CSE 4074

COMPUTER NETWORK

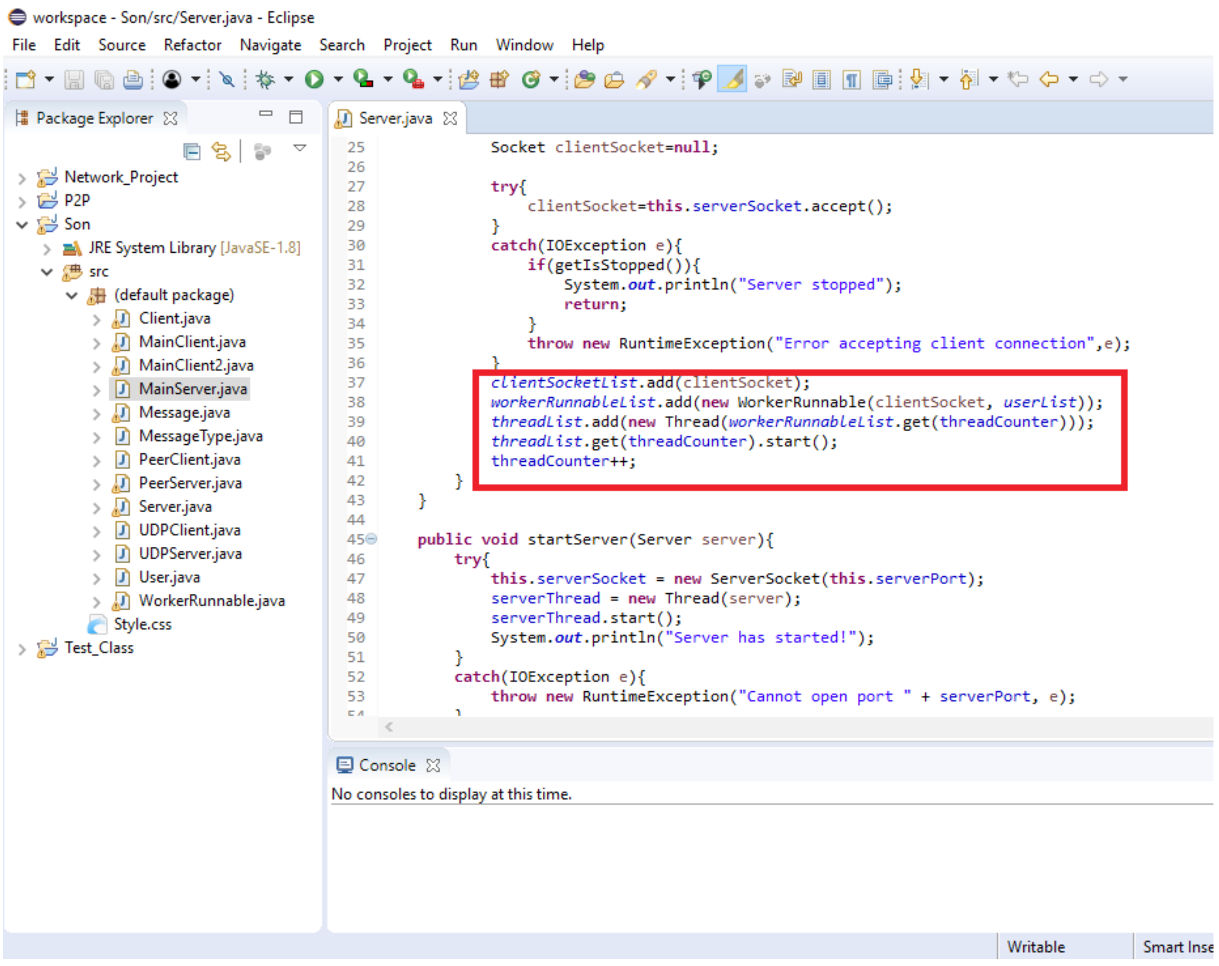
PROJECT REPORT

<u>NOTE:</u> In order to run this program, you should compile/run MainServer.java and MainClient.java. Since peerServer has a constant port, you should open clients in different computers.

<u>Project Summary:</u>

First of all, this project is consisting of several different components. To begin with, our chatting program provides a multi-threaded server and serves incoming clients. Every time a new client tries to connect a server, the registry server starts a new server thread with corresponding socket to communicate with the client. In addition, the registry thread itself is a thread. This registy server operation is handled by **Server.java** in our project. Registry server can be accessed by clients on port number 9000.

Here you can clearly see that registry server is creating new worker threads.



```java
            Socket clientSocket=null;

            try{
                clientSocket=this.serverSocket.accept();
            }
            catch(IOException e){
                if(getIsStopped()){
                    System.out.println("Server stopped");
                    return;
                }
                throw new RuntimeException("Error accepting client connection",e);
            }
            clientSocketList.add(clientSocket);
            workerRunnableList.add(new WorkerRunnable(clientSocket, userList));
            threadList.add(new Thread(workerRunnableList.get(threadCounter)));
            threadList.get(threadCounter).start();
            threadCounter++;
        }
    }

    public void startServer(Server server){
        try{
            this.serverSocket = new ServerSocket(this.serverPort);
            serverThread = new Thread(server);
            serverThread.start();
            System.out.println("Server has started!");
        }
        catch(IOException e){
            throw new RuntimeException("Cannot open port " + serverPort, e);
```

```
No consoles to display at this time.
```

In **WorkerRunnable.java** class, the major operation of the server is done. The worker thread starts listen to port and accepts the particular client's connection request and adds information of user such as username, password and IP to the user list. It also checks user list depending on the client actions, checks username similarity and in case of sign in or sign up, it changes user's connection state.

We also have a decent GUI interface and by executing **MainClient.java**, also its derivatives and **MainServer.java**, user can easily use the program. MainServer.java includes graphical user interface for the owner of the server and also the same for the client, MainClient.java.

Client.java is basically a constructor of client. It has parameters of username, password, server's IP and port. It responsibles for sending its info message to the registry server.

In **Message.java**, this is the class what we are sending and it has different types. For type selection, **MessageType.java** is used. Client or server can send different type of messages in order to make server/client understand what is used for. For instance, it can pick chatRequest for chatting request message.

In **PeerClient.java** and **PeerServer.java**, each client has both of these classes. PeerClient.java works for sending messages to its corresponding peer's server part. Likewise, other side of the peer-to-peer connection can send message over PeerClient class and receives messages over its PeerServer class.

User.java has get methods of user's information and sets user is online now.

UDPAcceptingServer.java listens at a constant port number which is 65507. When Client sends connection request, it creates a new UDPPortFinder thread with parameters of client's socket, time out value in milliseconds and the list of users.

UDPClient.java sends "HELLO" message to the UDP Server. It creates a suitable socket that received from UDPPortFinder.

UDPSender.java receives HELLO messages and checks time if it is passed or acceptable. Regarding that, it can update user list with new states of online/offline.

Solution Approach and Problems We Encountered:

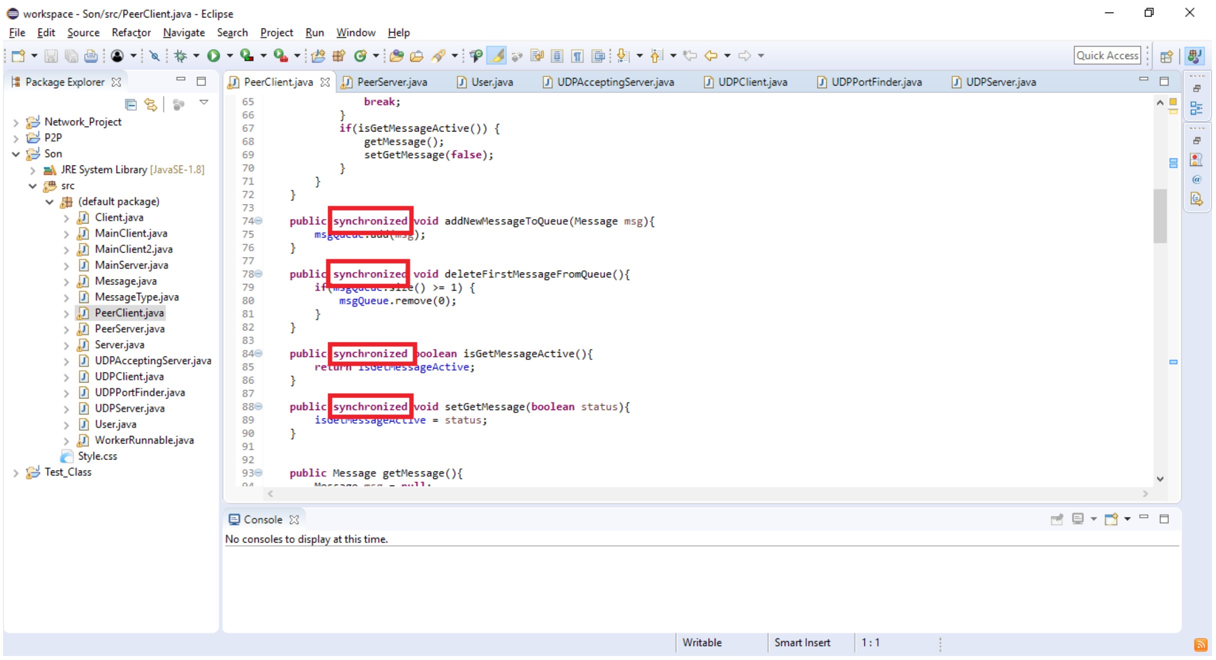
Most of the time, we have tried to solve problems with the help of multi-threading. Registry server with multi-threaded approach leads us solving many confusing problems like more than one client, request or any of incoming type of data. It was the best way to do so. Buffers for the messages were always useful and prevented data loss.

Majority of the problems were synchronization and race conditions. Because we were dealing with threads and normally their nature of difficult handling. Furthermore, finding the port numbers out were also one of the major problems we have encountered. We have discovered operating system was already using some of them.

To solve those problems, we took care of synchronization problem first. Java is really good at dealing with race conditons and synchronization. There were two options, one of them was making methods inaccessible by other threads at the same time. We have used synchronized key and also there were lock() and unlock() methods for the same purpose. However, we used synchronized keyword all the time.

As we mentioned before, the other major problem was finding out which ports were in usage and which ports do not. If we pass to the constuructors port number “0”, operating system takes care of automatically and allocates a port for our program.

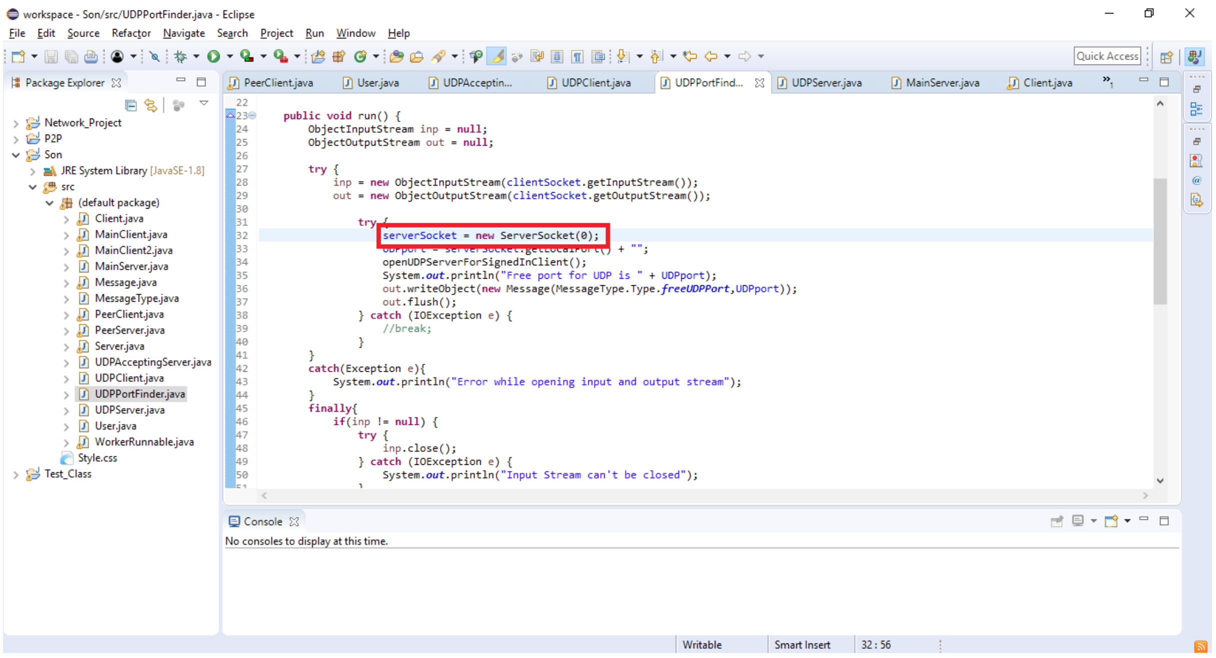
Screenshots are provided to demostrate how we used all of these.



workspace - Son/src/PeerClient.java - Eclipse

```java
                break;
            }
            if(isGetMessageActive()) {
                getMessage();
                setGetMessage(false);
            }
        }
    }

    public synchronized void addNewMessageToQueue(Message msg){
        msgQueue.add(msg);
    }

    public synchronized void deleteFirstMessageFromQueue(){
        if(msgQueue.size() >= 1) {
            msgQueue.remove(0);
        }
    }

    public synchronized boolean isGetMessageActive(){
        return isGetMessageActive;
    }

    public synchronized void setGetMessage(boolean status){
        isGetMessageActive = status;
    }


    public Message getMessage(){
```

```
No consoles to display at this time.
```



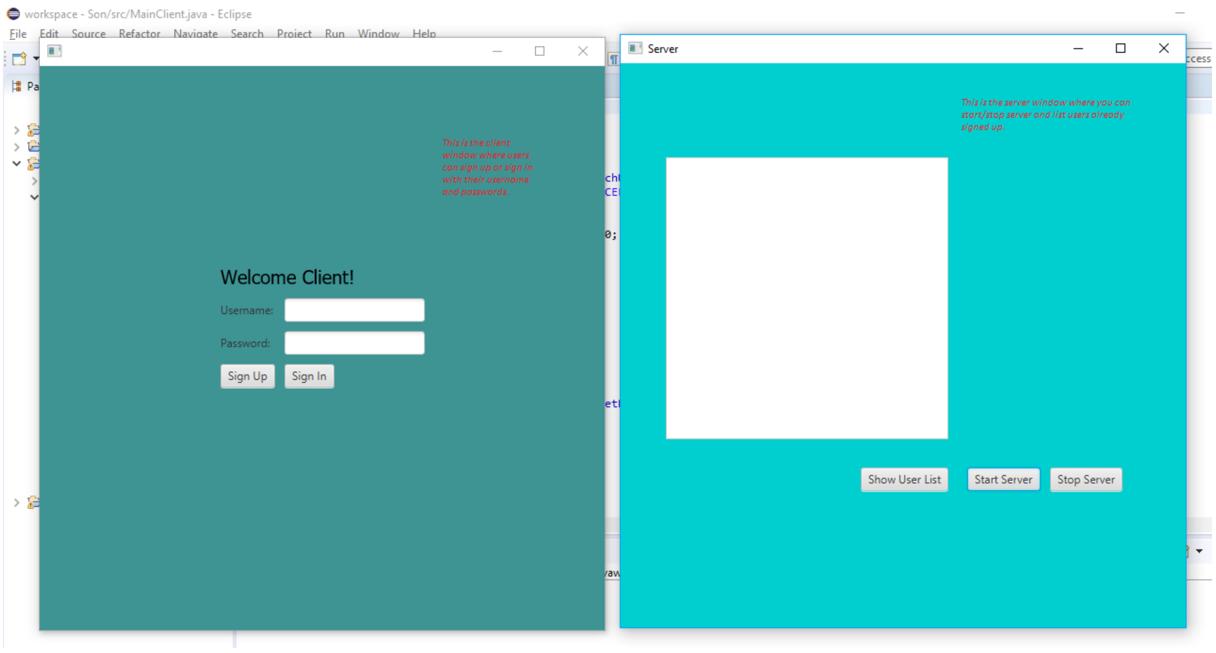
workspace - Son/src/UDPPortFinder.java - Eclipse

```java

    public void run() {
        ObjectInputStream inp = null;
        ObjectOutputStream out = null;

        try {
            inp = new ObjectInputStream(clientSocket.getInputStream());
            out = new ObjectOutputStream(clientSocket.getOutputStream());

            try {
                serverSocket = new ServerSocket(0);
                UDPport = serverSocket.getLocalPort() + "";
                openUDPServerForSignedInClient();
                System.out.println("Free port for UDP is " + UDPport);
                out.writeObject(new Message(MessageType.Type.freeUDPPort,UDPport));
                out.flush();
            } catch (IOException e) {
                //break;
            }
        }
        catch(Exception e){
            System.out.println("Error while opening input and output stream");
        }
        finally{
            if(inp != null) {
                try {
                    inp.close();
                } catch (IOException e) {
                    System.out.println("Input Stream can't be closed");
```

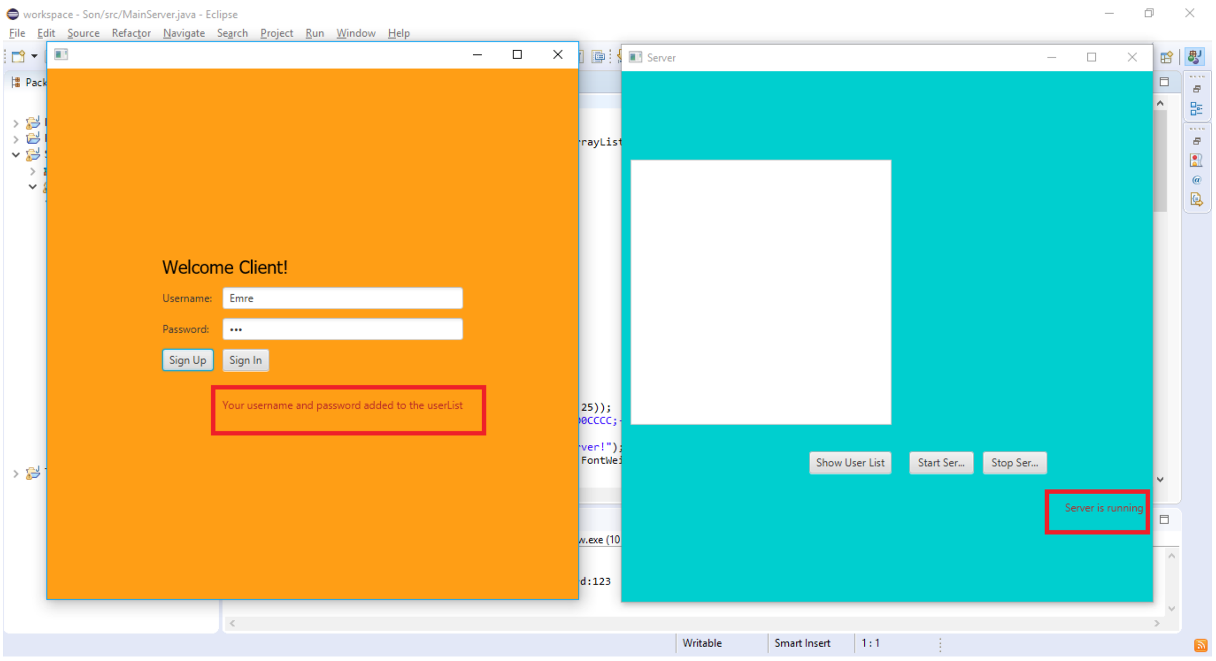
```
No consoles to display at this time.
```

We have used a constant peer server port which listens and when other peer wants to chat, it sends its available port address. With the available port address used for creating peerServer and peerClient.

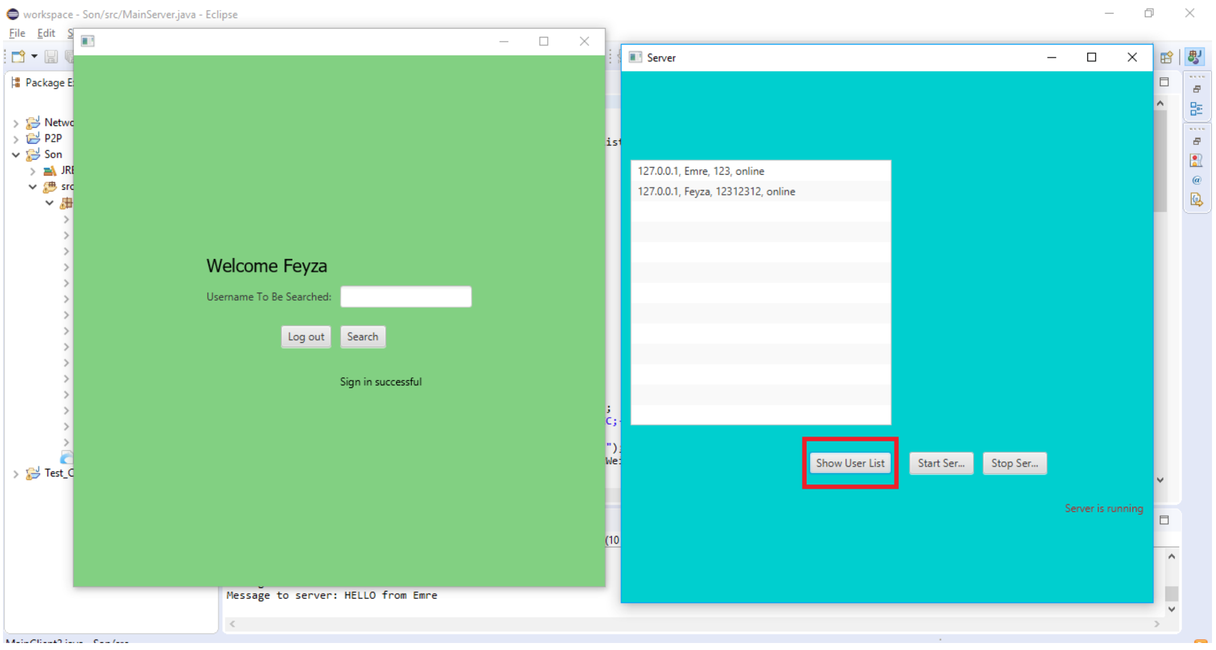
Usage is quite easy. Anybody has a basic English knowledge can easily use it. GUI gives this opportunity mainly. There are two Windows initially. Screenshots is provided for the details:


This is the client window where users can sign up or sign in with their username and passwords.

This is the server window where you can start/stop server and list users already signed up.
workspace - Son/src/MainClient.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Server
Welcome Client!
Username:
Password:
Sign Up
Sign In
Show User List
Start Server
Stop Server


Your username and password added to the userList

Server is running



Welcome Feyza

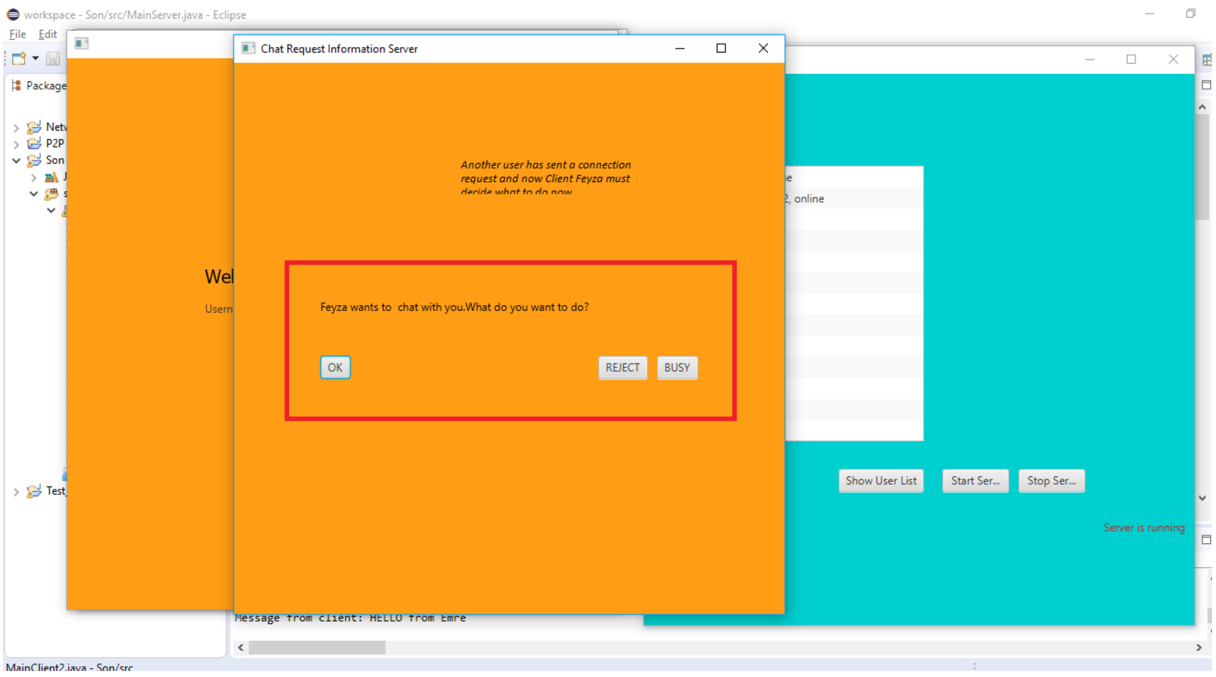
Username To Be Searched:

Sign in successful

127.0.0.1, Emre, 123, online
127.0.0.1, Feyza, 12312312, online

Server is running

```
Message to server: HELLO from Emre
```

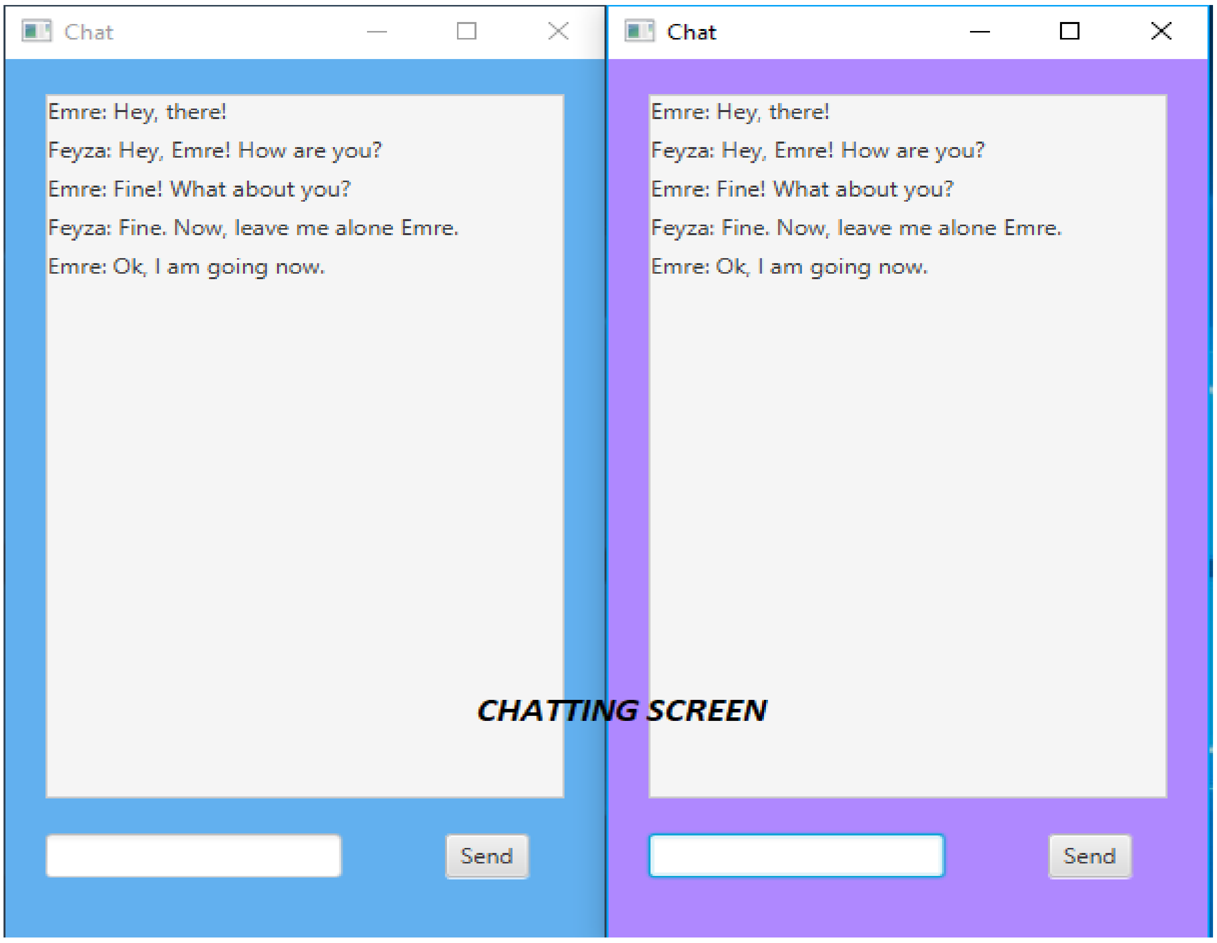

workspace - Son/src/MainServer.java - Eclipse
Chat Request Information Server
OK
REJECT
BUSY
Show User List
Start Ser...
Stop Ser...
MainClient2.java - Son/src

Another user has sent a connection request and now Client Feyza must decide what to do now

Feyza wants to chat with you.What do you want to do?

Server is running

```
Message from client: HELLO from Emre
```



Emre: Hey, there!
Feyza: Hey, Emre! How are you?
Emre: Fine! What about you?
Feyza: Fine. Now, leave me alone Emre.
Emre: Ok, I am going now.

Emre: Hey, there!
Feyza: Hey, Emre! How are you?
Emre: Fine! What about you?
Feyza: Fine. Now, leave me alone Emre.
Emre: Ok, I am going now.

CHATTING SCREEN

As we mentioned before, Server/Client communication is done by using multi-thread server. Whenever a client try to connect server, it is directly directed to the corresponding WorkerRunnable thread. So this takes 3 steps. First one is handshaking with registry server and secondly, a new WorkerRunnable starts working and third and lastly, client starts sending its requests regarding user’s choices to the WorkerRunnable thread.

For UDP connections, the same methodology seemed reasonable and we again made a multi-threaded registry and UDP registry sends port number to the client over TCP connection. UDP Clients can access UDP Server via port number 65507 which is a constant number. When the time UDP Client receives port number of UDP worker thread, it starts sending “HELLO” messages to the that worker UDP Server thread.

In a nutshell, each client consists of 3 main components. A UDP part, a Peer which can both send and receive messages via TCP ports and the client itself. Each client sends its chat messages over Peer’s client part and receives via Peer’s server part. Whenever a message is sent, the GUI updates itself and for the receiver client, whenever Peer’s server part receives message, it also changes the chat window including that message on the screen.

NOTE: UDP Server’s timeout period and “HELLO” message rate is not identical to the assignment’s. Please consider this when testing.

Feyza EKSEN 150113005

Zafer Emre OCAK 150113075