Computer Network

Server-Side Socket Programming

GROUP 2:

Izaaz Rahman Akbar	(21/472855/PA/20348)
Matthew	(21/478240/PA/20736)
Rabbani Nur Kumoro	(21/472599/PA/20310)
William Hilmy Susatyo	(21/472585/PA/20308)

Computer Science Study Program

Department of Computer Science and Electronics

Faculty of Mathematics and Natural Science

Universitas Gadjah Mada

Yogyakarta

2022

I. Introduction

Socket programming is a way for computers to communicate with each other over a network using a specific set of rules. It allows a server to listen for incoming connections, and a client to connect to the server and send data. Sockets can send data between a client and a server running on the same computer or between two different computers on a network. Socket programming can be used to create a variety of different types of programs, such as web servers, file transfer programs, and chat applications.

Server-side socket programming refers to the use of sockets on a computer acting as a server in a client-server model. When a server is waiting for incoming connections, it is said to be "listening" to them. To create a server-side socket, a programmer first creates a "listening" socket that listens for incoming connections on a specific port. When a client connects to the server, the server creates a new socket to communicate with the client. This allows the server to handle multiple client connections simultaneously, using a separate socket for each client. Server-side socket programming involves creating a socket, binding it to a specific port, and then listening for incoming connections. Once a connection is established, the server can send and receive data with the client using the socket. The server can also close the connection when it is no longer needed. Server-side socket programming is useful for creating networked applications such as web servers, chat servers, and file transfer programs.

II. Problem Definition

Throughout this report, we will create a server-side chat platform that fulfills the following conditions:

1. Servers able to handle messages from multiple clients,
2. Clients can input their nicknames,
3. Clients can send and receive messages either with another client personally (private chat) or with multiple clients simultaneously (group chat).

In this project, we use Java Programming Language as a tool to accomplish the goals listed above. Java is widely known for its portability, ease of use, object-oriented design, concurrency support, and security features making it a good choice for socket

programming tasks. Furthermore, to approach the solution better, we will focus on several important aspects including the architecture, protocol, and mechanism of the server-side socket programming in the subsequent section.

III. Architecture

Sockets are used for two-way communication between programs on a network. In the Client-Server architecture, there are two sockets: one representing the client and one representing the server.

IV. Protocol

In server-side socket programming, a protocol is a set of rules that govern the communication between the server and the client. Protocols define the format of the messages that are exchanged between the server and the client, as well as the actions that are taken based on the messages. In our case, the message type will be a string of characters. In addition, the client needs to specify another user's username anywhere in the message so that the message will be sent to that specific user.

In this case, we use the Transmission Control Protocol (TCP) as the protocol. TCP is considered a reliable, connection-oriented protocol that ensures that all data is delivered to the destination. It is often used for applications that require reliable, error-free communication, such as file transfer and email.

V. Message Types

Our project implements several features regarding the types of messages, namely:

1. Provide a feature that displays the current time,
2. Directly remove text in the text box after the message is sent.
3. Show the exact sending time of every message from each client,
4. Enable the client to leave the chat room or rejoin using another username by clicking the button on the upper right side of the screen.

VI. How it Works

Three parts become the main concern in this project: Server, Client, and Client Handling Process. Hence, we create separate files that depict the workflow of those parts

Server.java

This Java code defines a class called Server that represents a server in a client-server architecture. The server listens for incoming connections on a specified port, and when a new client connects, it creates a ClientHandler object to handle the communication with the client.

The Server class has the following methods and fields:

- **ServerSocket serverSocket**: a field that represents the server socket, which is used to listen for incoming connections.
- **Server(ServerSocket serverSocket)**: a constructor that initializes the serverSocket field with the provided serverSocket parameter.
- **void startServer()**: a method that starts the server by continuously listening for incoming connections and creating a ClientHandler for each new client.
- **void closeServerSocket()**: a method that closes the serverSocket field.
- **public static void main(String[] args)**: the main method of the Server class, which creates a ServerSocket object and starts the server by calling the startServer() method.

Moreover, the Server class also imports three other classes, namely java.io.IOException, java.net.ServerSocket, and java.net.Socket. IOException is a class that represents an input/output exception, which can be thrown when there is an error reading from or writing to a stream. ServerSocket is a class that represents a server-side socket, which can be used to listen for incoming connections. The socket is a class that represents a socket

for communication between two computers.

Server.java

```java
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {

  private ServerSocket serverSocket;

  public Server(ServerSocket serverSocket) {
    this.serverSocket = serverSocket;
  }

  public void startServer() {
    try {
      while (!serverSocket.isClosed()) {
        Socket socket = serverSocket.accept();
        System.out.println("A new client has connected");
        ClientHandler clientHandler = new ClientHandler(socket);
        Thread thread = new Thread(clientHandler);
        thread.start();
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void closeServerSocket() {
    try {
      if (serverSocket != null) {
        serverSocket.close();
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    try {
      ServerSocket serverSocket = new ServerSocket(3000);
      Server server = new Server(serverSocket);
      server.startServer();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Client.java

This Java code defines a class called Client that represents a client in a client-server architecture. The client can connect to a server using a specified IP address and port, and send and receive messages over the connection.

The Client class has the following methods and fields:

- **Socket socket:** a field that represents the socket for communication with the server.
- **BufferedReader bufferedReader**: a field that represents a reader for reading data from the server.
- **BufferedWriter bufferedWriter**: a field that represents a writer for writing data to the server.
- **String username**: a field that represents the client's username.
- **Client(Socket socket, String username)**: a constructor that initializes the socket, bufferedReader, bufferedWriter, and username fields with the provided socket and username parameters, and creates a BufferedReader and BufferedWriter for reading and writing data over the connection.
- **void sendMessage()**: a method that sends messages to the server. The method reads input from the user and writes it to the server using the bufferedWriter.
- **void listenForMessage()**: a method that listens for messages from the server. The method creates a new thread that continuously reads data from the server using the bufferedReader and prints it to the console.
- **void closeEverything(Socket socket, BufferedReader bufferedReader, BufferedWriter bufferedWriter)**: a method that closes the socket, bufferedReader, and bufferedWriter fields.
- **public static void main(String[] args)**: the main method of the Client class, which prompts the user for a username and creates a Socket object to connect to the server. It then creates a Client object and starts listening for and sending messages.

The Client class also imports three other classes: java.io.IOException, java.net.Socket, and java.util.Scanner. IOException is a class that represents an input/output exception, which can be thrown when there is an error reading from or writing to a stream. A socket

is a class that represents a socket for communication between two computers. The scanner is a class that can be used to read input from the console.

```java
import java.io.*;
import java.net.Socket;
import java.util.Scanner;

public class Client {

  private Socket socket;
  private BufferedReader bufferedReader;
  private BufferedWriter bufferedWriter;
  private String username;

  public Client(Socket socket, String username) {
    try {
      this.socket = socket;
      this.bufferedWriter = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
      this.bufferedReader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      this.username = username;
    } catch (IOException e) {
      closeEverything(socket, bufferedReader, bufferedWriter);
    }
  }

  public void sendMessage() {
    try {
      bufferedWriter.write(username);
      bufferedWriter.newLine();
      bufferedWriter.flush();

      Scanner scanner = new Scanner(System.in);
      while (socket.isConnected()) {
        String messageToSend = scanner.nextLine();
        bufferedWriter.write(messageToSend);
        bufferedWriter.newLine();
        bufferedWriter.flush();
      }
    } catch (IOException e) {
      closeEverything(socket, bufferedReader, bufferedWriter);
    }
  }
```

```java
  public void listenForMessage() {
    new Thread(
      new Runnable() {
        // @override
        public void run() {
          String msgFromGroupChat;

          while (socket.isConnected()) {
            try {
              msgFromGroupChat = bufferedReader.readLine();
              System.out.println(msgFromGroupChat);
            } catch (IOException e) {
              closeEverything(socket, bufferedReader, bufferedWriter);
            }
          }
        }
      }
    )
      .start();
  }

  public void closeEverything(Socket socket, BufferedReader bufferedReader, BufferedWriter
bufferedWriter) {
    try {
      if (bufferedReader != null) {bufferedReader.close();}
      if (bufferedWriter != null) {bufferedWriter.close();}
      if (socket != null) {socket.close();}
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    try {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter your username for the group chat: ");
      String username = scanner.nextLine();
      // Make sure to change the ip address and port below to match the server's ip
      // and port
      // The server's port is located in Server.java
      Socket socket = new Socket("192.168.1.21", 3000);
      Client client = new Client(socket, username);
      client.listenForMessage();
      client.sendMessage();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

ClientHandler.java

This Java code defines a class called ClientHandler that represents a handler for communication with a client in a client-server architecture. The ClientHandler class implements the Runnable interface, which means it has a run() method that can be executed in a separate thread.

The ClientHandler class has the following methods and fields:

- **static ArrayList<ClientHandler> clientHandlers**: a static field that represents a list of all ClientHandler objects.
- **Socket socket**: a field that represents the socket for communication with the client.
- **BufferedReader bufferedReader**: a field that represents a reader for reading data from the client.
- **BufferedWriter bufferedWriter**: a field that represents a writer for writing data to the client.
- **String clientUsername**: a field that represents the client's username.
- **ClientHandler(Socket socket)**: a constructor that initializes the socket, bufferedReader, bufferedWriter, and clientUsername fields with the provided socket parameter. It also adds the ClientHandler object to the clientHandler list and reads the client's username from the bufferedReader.
- **public void run()**: the implementation of the run() method inherited from the Runnable interface. This method continuously listens for messages from the client and directs them to the appropriate recipients.
- **void directMessage(String messageToSend)**: a method that directs a message to the appropriate recipient(s). If the message contains a client's username, it is sent to that client.
- **void removeClientHandler()**: a method that removes the ClientHandler object from the clientHandler list and sends a message to all clients that the client has left the chat.
- **void closeEverything(Socket socket, BufferedReader bufferedreader, BufferedWriter bufferedWriter)**: a method that closes the socket, bufferedReader, and bufferedWriter fields and removes the ClientHandler object from the clientHandler list.

The ClientHandler class also imports three other classes: java.io.IOException, java.net.Socket, and java.util.ArrayList. IOException is a class that represents an input/output exception, which can be thrown when there is an error reading from or

writing to a stream. The socket is a class that represents a socket for communication between two computers. ArrayList is a class that represents a resizable array of objects.

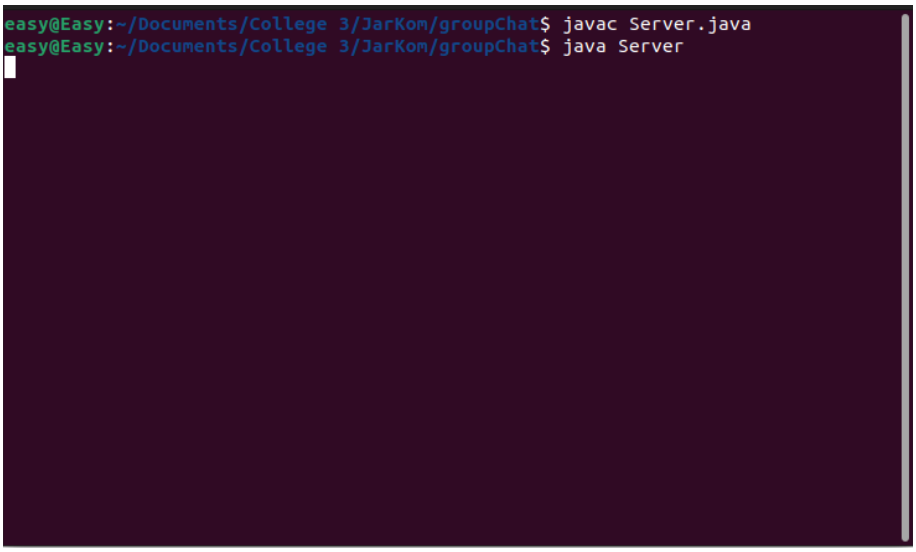
ClientHandler.java

```java
import java.io.*;
import java.net.Socket;
import java.util.ArrayList;

public class ClientHandler implements Runnable {

  public static ArrayList<ClientHandler> clientHandlers = new ArrayList<>();
  private Socket socket;
  private BufferedReader bufferedReader;
  private BufferedWriter bufferedWriter;
  private String clientUsername;

  public ClientHandler(Socket socket) {
    try {
      this.socket = socket;
      this.bufferedWriter = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
      this.bufferedReader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      this.clientUsername = bufferedReader.readLine();
      clientHandlers.add(this);
    } catch (IOException e) {
      closeEverything(socket, bufferedReader, bufferedWriter);
    }
  }

  public void run() {
    String messageFromClient;
    while (socket.isConnected()) {
      try {
        messageFromClient = bufferedReader.readLine();
        directMessage(messageFromClient);
      } catch (IOException e) {
        closeEverything(socket, bufferedReader, bufferedWriter);
        break;
      }
    }
  }

  public void directMessage(String messageToSend) {
    for (ClientHandler clientHandler : clientHandlers) {
      System.out.println(clientHandler.clientUsername);
      try {
        if (messageToSend.contains(clientHandler.clientUsername) && !clientHandler.clientUsername.equals(this.clientUsername)) {
          clientHandler.bufferedWriter.write(this.clientUsername + ": " + messageToSend);
          clientHandler.bufferedWriter.newLine();
          clientHandler.bufferedWriter.flush();
        }
      } catch (IOException e) {
        closeEverything(socket, bufferedReader, bufferedWriter);
      }
    }
  }

  public void removeClientHandler() {
    clientHandlers.remove(this);
    directMessage("SERVER: " + clientUsername + " has left the chat");
  }

  public void closeEverything(Socket socket, BufferedReader bufferedreader, BufferedWriter bufferedWriter) {
    removeClientHandler();
    try {
      if (bufferedReader != null) {bufferedReader.close();}
      if (bufferedWriter != null) {bufferedWriter.close();}
      if (socket != null) {socket.close();}
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

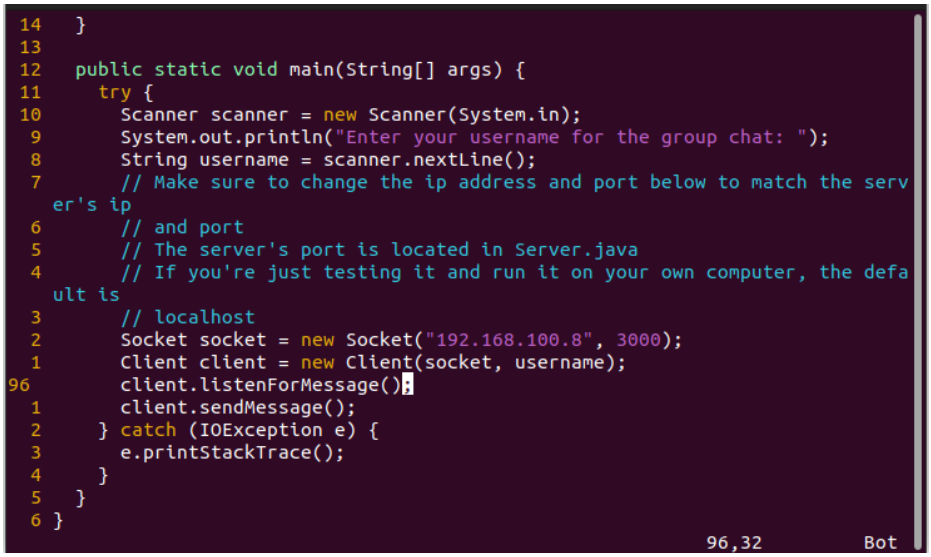
VII. How to Use It

In this section, we will provide explanations and illustrations regarding steps that need to be taken by the server and client so that the client could utilize the chat platform

1. Compile and run the Server.java file

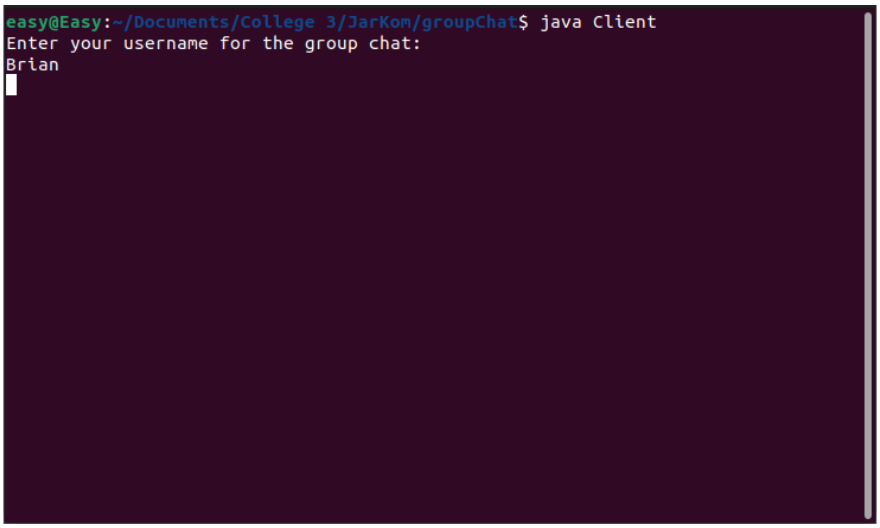


2. Put the local IP address and port of the server in the main method of Client.java

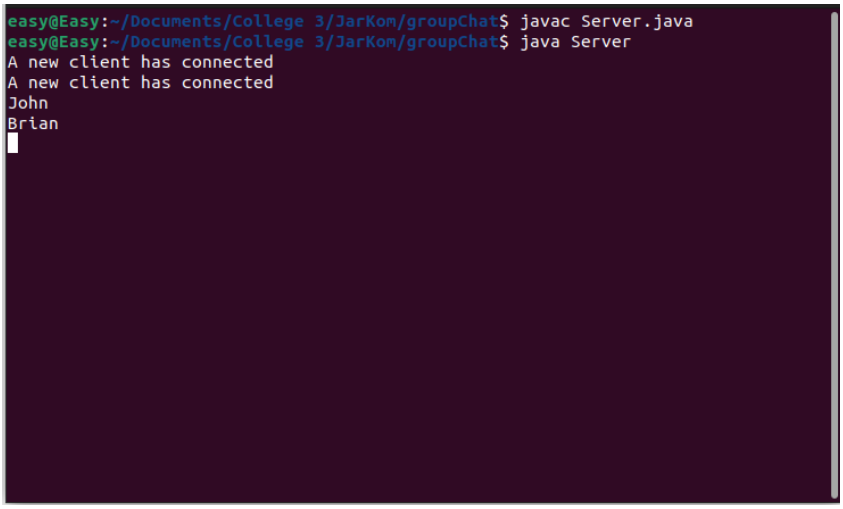


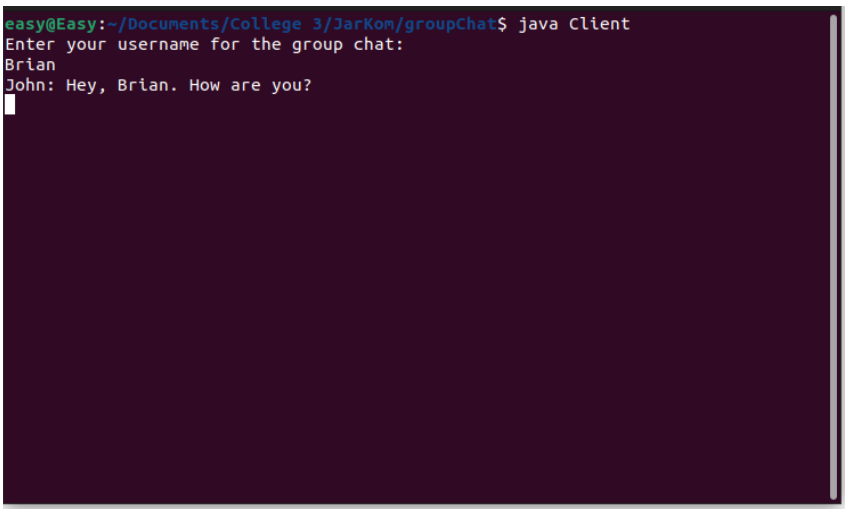
3. Compile and run the Client.java file
4. Enter the username for the client

5. Let other users connect to the server and start their client



6. Directly message another user by specifying the username anywhere in message





VIII. Conclusion

According to the result of the final project that we have made, it can be concluded that socket programming on the server side involves creating a server socket and binding it to a specific port, listening for incoming client connections, and accepting incoming client connections. Once a connection is established, the server can send and receive data through the socket. The server can also implement various features such as multiple client support, multithreading to handle multiple clients concurrently, and error handling to ensure the stability and reliability of the server. Furthermore, socket programming, especially on the server side, is considered a crucial aspect of network programming and is widely used in various applications such as web servers, file transfer protocols, and online multiplayer games.

References

[1] Xue, M; Zhu, C, "The Socket Programming and Software Design for Communication based on Client/Server", 2009 Pacific-Asia Conference on Circuits, Communications, and Systems, 2009, ISBN: 978-0-7695-3614-9

[2] Hanum S; Romie, O; Abdurahman, "Multi-network Transmission Using Socket Programming to Support Command and Control Systems", AISC, Vol. 1291, p. 59-68. 2021

[3] Maata, R.L.R., Cordova, R., Sudramurthy, BHalibas, A. "Design and Implementation of client-server based application using socket programming in a distributed computing environment.", 2017 IEEE international conference on computational intelligence and computing research (ICCIC), Coimbatore, pp. 1–4 (2017)

[4] Pankaj. 2022. Java Socket Programming - Socket Server, Client example. [online] digitalocean.com. Available at: <https://www.digitalocean.com/community/tutorials/java-socket-programming-server-client> (Accessed: November 6, 2022).

[5] Ashley, Apondi. 2021. Understanding Socket Programming in Java. [online] section.io. Available at: <https://www.section.io/engineering-education/socket-programming-in-java/> (Accessed: November 24, 2022}