

GROUP 1 - COMP1549 Coursework

Shuraiz El Malik - 001155341, Jawad Mahmud – 001174077, Naumaan A Malik – 001159264, Shaek Hussain – 001155344, Sri Avantika Yeka – 001155105

COMP1549: Advanced Programming

University of Greenwich

Old Royal Navy College

United Kingdom

Abstract - Most chats nowadays are usually online within a connected network in which multiple people are connected within a network and communicate with each other which makes ease of communication much more effective. This report will mention of how a distributed network system is created via Java in which will allow multiple clients connected to one server to communicate with each other. The creation of this is done by creating a Java Code will include **Client**, as they are the participants that will be a part of the chat, **Server**, as the client will join the chat through the server, **Coordinator**, where the first client in the server will oversee the main server and the other clients. Finally, **IP Address** for which the clients will be entering the server with the same address. Alongside mention of how JUnit, modularity, fault tolerance and components are involved for the creation of the system.

I. Introduction

In this task, we have implemented a networked distributed system for a group-based client-server communication. When a member joins/connects, he/she may provide the parameters as input, a unique ID for each member, a port and IP address. We also explained what each requirement is capable of, and how it works in the client-server communication application. The area of creating a GUI in Java is important in programming as it required to build an interface for Java applications. For the network distributed system to be created, 2 GUIs are to be created which one is for the server and the other for the client which it's design and how it's implemented is mentioned down below.

II. Design and Implementation

To create the design of the GUI, we used the NetBeans IDE and JFrame to display the window. This section will explain further

features/implementation of our solution, the design patterns applied and the JUnit Test.

Client-Side

When a new member joins, they are prompted to enter their student ID number, which then gets stored in the database, allowing each member to communicate in the chat space. To do this, we created the GUI to pop up on a separate window. Features in the GUI we included are radio buttons, including a toggle switch and a regular button. In the main interface, we included labels so that it is easier for the user to figure out which button is for what purpose.

Server-Side

When the GUI opens, the server will display the list of clients who have joined. It displays whoever is messaging in the chat and displays which clients have left the chat. When the first client joins, they are assigned as coordinator and the other clients are not co-ordinator. The rest of the clients on the chat can be considered as participants and the main coordinator can be considered as the admin of the chat. The participants of the chat have the option to join or leave the chat if the server and chat are running simultaneously.

Designs used for Implementation



Figure 2. Client Register

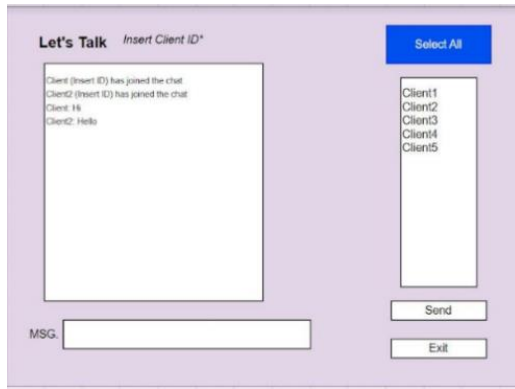


Figure 3. Client

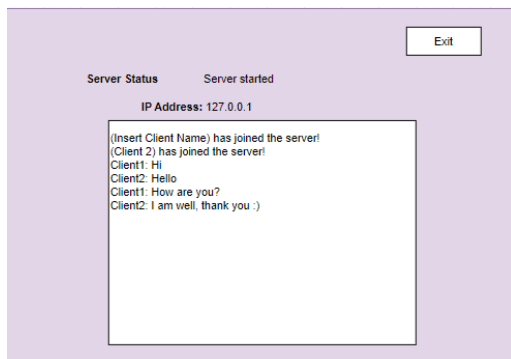


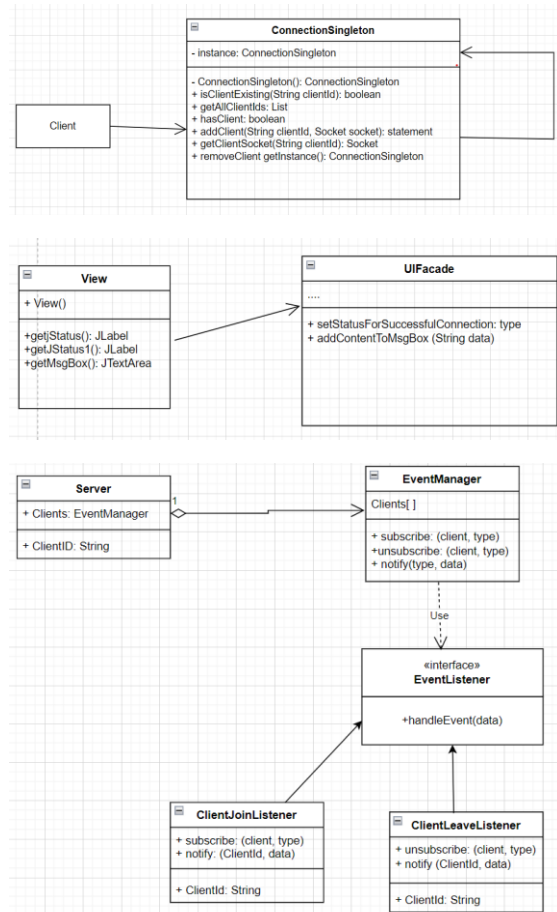
Figure 4. Server

To make our code work, we created a GUI (Graphical User Interface). We chose a GUI as opposed to a CLI (Command Line Interface) Command Line Interface, as we thought the GUI would be more user-friendly, and it would be easier to switch windows when we run the code.

The design for a GUI as well as the implementation of radio buttons was the best for us to implement into our code as we did not want the entire interface to completely change when the user presses a button. This means that the user can switch between the interface without changing it entirely. We created this via JFrame, which enabled us to create a user-friendly interface for users. Once a client logs in, it will display the chat area where they can communicate with other clients via broadcast messaging, who is still active in the chat and the clients that have left will display in the server.

Each time a client joins it will display in the JList box if the client is active or inactive. However, when the clients register, the JList box displays the client's separately once they've joined, which shows clients who are

active or aren't active, represented in how we've created the GUI. To make it even better it could have been improved by implementing a refresh button to display the client's activity together. For the server side, we have created a JFrame form, in which we have placed a JTextArea, JLabels and a JButton which is used to close the server.



For Design Patterns, we have decided to use three design patterns for our implementation, Singleton creational pattern (**Top**), Facade structural pattern (**Middle**) and the Observer behavioural pattern (**Bottom**). We have used these patterns as it will initialise the objects (clients) and execute the methods correctly by allowing the clients to be in the server. Moreover, it is to ensure the application more flexibility and robust of the code so that the application can run efficiently, and that the objects can be given specific responsibilities when using the application, e.g. when the client types in the message, it will be displayed in the text area.

III. Analysis and Critical Discussion

To create the client-side of our coursework, we have used Design Patterns, JUnit Test Cases and Components to be able to help us with the implementation side of the application.

Modularity

For Design Patterns, we have applied three patterns, Singleton creational pattern, Facade structural pattern and the Observer behavioural pattern. **Singleton** creation pattern ensures that a class has just one instance and provides an access point to the instance. We have done this through the methods in whether the client is still in the server, as well as the instance in which the client is added or removed from the server. **Observer** behavioural pattern is used when the Server gets the ClientID, and when it goes to the EventManager, it will use those IDs to show who has joined or left the chat. The EventListener is used to handle the Events of the Client joining and leaving the chat and they will be notified on which client has joined or disconnected to the server. **Facade** Structural pattern is used mainly for the Server side of the application, in which the server status, the IP address (jStatus1) shows that the connection has been successful for the clients to start accessing and for msgBox, it will show that the content from the Client-side will be added.

Fault Tolerance

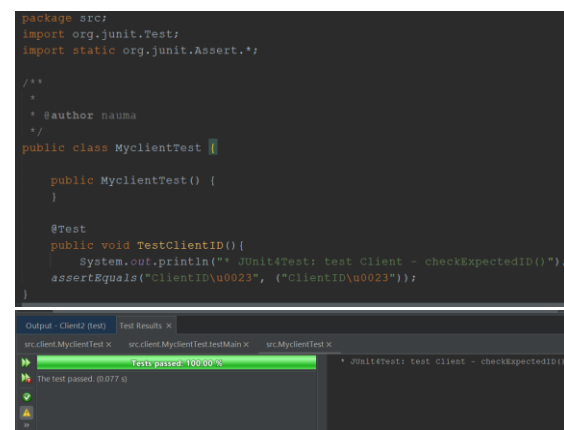
Fault tolerance is when a program functions even when there are errors in the code. The code created for both Server-side and Client-side show no visible errors which indicates no signs of fault therefore fault tolerance is excellent. For instance, when the server fails to send a message to the client, it knows that something is wrong with the client socket connection and handles that exception to let that client know that they are not available for further connection.

White-Box Testing/JUnit Testing

To test our code, we used white box testing as well as unit testing to ensure that each part of our code works as it should. White box testing was beneficial for us as we first had to understand the requirements of the coursework so that we could write up the source code, then we had to understand what each line of the code

does. After we did that, we had to create test cases for each section of our code before we put it together. White box testing was used as we had access to our source code so we could modify it as we wished.

Unit testing was beneficial because it finds bugs early in the code which makes the code seem more authentic. The code tends to be easier to read and more reliable once the bugs are removed from the code. Screenshots below show a JUnit test that has been performed in NetBeans IDE. The result below indicates the how successful the code was with the number of tests executed. When executing these tests, it gave an overall result of 100%, which means the test has been successful.



```
package src;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 * @author nauma
 */
public class MyclientTest {

    public MyclientTest() {
    }

    @Test
    public void TestClientID(){
        System.out.println("JUnit4Test: test Client - checkExpectedID()");
        assertEquals("ClientID\u0023", ("ClientID\u0023"));
    }
}
```

Figure 1. Test Case

Components

Components used: JFrame (required to display the window), JTextField (for entering message and Client ID), JButton (For connecting to the server, pressing send and exit), JLabel, JTextArea (For displaying the clients and their messages on client and server side)

JFrame is used to provide a window and create a frame which doesn't have a title and is initially visible. This has been applied to every GUI window for both Client and Server sides of the application. **JTextField** is a feature in Java which allows editing one line of text, which is used in the Client side so that the client can send a message to other fellow clients within the server to communicate with each other. **JButton** creates a button with a label which is implemented independently. This has been used as an 'Exit' button, in which the user will

be able to exit the chat at any point, as well as a 'Send' button in the client side, so that the client can be able to send their message to the other clients and a 'Connect' button, so that the user will be able to access the chat through their unique ID. **JLabel** can display text, images or both, and it has been applied on both sides of the application to give emphasis to the application. **JTextArea** is an area in Java with multiple lines which displays plain text and has been applied on both sides of the application to display the messages from each client.

Using these components, we were able to create a user-friendly GUI which enables us to do the following:

- Clients can interact in the chat space
- Different clients can login to the chat
- They have the option of leaving the chat or staying in the chat if they desire

With the use of these components incorporated, we were able to create a user-friendly interface that was able to perform the required the tasks to the best of our abilities.

Constraints

Despite the good functionality on the code, there were constraints which surfaced such as with coordinator. Although coordinator is successfully assigned to first client and then to the second client after first client leave, the coordinator is unable to use privileged features. Members list automatically updated rather than updated manually by clients, it is still able demonstrate a fully updated member list for those in the server.

IV. Conclusion

Overall, the creation of the distributed network system is adequate to ensure connection between clients to a server and communication between clients on the server. Different factors had to be considered before creating this code such as possible design patterns with modularity alongside the different components and necessary testing which were successfully met and carried out. Despite the constraints which were present alongside the code, the group was still able to create a functional client-server chat. For possible future work, it can

include trying to amend the constraints and make them much more functional. This also includes creating multiple GUIs to make accessibility much wider and easier and user-friendly.

References

1. Ahmad, R. (2020) *Client side programming multiple chat client on server in Java using multi-threading and socket, YouTube*. YouTube. Available at: <https://www.youtube.com/watch?v=ZzZeteJGncY&t=234s> (Accessed: February 15, 2023).
2. Ahmad, R. (2020) *MyServer multiple chat client on server in Java using multi-threading and socket, YouTube*. YouTube. Available at: <https://www.youtube.com/watch?v=rd272SCI-XE> (Accessed: February 15, 2023).
3. Chris, K. (2022) *What is localhost? local host IP address explained, freeCodeCamp.org*. freeCodeCamp.org. Available at: <https://www.freecodecamp.org/news/what-is-localhost/#:~:text=So%2C%20if%20you%20type%20localhost,address%20that%20starts%20with%20127> (Accessed: March 5, 2023).
4. Hartman, J. (2019). *Java Swing Tutorial: Examples to create GUI*. [online] Guru99.com. Available at: <https://www.guru99.com/java-swing-gui.html> [Accessed 16 Feb. 2023].
5. Java Design Patterns (2023). *Patterns - Java Design Patterns*. [online] java-design-patterns.com. Available at: <https://java-design-patterns.com/patterns/> [Accessed 20 Feb. 2023].
6. RefactoringGuru (2014). *Design Patterns in Java*. [online] refactoring.guru. Available at:

<https://refactoring.guru/design-patterns/java> [Accessed 28 Feb. 2023].

7. Shakula, D. (2022). *Component Class in Java*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/component-class-in-java/> [Accessed 27 Feb. 2023].
8. Suzuki, I. (2004). *Automate GUI tests for Swing applications*. [online] InfoWorld. Available at: <https://www.infoworld.com/article/2073056/automate-gui-tests-for-swing-applications.html> [Accessed 5 Mar. 2023].
9. Vogel, L. (2021). *Unit Testing with JUnit 4 - Tutorial*. [online] www.vogella.com. Available at: <https://www.vogella.com/tutorials/JUnit4/article.html> [Accessed 15 Mar. 2023].