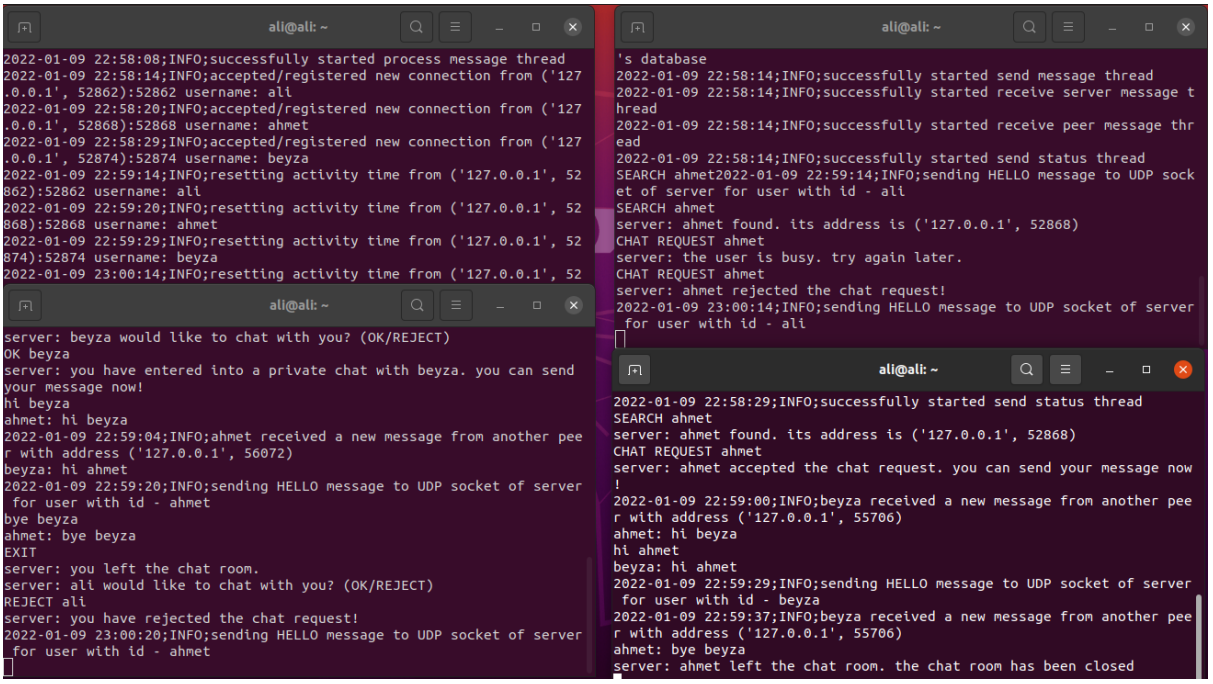

Figure 6: A demo for CHAT REQUEST, OK, REJECT, and CHAT

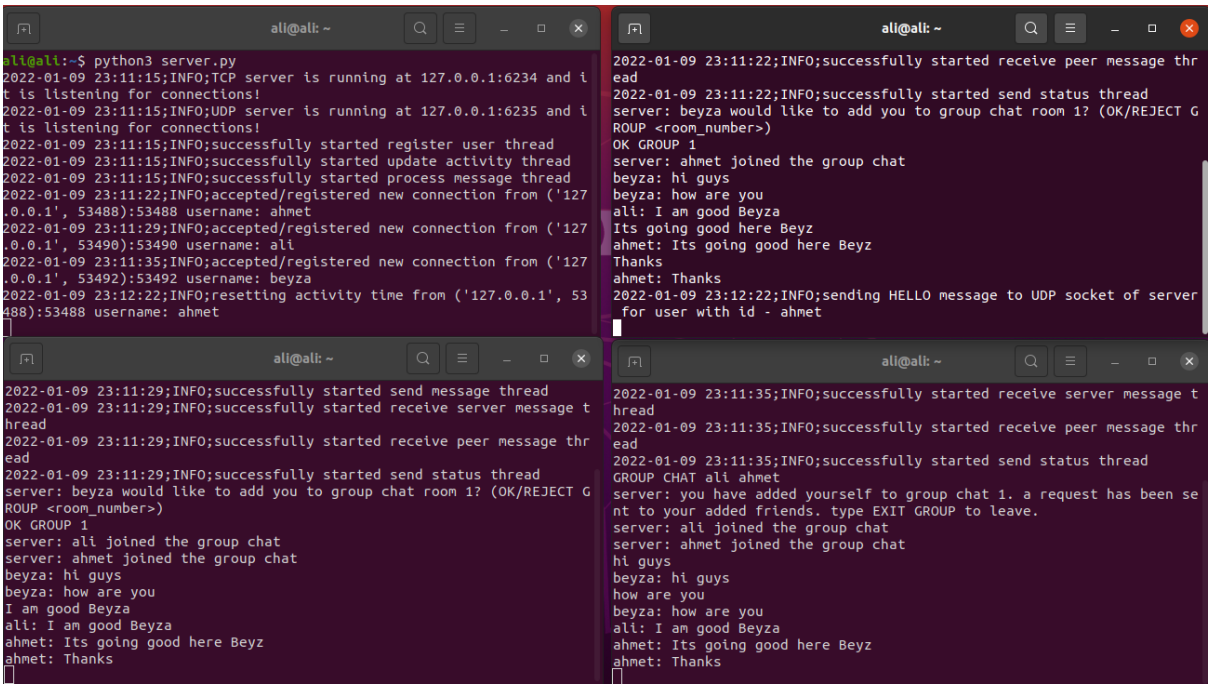

Figure 7: A demo for GROUP CHAT

Protocol Description

For better performance and simplicity, we have defined and determined our own protocols in this project. We believe that our defined protocols are enough for the purposes of this project. However, as the size and requirements of the project grows in future, we shall make necessary changes in order to adapt for new requirements. Please find the list of all available protocols used in this project below:

- **'HEADER&&SEARCH&&|searched_peer|client_username'**

  This protocol is used by the client to deliver a *SEARCH* message to the server. The client module expects the user to write the search message in the format *'SEARCH x'*, *x* being the username of the desired peer. Once a thread in the client process receives such an input from the user, it creates a string by combining the keyword *&&SEARCH&&, searched_peer* and the *client_username* as shown above. Please note that the *client_username* is the username of the client making the search. Later the length of the combined string is embedded in the *HEADER* section of the message and it is sent to the registry using the client's TCP socket.

- **'HEADER&&LOGOUT&&'**

  LOGOUT protocol is part of the *'Join'* service of the project. It is used by the client when the user wants to leave the current terminal session and go offline. The usage format is '*LOGOUT*'. If the user wants to logout, he/she shouldn't be part of an ongoing conversation whether it is a private chat or group chat. In other words, we are expecting the user to leave the chat room, then he/she will be able to logout from the current session. When sending the message on the client side, we are encrypting the LOGOUT message as '*&&LOGOUT&&*' and catching it from the server side as we process messages. We also check whether the user is in a session or not in the processing phase. When all requirements are met, we send a LOGOUT SUCCESS message to the client letting them know that they have successfully logged out from the chatting application. Afterwards, we send a SIGKILL signal to the OS from that process/thread, and then we inform the user with the following alert: *loggin out from session. goodbye <username>*

- **'HEADER&&GROUPCHAT&&|client_username|participant1|...|participantN'**

  This protocol is created for allowing users to group chat with one or more peers. The client module expects the user to write the group chat request in the format '***GROUP CHAT peer1 peer2 … peerN***', peerX being the username of the desired peer. Once a thread in the client process receives such an input from the user, it creates a string by combining the keyword *&&GROUPCHAT&&*, client_username, and usernames of the requested peers as shown in the above title. "client_username" is the name of the client who is sending that group chat request. The group chat request is sent to the peers whose names are listed in the string. Then OK/REJECT/BUSY responses return from the peers, requested client and approved peers start group chatting.

- **'HEADER&&REJECTGROUP&&|client_username|group_number'**

  This protocol is created for replying as a reject to the group chat request. A client sends a group chat request to other clients as explained above. Requested peers are asked '*server: <**client_name**> would like to add you to group chat room <**room_number**>? (OK/REJECT GROUP <**room_number**>)*'. Then the client module expects the peer to write '*REJECT GROUP <**room_number**>*'. The client is

sending this message as *'&&REJECTGROUP&&|{client_username}|{group_number}'* to the server. In this message *client_username* is the username of the peer who rejected the group chat request. After getting a REJECT response, the server sends '*server: you have rejected the group chat request!*' message to that peer. Also the server sends '*server: <**peer_name**> rejected the group chat request!*' message to the requested client.

- **'HEADER&&OKGROUP&&|client_username|group_number'**

  This protocol is created for accepting the group chat request. A client sends a group chat request. Requested peers are asked '*server: **client_name** would like to add you to group chat room <**room_number**>? (OK/REJECT GROUP <**room_number**>)*'. Then the client module expects the peer to write '*OK GROUP <**room_number**>'*. The client is sending this message as '*&&OKGROUP&&|{client_username}|{group_number}*' to the server. In this message *client_username* is the username of the peer who accepted the group chat request. After getting an OK response, the server sends '*server: **<peer_name>** joined the group chat*' to all members of that group chat.

- **'HEADER&&CHATREQUEST&&|searched_peer|client_username'**

  CHAT REQUEST protocol is used when the user wants to contact a peer user for a private P2P chat. It belongs to the 'Chat' part of the project. The protocol's goal is to establish peer to peer chat, private chat room in other words, with desired user. Usage of it is 'CHAT REQUEST <username>', where username corresponds to the desired user's username. We are sending message as *'&&CHATREQUEST&&|{searched_peer}|{client_username}'* where searched_peer corresponds to the username that the current user wants to connect with, and client_username corresponds to the current user's username. On the server side,when we are processing messages, we are holding peer in *searched_peer* and sender in *sender_username*. The socket of the searched peer is assigned to *client_socket* and sender socket is assigned to *sender_socket*. Then we check whether the requested user is busy or not by traversing private and public chat room lists. If the user is in a chat room we give the following alert *'the user is busy. try again later.'*. On the other hand, if the peer user is available we are asking the following question *'{sender_username} would like to chat with you? (OK/REJECT)'* where sender_username corresponds to the user that sends the private chatting request.

- **'HEADER&&REJECT&&|client_username|sender_username'**

  REJECT protocol is part of the P2P (Peer-to-Peer) chatting process. It is used when the peer user wants to reject the chat request. Usage of the protocol is 'REJECT <sender_name>' where sender_name corresponds to the sender of the chat request. Client sends *'&&REJECT&&|{client_username}|{sender_username}'* as a message where client_username is the user who receives a private chat request and sender_username is the user that requests private chat. On the server side, when we

are processing the messages, we are sending two seperate messages. Both messages are sent by the server and one of them is *'server: you have rejected the chat request!'* which is sent to the peer user that receives the chat request. The other message is sent to the user that sends the chat request and it's content is : *'server: <peer_username>rejected the chat request!'* where peer_username corresponds to the user that receives the chat request.

- **'HEADER&&OK&&|client_username|sender_username|tcp_server_port'**

  OK protocol is part of the P2P (Peer-to-Peer) chatting process. It is used when the peer user wants to accept the chat request. Usage of the protocol is 'OK <sender_username>' where sender_username corresponds to the sender of the chat request. Client sends *'&&OK&&|{client_username}|{sender_username}|{tcp_server_port}'* as a message where client_username is the user who receives a private chat request, sender_username is the user that requests private chat, and tcp server port is the port number for the TCP server socket of the client. On the server side, when we are processing the messages, we are sending two seperate messages. Both messages are sent by the server and one of them is *'server: you have entered into a private chat with <sender_username>. you can send your message now!'* where sender_username is the username of the user that sends the chat request. This message is sent to the peer user that receives the chat request. The other message is sent to the user that wants to chat and it's content is : *'server: <peer_username> accepted the chat request. you can send your message now!'* where peer_username corresponds to the user that receives the chat request. We keep track of the number of private rooms in the *total_private_rooms* variable which is used to add current peers socket into *private_chat_rooms* list. Corresponding users' session information is also updated by setting in_session data into true. So in further chat requests, we can check the availability of them.

- **'HEADER&&EXIT&&|client_username'**

  This protocol is used by a client whenever they want to leave a private chat session. When the client wants to leave a private session, they should write *'EXIT'* and then the chatting application will remove all members (a total of two since it is P2P) from the chat room. A CLIENT EXIT message will be then sent by the server to the client making the EXIT. After receiving the server's response, the client will acquire the mutex lock and clear out the data structure which holds the peers data.

- **'HEADER&&EXITGROUP&&|client_username'**

  This protocol is very similar to the EXIT protocol explained above. The only difference is that the chat room is not destroyed after a peer decides to leave the conversation. Only the peer making the EXIT GROUP message will be removed from the group chat room and all members of the room will be notified about his/her leave.

If a client wants to leave the group chat room, they are expected to write *'EXIT GROUP'*.

- **'HEADER&&MESSAGE&&|client_message'**

  This protocol is used by the client to send their chat message (private/group) to other peer(s). If a client is in a private session, the message is sent to the TCP server socket of another client in order to follow the P2P approach. On the other hand, if the client is in a group chat session, the message is routed through the registry to other peers.

- **'HEADER&&LOGOUTSUCCESS&&'**

  This protocol is used by the server to give a response to the user's LOGOUT message. It is sent implicitly by the server after it makes sure that the user can log out successfully.

- **'HEADER&&HELLO&&|client_username'**

  This protocol is used by the client module to send a HELLO message every 60 seconds to the UDP socket of the registry. The HELLO message is sent to the server in an implicit way without the knowledge of the user as long as it is online. The server will then retrieve the client username from the message and reset the activity time (i.e. set to 200 seconds) for that client.

- **'HEADER&&REGISTER&&|client_username|tcp_server_port'**

  This protocol is used by the client module to register a newly signed-in client. After a user enters their username and password, the client module uses this protocol and sends the information embedded in the message to the TCP socket of the server. The server will then retrieve the username and other important attributes from the message and store the client socket in the clients dictionary.

- **'HEADER&&CLIENTOK&&|message_body|tcp_server_port'**

  This protocol is used by the server to send a response to both sender and receiver of a chat request message after the receiver accepts to chat with the chat request sender. The *message_body* will contain an informative message about the peer, and the *tcp_server_port* contains the server port of the client. That is, the chat request sender will get the TCP server port of the receiver, and the chat request receiver will get the TCP server port of the sender. Then, the clients will create a new socket and bind it to the new address (IP, TCP server port). Please note that IP is always 127.0.0.1 since we are only working on a localhost network.

- **'HEADER&&CLIENTEXIT&&|message_body'**

  This protocol is yet another protocol used by the server as a response to a user who is sending the EXIT message whenever they want to leave a private chat session. This

protocol is used to send an implicit message to the client. The *message_body* contains an informative message about the situation.

- **'HEADER&&FOUND&&|message_body'**

  This protocol is used by the server to send a response to the user who already sent a SEARCH message. If a client with the given peer name is available, the server will use this protocol to send a response to the client. The *message_body* contains an informative text about the situation.

- **'HEADER&&NOTFOUND&&|message_body'**

  This protocol is used by the server to send a response to the user who already sent a SEARCH message. If a client with the given peer name is not available, the server will use this protocol to send a response to the client. The *message_body* contains an informative text about the situation.

- **'HEADER&&BUSY&&|message_body'**

  This protocol is used by the server to send a response to the user who already sent a SEARCH message. If a client with the given peer name is already in another room, the server will use this protocol to send a response to the client. The *message_body* contains an informative text about the situation.

- **'HEADER&&INVALIDSEARCH&&|message_body'**

  This protocol is used by the server to send a response to the user who already sent a SEARCH message. If a client with the given peer name is the username of the client making the SEARCH, the server will use this protocol to send a response to the client. The *message_body* contains an informative text about the situation.

Developer Notes and Future Work

We would like to emphasise the importance of reading the project report prior to using it. The program does not necessarily cover edge cases (i.e., a junk input where the program is not expecting it). We recommend reading and understanding all of the protocols in this report prior to testing the chatting application. Finally, we would like to thank the users of our chatting application. In case of any problems, please feel free to create an issue on our github repository. Furthermore, as part of our future work, we plan to develop a Graphical User Interface (GUI) for our chatting application and improve our protocols for better communication.