**Java Chat Report**

Coursework

**Author:** COMP1549: Advanced Programming

Victor Dinev – 001115103 University of Greenwich

**Co-authors:** Old Royal Naval College

Zhenyuan Shen – 001197922 United Kingdom

Sapphire Nsinsi Nsiala – 001071889

Kurt K Parlak – 001028714

Tharshan Vijayakumaran – 001112250

***Abstract*** – **In this document we explain the process behind the given task to create a Java Chat System. The system works with a client and a server. They correspond through sockets, sending and reading strings. Each Client (user) chooses a unique username (ID) upon launch and gains access to the chat, where they can send group messages or private messages. The client also displays all online member and the coordinator, which is always the first to join the chat and changes whenever the coordinator disconnects. The server also keeps a record of the chat session in a Memory.txt file, found in the src folder. In this file you can find all recorded server sessions with a start date & time and then the server activities with timestamps.**

**I. Introduction**

In this report you will get familiar with basic and advanced programming principles and techniques used in building a software project. We will look not only at how to write a java chat program, but also how a project is organised and developed. We will also be exploring and demonstrating the main programming principles such as Design Patters, JUnit testing, Fault tolerance and more. By the end of this report, you should know the working process behind the development of software. In other words, you will be prepared to start your first job as a software developer.

**II. Design & Implementation**

First let’s look at the requirements for our application:

* Unique ID/username
* Send/Receive messages
* Public/Private messages
* Coordinator Role
* Display active members
* Timestamp messages
* Record and Store server activity

Those are the basic requirements our chat system should cover. In other words, brief description of what it should look and act like.

**Initial Design**

We decided to use the code from Lab 3 (DemoThree) as the base of our chat design. This code consists of ChatClient and ChatServer, which communicate by passing strings to one another, using socketing.

The Server is multithreaded allowing many users to join. It uses two sets, one for the users and one for the writers (PrintWriter used for communicating with the Clients) and a “Handler” class to handle login, messaging and disconnect. The Handler uses Scanner for input and PrintWriter for output. It works by using a Flag based system, applying a flag to every output. Flags are then interpreted by the Client. The flags are: “SUBMITNAME” asking for a name to be submitted, “NAMEACCEPTED” notifying that the name has been accepted and added to the set of names, “MESSAGE” meaning that the sting after is a message and should be displayed as such. It uses try/catch/finally to handle different scenarios. The login and messaging are part of “try” and disconnecting is handled under “finally” (executes if the “try” is over, meaning the client disconnected). The disconnect works by notifying everyone in the server about the disconnected user, removing them from the set of names and writers and closing the socket.

The client uses “java.swing” for the interface, which consists of message area (containing messages) and text field (used to write and send messages). It also uses a Scanner and PrintWriter for input/output (just like the server) and needs the server IP to be passed as an argument. It also has a getName() function, which shows a Input Dialog box, asking for username. In the run function uses a try/catch/finally statements, similar to the server. In the “try” part it tries to connect to the server and handles the input from the server (interprets the Strings received from the server and the Flags with which they start). In “finally” it disables the interface, executing that code in case of server disconnect.

Now let’s explore what else we must do:

* Coordinator Role
* Private Messages
* Display Active Members
* Timestamp Messages
* Record and Store server activity

**Timestamp**

Let’s start from an easier task and build up. Timestamp should be easy to do and won’t change as we complete the other tasks. In order to timestamp messages, we need to know what the local time is, and add it to the message. We can import and use java.time.LocalTime for that purpose. Now we can add to the messages a timestamp by using the following function: LocalTime.now(), however this displays seconds and nanoseconds as well, which is a bit overwhelming to look at. In order to make it simpler and display only hours and minutes, we can import and use java.time.format.DateTimeFormatter. We also decided to add the timestamp in brackets next to the name (of the sender of a message) and after the server messages (i.e., name has joined). We can do this by simply adding the brackets in the appropriate places and using the following code to add the time: LocalTime.now().format(DateTimeFormatter.ofPattern("HH:mm")). The timestamps are now ready!

**Private Message**

Next, we will do the private messaging. For the private messages to work, we need to be able to send messages only to one user. Currently our code has a set of writers and whenever a message is sent, we loop through that set and send to everyone. Firstly, we need to be able to know which writer is for which user. To do this we will change the set of writers to a dictionary, with the name as key and the writer as value. But how would we know if a message is private or public? In order to differentiate between them we will implement a rule for the private messages. To send a private message the user will have to type “/[username] message” and this will send the message only to the given user. Now, that we have chosen the rule for sending private messages we will have to tell the server how to interpret and act with the new command. To do that we go where the messages are being sent/received in the server code and add a simple if/else statement. It will know that the message is supposed to be private if it starts with “/[”, then it would take the recipient’s name, given in the brackets. To get the name we can use the substring function and indexOf(“[”) method to read what is in the brackets. Then we can use the dictionary to find the name of the person we want to message and send him the private message. The message will start with: <sender>(pm)(<time>):<message> and to display the message to the sender we will do: pm to <receiver>(<time>): <message>.

**Coordinator**

So, we’ve done the timestamps and the private messaging, now let’s set up the coordinator role. We will do this by adding a coordinator variable which stores the name of the coordinator. This will be done in the client and the server, however we will do it for the client, when we do the display member list. This is because currently the client has no record of the members and we will add a way for the server to pass coordinator and members, later. After adding the coordinator variable, we need to give it value, or change its value, when the current coordinator leaves. The coordinator will be initially set to null and will be passed a value when the first user joins and when the current coordinator leaves. It will also need to be set to null when the last user leaves. We can do this with simple if/else statements in the appropriate places, which is straight forward, so we won’t be explaining it in detail here.

**Member List**

After the coordinator has been setup, we have to actually add it to the client as well. Now we will do this and also, setup the displaying of the active members, since they pretty much go together. For this we will add to the client a new JTextArea to display the members and coordinator. They will be displayed on the right-hand side of our main frame. Now we actually need to update the member list whenever someone joins or leaves. We will do this by a sending a string from the server with the appropriate Flag (“COORDINATOR”, “MEMBERS”). In the “COORDINATOR” message we will give the name of the coordinator and in the “MEMBERS” message we will send a list of all members.

It is important to add these to every code related to joining or leaving, so that every time there is a change the client will receive the new values. The client will then interpret the flags, just as it interprets every other Flag we use, meaning it will detect what the input starts with and act accordingly. It will store the coordinator in its own coordinator variable and then it will remove him from the members list. Removing the coordinator might be a bit tricky because, although we can use the replaceAll(coordinator, “”) function, it might remove part of someone’s name if it matches with the coordinator name (i.e., co, nameco – nameco will be only displayed as name). To go around this we will replaceAll(“ ” + coordinarot + “, ”, “ ”), this way we will only replace the coordinator, since another user cannot have “ ” in their name. Then we will clear all “ ”, and replace all “,” with “\n”. In order to not have any problems we have reworked the getName() function to actually check for “ ” and “,” and remove them. They are considered as unallowed characters.

**Server Memory**

The only requirement we have not covered yet is to record and store the server activity. This will be easy to do and then we can get to a little bonus, which will make the client work even if the server is offline (avoiding an annoying error, from the original code). Storing the server activity is fairly simple task. We will do this by with a function called addToMemory() and a Memory.txt file, located in the src folder. The function will take a string as input and store it to the Memory.txt file. We will use a FileWriter for that purpose and after writing our simple function we can start implementing it in the code where needed. First, we want to store when the server was started, which would be in the main function. We will also store the date and time of the server start. Then we want to store every message that has been sent, as well as join/leave actions, which means calling our new function from the appropriate place in the code. The only important thing to keep in mind here, is that we want to do this after the loops that go through all users.

**Server Not Responding**

Remember that bonus, we mentioned in the last paragraph? We will now create a refresh button for the client, so that it can work even if the server is offline. We will also make the client remember the last session it was in, meaning if you were already logged in, the client will automatically log you back in when refreshed. In order to do so, first we will declare a name variable in the client. Whenever we receive a NAMEACCEPTED we also receive the name with it. We add this name to the name variable. Then we would have to change the getName() function to check if name is empty. If it is empty we display the login window, otherwise we return the name. But all that would be pointless if the client closes whenever the server disconnects. We will now create a pop-up window to show that the server is offline and give the option to retry connection. We will do this as an exception in the run method of the Chat Client. So if “try” fails (meaning the connection was interrupted), we will display a JOptionPane.showOptionDialog(…) with retry and cancel option. The option will be stored in an int where “1” would be cancel “-1” would be close (both exit the client) and 0 would be to retry connection. When we test it we will see that it works every time the server disconnects, even if initially it was online. Worth noting is that the client session is stored locally and when the server restarts it doesn’t remember anything from the previous session (apart from what is stored in the Memory.txt file), but the clients will remember it until closed.

**III. Analysis & Critical Discussion**

We have looked at the design and implementation of our project, how it works and what we’ve done. Now we will analyse and discuss the workflow behind it. In other words, what was the process behind building this project.

We started by setting up a GitHub repository. This way we could all work simultaneously and easily keep track of the progress. We analysed the requirements and created issues for every Main task. Then we would create subtasks and declare whenever someone is working on a subtask. We also added labels to easily navigate through the issues. This helped a lot with the workflow of our team.

**Design Pattern**

**JUnit Testing**

**Fault Tolerance**

**Component-Based Development**

**IV. Conclusion**