Multi-client Chat Server Program

GROUP 4

Jubril-Awwal Shomoye, Arbaaz Sheikh 001086906, Phakonekham Phichit 001041931, Mariana Villar Pacheco 001088079

COMP1549: Advanced Programming
University of Greenwich
Old Royal Naval College
United Kingdom

***Abstract*—Network communication between different users has continuously been advancing which has made sending and receiving information easier than someone could expect. One of the main models which is currently used nowadays is the 'Client-Server' model, which is based on a server that has a unique IP address and port number which clients can use to establish a connection.**

This paper will provide information about a multi-chat client-server application model using *java* sockets in terms of its design, real-life application, development process and any underlying issues with the model.

Key words: *(multi-client, server, socket programming, concurrency, multithreading, GRASP)*

I. Introduction

As the development of Internet Technology is moving at a swift rate, the use of the internet and web-based applications has started to become an essential element in our lives and is involved in most of our everyday activities (Hai, 2013). Moreover, the application of the Internet has not been limited to desktops but has now been implemented into different digital devices such as 'smart wearables. One of the main uses of the internet is for communicating with other users, and surveys show that across 21 countries surveyed, a medium of 75% of users stated that they use the internet to communicate with other people (Kohut, 2011). The WEB has been built off the client-server model, which means that once the client (person or organisation) sends a request to the server, the server will retrieve and parse that information over to the client if the information is present on the database.

To create a client-server application on a smaller scale, we have designed and created a multi-chat client-server application that allows for multiple clients to connect to the server and communicate either globally or privately with another client. As you can see from figure 1, once a client has successfully connected to the server via the unique server socket, each client is provided with their own socket that allows for reading and writing to be sent and received from either the client or the server. As the project involves multiple clients to send and receive messages to each other, we have decided to implement the use of threads to ensure that each client that connects to the server will receive their own unique socket. This report will describe how a multi-chat program was developed using a client-server model implementing sockets and multi-threading to communicate with other client members concurrently.

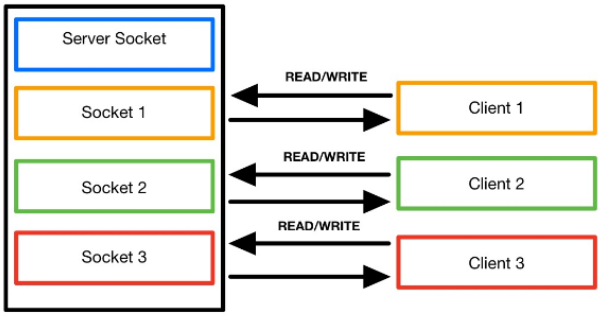

Figure 1: Multi-Chat Model

II. Design/implementation

Socket programming and multithreading are applied in order to create the multi-client chat program. In this

implementation for socket programming, Java is used preferably due to its ability to handle robust operations and I/O devices regarding network programming, as Java has its own independent platform (Kalita, 2014). With the addition of Java multithreading, it allows a convenient path to create a dynamic and robust server that suits the need of the multi-client chat application through the developing cycle. Utilising Java also provides better security. Therefore, it is ideal in this implementation to use Java to perform the best-optimised socket programming(*ibid*). For this implementation of server sockets, the port number and IP address have been set to 7000 and 127.0.0.1, respectively.

As a graphic representation of the client-server program, a Unified Modeling Language (UML) class diagram was implemented. UML is a software analysis and design standard often used as a tool to model an object-oriented system (Berardi *et al.*, 2005). Moreover, class diagrams, the most used UML diagram, enables developers to visualize and represent the model's classes, relationships, methods and attributes within an OO system (Evans, 1998).

It is possible to visualize and understand at a very low level the classes that compose the system and their functionality from the diagram in figure 2. This design was also developed taking into account the general priorities that the project required, such as; Ease of use, simplicity, efficiency and reliability.

In order to apply the concepts of socket programming and multi-threading six classes were created: *Client*, *ClientHandler*, *MessageHandler*, *ReadOnlineClients* , *Server* and *ServerConnection*. From the server-side, the core class is *Server*, which inherits most of its attributes and methods from the abstract class *ServerConnection*, a class that extends the Thread class. Further, the server can only be instantiated from the *ServerGUI* and its methods, such as startConnection() and closeConnection(), are also implemented through the GUI class. The server, because of its inherited methods, can have multiple *ClientHandlers*, which handle the client's messages and overall connection. At the same time, a server instance uses the *OnlineClients* class, a serializable class created to store a client's username and send it through the ObjectOutputStream() method to the client.

From the Client side, the class *ReadOnlineClients* also uses the *OnlineClients* class, which is used within the *Client* class. Client objects can only be instantiated from the *LoginGUI*, where the main attributes from the client come from. Similarly, a *ClientGUI* is also instantiated by a connected client (when the connection is successful). Moreover, each connected client has a *MessageHandler*, where the special commands and messages from the server are handled.

The image below displays the UML Class Diagram of the multi-chat application.

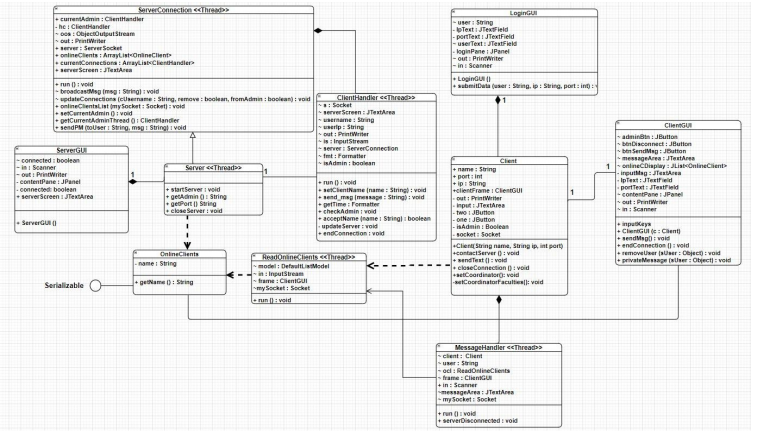

Figure 2: Multi-Client server UML class diagram

As for GUI design and implementation, the GUI was developed with three main goals: Simplicity, responsiveness, integrated functionality (with other classes), ease of use and aesthetically appealing to the system's user, which led to the decision to choose the java.swing widget toolkit. Also, it must enable the user to conduct the program objectives and specification:

1. Start the server
2. Close the server
3. Accept input client details
4. Start the client-server connection
5. Show the broadcasted and private messages with timestamps
6. Display a list of the current clients connected to the server
7. Press enter to send a message
8. Send private messages
9. Ctrl + C to exit the program
10. Access coordinator privileges, if applicable

The UML diagram focuses around GRASP patterns. General Responsibility Assignment Software Patterns (or

Principles), GRASP, uses different patterns and principles for object-oriented software design. These patterns and principles are low coupling, high cohesion, controller, creator, information expert and pure fabrication. GRASP patterns were chosen as they allow developers to determine the classes needed for the software and decide their responsibilities.

One of the principle patterns implemented is low coupling.The term coupling is commonly used to measure the relative interdependency between various classes as one class connected to another class. There are many cases of low coupling in the presented class diagram (Figure 2). For example, *ClientGUI* is dependent on *Client* and *ServerConnection* is dependent on Server. Cohesion is known as the strength of the attributes inside a class. It is noticeable in the UML diagram (Figure 2) that the classes *ClientHandler*, *ServerConnection* and *Client*. Controllers are given the responsibility of dealing with the system events to a non-UI class which represents the whole system. An example of a controller in Figure 2 are *Server*, *ClientHandler* and *ServerConnection*, as they manage system events that occur during the program as it runs.

Another important GRASP principle is the implementation of creators. A creator is a class that is responsible for the creation of objects. The *Client* is a creator as it creates the object *ClientGUI* which allows the user to send and receive messages to and from other users. A further pattern is Information Expert(s) which are employed to assign specific responsibilities to classes. *MessageHandler* and *ServerConnection* are an illustration of information experts in the developed model. These classes handle the messages and activity that occur during the server connection.

Pure Fabrication is a fabricated class which has a set of related responsibilities in order to achieve low coupling and high cohesion. A representation of this principle in our program are *LoginGUI* and *ReadOnlineClients*. *LoginGUI* is responsible for assimilating the information of the user when they attempt to login. Additionally, *ReadOnlineClients* has the responsibility of storing the data of a user. Despite it not being a part of GRASP, inheritance has also been implemented. for our UML. Inheritance is interpreted as the mechanism of "inheriting" the attributes of a class or object into another object. An illustration of this is *Server* and *ServerConnection*. As *Server* inherits the attributes of *ServerConnection* abstract class.

III. Analysis and Critical Discussion

As described previously from the UML, the design has two core classes which start the program's main functionality (server and client), which are displayed to the user through its respective Graphical User Interface (GUI).

When the *ServerGUI* class is run, its main method creates a JFrame where the Server can be created and started from a JButton. After it has been started and is waiting for a connection, the button can also close the server connection. To establish a connection with the server, a user must run the *LoginGUI*, an interface where it can input their details (ID, server port and IP) in order to create its Client instance and call the contactServer() method. If the connection is unsuccessful, a messageDialog will be prompted to the user to let them know the cause: Invalid details, Invalid ID or unavailability of the server.

With a successful connection, the *ClientGUI* is instantiated and immediately visible to the client. The ClientGUI window is composed of a scrollable JTextArea field called *messageArea* where all the incoming messages to the server will be displayed, another JTextArea *inputMsg* where the user is able to write their message and a *onlineCDisplay* JList component where the list of current connected users will be visible. In addition, each *ClientGUI* will have from three to four JButtons depending on their privileges (coordinator or regular client): Send Message, Private Message, Remove user and Disconnect button. It is important to note that the *Remove User* and *Private Message buttons* are not immediately enabled to the user, as it must click on an online user before.

Behind the *ClientGUI*, the classes that have most of the functionality is the *Client* class and its thread class to read messages called *MessageHandler.* Simultaneously, the server (thread) handles clients' connections and messages with another thread class called *ClientHandler.* When a connection is established and the *ClientGUI* is visible, the client is informed if they are the server's first client. Every client can broadcast messages with the *Send* button and send private messages to a selected user from the online users list with the *Private message button*. Messages can also be broadcasted directly after pressing

the *enter* key. Both broadcasted and private messages are displayed in the *ClientGUI's messageArea* with a distinctive terminology for the user, shown in the table below (table 1):

[clientName < everyone]	broadcasted message
[clientName1 < clientName2]	private message

Table 1: Message terminology

The server will automatically assign as coordinator the first connected client, which enables this user to modify and remove a client from the current online users list. The button *Remove Button* is only visible and enabled to the server's current coordinator. If for any reason the current coordinator is disconnected from the server, the next client who joined after them becomes the current coordinator, as the array list which stores all the active connections is constantly updated (when someone joins or disconnects). The chosen implementation for the server's coordinator role is through a *isCoordinator* boolean in both the *ClientHandler* and *Client* class. The current coordinator's thread is stored in the server under the variable name *currentCoordinator*. Concurrently, the client can also know if they are a coordinator through the *isCoordinator* boolean, which enables the functionality of the *setCoordinatorFaculties()* and *removeUser()* methods from the Client class.

Furthermore, special commands are sent from the server to the client and vice versa, through the connected socket, to conduct specific functionality such as add users to the online clients display, remove users from the server, etc. The tables below show the special messages in the server-client connection.

Server special commands to client	
#INVALIDNAME	If clients ID is repetitive or invalid
#A	Add new user to the list of online clients
#R	Remove new user to the list of online clients
You are the coordinator.	Lets the client know if they are the server's current user

Table 2: Server special commands

Client special commands to server	
first message: client's name	For the server to identify each *ClientHandler* thread
second message: client's ip	
#PRIVATEMSG	Indicates when a private message is sent
(coordinator) #USEREMOVE	Indicates which server the coordinator wants to remove

Table 3: Client special commands

If a client has not been active for more than two minutes, it will be automatically disconnected from the server. This has been achieved implementing setSoTimeout() method to every client's socket after a successful connection. It will then prompt the user a *messageDialog* with a "Server disconnected" message, which will close its *ClientGUI* window. The same message appears to every connected client if the server has been closed from the *ServerGUI*.

Fault Tolerance for the coordinator is executed when the coordinator is disconnected or inactive for 2 minutes. When either scenarios are met, the coordinator will set a role to the second joined client. The fault tolerance algorithm is done through the ServerConnection public method which is also implemented by using the setSoTimeout(). It is imported from a Javabean API component from socket programming in Java.

To ensure that the developed program is fully functional, we produced a variety of 'unit tests' to check several components that have been used in the program. The tool implemented to conduct our unit testing was 'Junit' which uses an open-source framework to create and run automated tests of components used in the program.

As the created program is a multi-chat application program using the client-server model, we decided to create a few test cases to check the functionality of the main three classes: Client, Client Handler and Server. To check the client class was functioning correctly, our test case creates a 'test' client named *Jake* and attempts to connect the user to the server which was automatically started through the Junit program. Adding onto this, we also created a test to check whether the first member to

connect to the server would automatically be assigned a 'co-ordinator role'. In order to test this we created two test clients *Jake & Ben* and checked if the program would assign the coordinator role to the first user. Once executed, the first user *Jake* received the co-ordinator permissions while *Ben* was provided with a client role instead, this shows that the test was successful. To test the *ClientHandler* class, we created a user named *Jake* and connected it to the server. Once the connection had been made successfully, we checked if the *ClientHandler* was able to detect the users that were currently connected to the server and found out that the *ClientHandler* class was functioning correctly. Finally, the last test that we had created was to check whether the *Server* class would be able to allow for a client to connect to the server and if the server is able to disconnect a user from the server. Once the Junit test was executed, both cases passed and the server was able to create and break a connection with a client named *Jake*.

In order to improve the current model, we could have implemented java cryptography to encrypt data that could be used in a malicious way, this includes IP Addresses and Port Numbers that are used to connect clients to the server. This ensures that no unwanted user is able to connect to the chat program and communicate with the other clients. In addition to this, we could have implemented a method that allows for the coordinator to accept or reject a client once they are attempting to connect to the server. This would ensure that only users that have been accepted to create a connection with a server would be able to join the multi-chat program and any user that had been rejected would have to wait 5 minutes before attempting to reconnect. Lastly, another improvement that could be made to the program would be to allow for clients to be able to access previous message history from the server as once the server is closed the messages are deleted and the clients are unable to retrieve the previous messages. This would be useful as if there was an issue with the server that caused a disconnection, the clients would be able to retrieve the previous messages once a new connection is made to the server.

IV. Conclusions

To conclude, the client-server model has been around for many years and has been proven to be both an effective and reliable method to receive and send information to many clients that are connected to the server. Through our work, we have seen how a client-server model can be used to create a multi-chat application program using both sockets and threads to allow for users to communicate either globally or privately to another client on the same network. However, the client-server model is very susceptible to attacks on either the client or the server side which could prove to be damaging to an organisation if they used this method to transfer files and documents to other users. In addition to this, if the number of clients that connected to the server increases at a rapid rate this could result in a server overload and potentially a loss of data. As more users connect to the server, the server would have to be scaled to accommodate for the increase in users which would result in high costs of maintenance and deployment. Currently, the client-server model still remains the best architecture to date and the advantages outweigh the disadvantages and will likely remain the core model used in computing developments for the many years to come.

References

Berardi, D., Calvanese, D. and De Giacomo, G., 2005. Reasoning on UML class diagrams. Artificial Intelligence, 168(1-2), pp.70-118.

Evans, A., 1998. Reasoning with UML class diagrams. Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, pp.102-113.

Joshi, P. and Sen, K., 2008, September. Predictive typestate checking of multithreaded java programs. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering* (pp. 288-296). IEEE.

Kalita, L., 2014. Socket programming. *International Journal of Computer Science and Information Technologies*, *5*(3), pp.4802-4807.

Kohut, A. and Wike, R., 2011. *Global digital communication*. [Washington, D.C.]: Pew Research Center

Taboada, G.L., Touriño, J. and Doallo, R., 2008. Java Fast Sockets: Enabling high-speed Java communications on high performance clusters. *Computer Communications*, *31*(17), pp.4049-4059.

Zhang, H., 2013. Architecture of network and client-server model. *arXiv preprint arXiv:1307.6665*.