Design & Implemenation: Internet Text Chatting Program

Tan Wei Xuan (49003140) tanweixuan@postech.ac.kr
Zhang Xin Yue (49003143) xyzhang@postech.ac.kr

March 15, 2019

1 Project Overview

The **Internet Text Messaging Program** that I have implemented is a **Multi-threaded Chat Application** that utilises the Transmission Control Protocol *(TCP)* and allows for mutiple clients to send/receive messages over a server, to one another, at the same time. My program is written in the ***Java language*** and the source code files are as follow:

1. **Server Files**
   - ServerMain.java
   - Server.java.
   - ClientThread.java
2. **Client Files**
   - Client.java

2 Design

My program is built on the *Java Socket Programming API* and the transfer protocol used is the **Transmission Control Protocol *(TCP)***. I have chosen to use **TCP** instead of **User Datagram Protocol *(UDP)*** as my transfer protocol as TCP outweighs UDP in terms of reliablity and it has in-order data delivery.There are 4 tuples in a TCP connection and there are as follow:

1. Source IP Address
2. Source Port
3. Target IP Address
4. Target Port

My program utilises the **Sockets API Library** for Java, *java.net.ServerSocket* and *java.net.Socket*, to implement the Socket Client and Socket. *java.net.ServerSocket* provides a system-independent implementation of the server side of a client/server socket connection. *java.net.Socket* implements client sockets and the socket is an endpoint for communication between two machines.The

design of my program is based on the sequence of socket API calls and client/server communication flow for TCP (shown below).

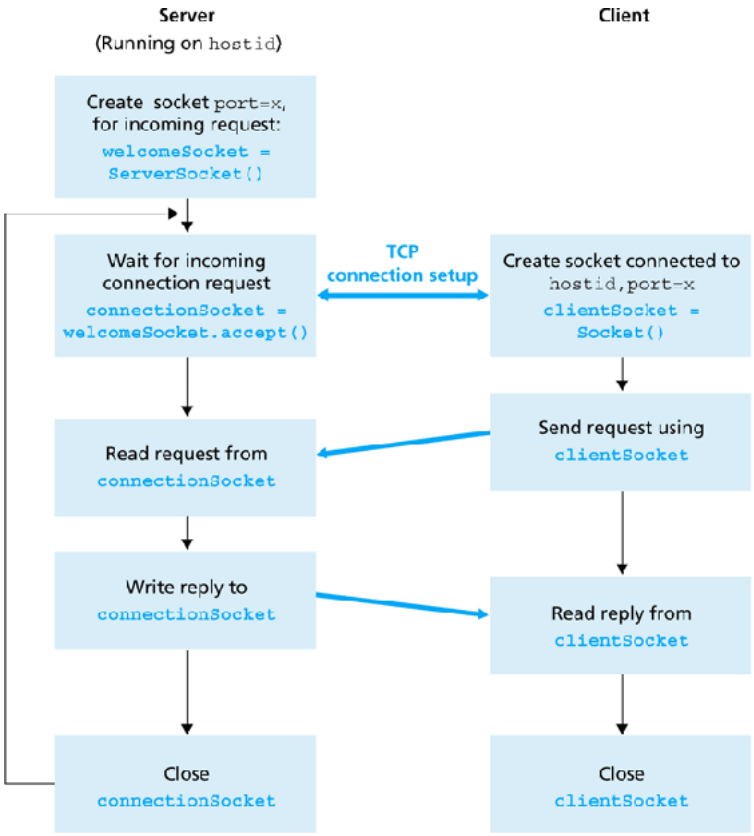

Figure 1: "TCP Client/Server Communication Flow"

3 Implementation

3.1 Server

The server utilises the *ServerSocket* class to implement the server side of the client/server socket connection. Usually, a server runs on a specific computer on the network and has a socket that is bound to a specific port number and a Host name *(IP-to-Host name resolution)* obtainable through the*InetAddress class*. In my project, we use the same computer as both the client and the server.

3.2 Client

The Client utilises the *Socket* class to implement the client side of the client/server socket connection. The client must know the hostname or IP of the machine on which the server is running and the port number on which the server is listening. In my project, the port I am using is *2222* and the hostname is *localhost*

3.3 Client Thread

The Client Thread is a **thread** utilised by the Server to service multiple Clients simultaneously by allowing multiple clients to be connected to the server concurrently and creating a new socket for every new client. It services each client's request different thread. The number of clients being served simultaneously will equal the number of threads running. The Client thread reads from and writes to the client connection as necessary.

4 Client/Server Communication Flow Implementation

4.1 Create Server Socket port

My program begins by creating a new *ServerSocket* object to listen on a specific port. When running this server, choose a port that is not already dedicated to some other service. In my program, the server is started on port *2222* with *localhost* as the *Host Name*.

```
//ServerMain.java
//Port number can be changed
final int portNumber = 2222;
//IP-to-Host Name Resolution
//Can be configured to connect to external IP address
final InetAddress addr = InetAddress.getByName("127.0.0.1");
```

```
//Server.java
//Open a Server Socket on the Port Number
serverSocket = new ServerSocket(serverPort,0,addr);
```

4.2 Wait for Incoming Connection Request

If the server successfully binds to its port, then the *ServerSocket* object is successfully created and the server continues to the next step - accepting a connection from a client .The accept method waits until a client starts up and requests a connection on the host and port of this server.

```
//Server.java
Socket clientSocket = serverSocket.accept();
```

4.3 Create Client Socket connecting to Server

The server should already be running and listening to the port, waiting for a client to request a connection. The first thing the client program does is to open a socket that is connected to the server running on the specified host name and port:

```
//Client.java
// The client socket
private static Socket clientSocket = null;
// The output stream
// The server port.
int portNumber = 2222;
// The server host.
// For now we are using localhost
String host = "localhost";
clientSocket = new Socket(host, portNumber);
```

4.4 Server - Write/Receive message from Client

At this point, the new Socket object puts the server in direct connection with the clientl, allowing us to access the output and input streams to write and receive messages to and from the client respectively. The server is capable of exchanging messages with the client endlessly until the socket is closed with its streams.

```
//ClientThread.java
//For Receiving Messages from Client
inputStream = new DataInputStream(clientSocket.getInputStream());
//For Writing Message to Client
outputStream = new PrintStream(clientSocket.getOutputStream());
```

4.5 Client - Write/Receive message from Server

When the server has accepted the connection, we can then obtain input and output streams from the *Client Socket* to communicate with the server. The input stream of the client is connected to the output stream of the server, just like the input stream of the server is connected to the output stream of the client.

```
//Client.java
//Output Stream is connected to the Input Stream of the server
outputStream = new PrintStream(clientSocket.getOutputStream());
//Input Stream is connected to the Output Stream of the server
inputStream = new DataInputStream(clientSocket.getInputStream());
```

4.6 Client - Closing Connection

In my program, stopping the connection to the server is done when the client input the word, *"/quit"*. By doing this, alll the Client's Input and Output Streams and the Client's socket will be closed

```
//ClientThread.java
while (true) {
  String line = inputStream.readLine();
  if (line.length() <= 0)
  continue;
  //Quits the chatroom
  else if (line.startsWith("/quit")) {
    break;
  }
}
//Closes the Input/Output Stream and the Client Socket
inputStream.close();
outputStream.close();
clientSocket.close();
```

4.7 Server - Closing Connection

As my program is a multi-threaded chat server where multiple clients will be able to connect/disconnect at any time as long as the server is open, closing off the server will require the *host* to close off the *Server Socket* explictly on his/her end. This will release the port that the server is being binded to and prevent any more clients from connecting to it. The server can be terminated through either the *command prompt/terminal* or the *Integrated Development Environment (IDE).*

5 Running the Program

** For detailed program execution, please refer to* ***Screen Record of Programme Execution.mp4***

5.1 Integrated Development Environment

1. Import the Project
2. Run ***ServerMain.java*** to start the **server**
3. Run ***Client.java*** to start the client (Run ***n*** times to start ***n*** clients)

5.2 Exectuables

1. Run ***ServerMain.bat*** to start the **server***(*DO NOT run SeverMain.jar)*
2. Run ***Client.jar*** to start the client (Run ***n*** times to start ***n*** clients)

6 References

[1] A Guide to Java Sockets
https://www.baeldung.com/a-guide-to-java-sockets

[2] Writing the Server Side of a Socket
https://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html

[3] Java Socket Programming - Socket Server, Client Example
https://www.journaldev.com/741/java-socket-programming-server-client