# Name:Atharva Pawar

# Roll No:-9427 Exp:5 BE-COMPS A

**Aim: To implement group communication**

Lab Outcome:

Develop test and debug using Message-Oriented Communication or RPC/RMI based client-server programs.

Theory:

Group communication is a paradigm for multi-party communication that is based on the notion of groups as a main abstraction. A group is a set of parties that, presumably,want to exchange information in a reliable, consistent manner. For example:

* The participants of a message-based conferencing tool may constitute a group. Ideally, in order to have meaningful communication, each participant wants to receive all communicated messages from each other participant. Moreover, if one message isa response to another, the original message should be delivered before the response.(In this example, if two participants originate messages independently at about the same time, the order in which such independent messages are delivered is not important)
* The set of replicas of a fault-tolerant database server may constitute a group. Consider updating messages to the server. Since the contents of the database depend on the history of all update messages received, all updates must be delivered to all replicas. Furthermore, all updates must be delivered in the same order. Otherwise, inconsistencies may arise.

Group Communication Primitives

Group communication is implemented using middleware that provides two sets of primitives to the application:

* Multicast primitive (e.g., post): This primitive allows a sender to post a message to the entire group.
* Membership primitives (e.g., join, leave, query\_membership): These primitives allow a process to join or leave a particular group, as well as to query the group for the list of all current participants.

Three types of group communication:

* One to many (single sender and multiple receivers)

In this scheme, there are multiple receivers for a message sent by a single sender. The one-to-many scheme is also known as multicast communication. A special case of multicast communication is broadcast communication, in which the message is sent to all processors connected to a network.

* Many to one (multiple senders and single receiver)
* Multiple senders send messages to a single receiver.
* The single receiver may be selective or nonselective.
* A selective receiver specifies a unique sender; a message exchange takes place

only if that sender sends a message.

* A nonselective receiver specifies a set of senders, and if anyone sender in the

set sends a message to this receiver, a message exchange takes place - an important issue related to the many-to-one communication scheme is nondeterminism

* Many too many (multiple senders and multiple receivers) ➔

Multiple senders send messages to multiple receivers.

* An important issue related to many-to-many communication scheme is that of

ordered message delivery

* Ordered message delivery ensures that all messages are delivered to all receivers

in an order acceptable to the application. This property is needed by many applications for its correct functioning.

* Ordered message delivery requires message sequencing.
* The commonly used semantics for ordered delivery of multicast messages are

absolute ordering, consistent ordering, and causal ordering.

Steps to run the Single Client Server Communication application

1. Start GossipServer program. It will be ready to accept connections from the GossipClient.
2. On another terminal start the GossipClient program and send some message to GossipServer.
3. GossipServer will display the output.

Server.java

import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.net.ServerSocket; import java.net.Socket;

public class GossipServer {

public static void main(String[] args) { try {

ServerSocket serverSocket = new ServerSocket(12345);

System.out.println("GossipServer is running and ready to accept connections...");

while (true) {

Socket clientSocket = serverSocket.accept(); System.out.println("Client connected from: " +

clientSocket.getInetAddress().getHostAddress());

BufferedReader reader = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

String message;

while ((message = reader.readLine()) != null) { System.out.println("Message from client: " + message);

}

clientSocket.close();

}

} catch (IOException e) { e.printStackTrace();

}

}

}

Client.java

import java.io.IOException; import java.io.OutputStreamWriter; import java.io.PrintWriter; import java.net.Socket;

import java.util.Scanner;

public class GossipClient {

public static void main(String[] args) { try {

Socket socket = new Socket("localhost", 12345); PrintWriter writer = new PrintWriter(new

OutputStreamWriter(socket.getOutputStream()), true); Scanner scanner = new Scanner(System.in);

while (true) {

System.out.print("Enter message (or 'exit' to quit): "); String message = scanner.nextLine();

if (message.equalsIgnoreCase("exit")) { break;

}

writer.println(message);

}

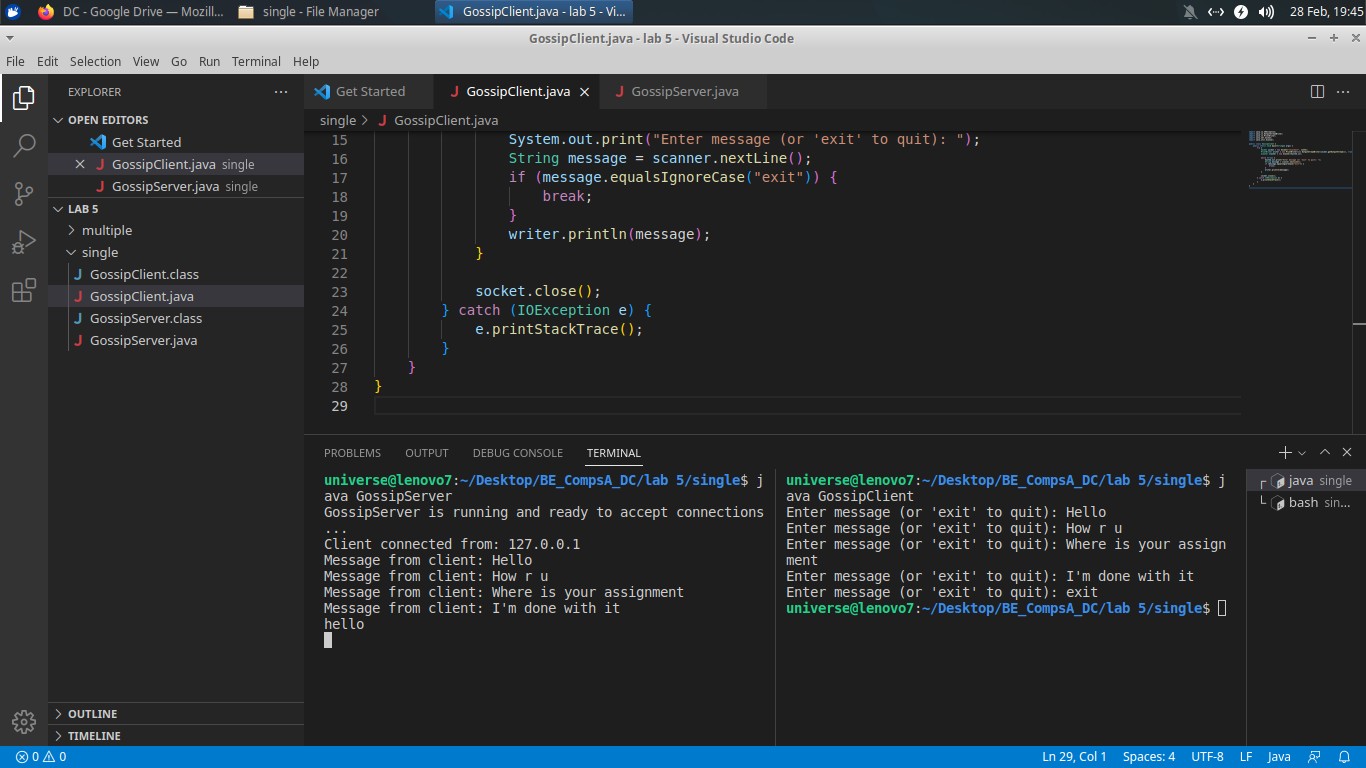
socket.close();

} catch (IOException e) { e.printStackTrace();

}

}

}

Output:

# Multi :

Steps to run the Multi Client Server Communication application

1. Start Server program. It will be ready to accept connections from the Master.
2. On another terminal start the Master program followed by the Slave and send some message from the Master to the Slave.
3. Multiple Slaves can be started to depict group communication.

Server.java

import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.net.ServerSocket;

import java.net.Socket; import java.util.HashSet; import java.util.Set;

public class MultiClientServer {

private static final int PORT = 12345;

private static Set<Socket> clients = new HashSet<>();

public static void main(String[] args) { try {

ServerSocket serverSocket = new ServerSocket(PORT);

System.out.println("Server is running and ready to accept connections...");

while (true) {

Socket clientSocket = serverSocket.accept(); clients.add(clientSocket); System.out.println("New client connected: " +

clientSocket);

ClientHandler clientHandler = new ClientHandler(clientSocket);

new Thread(clientHandler).start();

}

} catch (IOException e) { e.printStackTrace();

}

}

public static void broadcastMessage(String message) { for (Socket client : clients) {

try {

client.getOutputStream().write(message.getBytes());

} catch (IOException e) { e.printStackTrace();

}

}

}

static class ClientHandler implements Runnable {

private Socket clientSocket;

public ClientHandler(Socket clientSocket) { this.clientSocket = clientSocket;

}

@Override

public void run() { try {

BufferedReader reader = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

String message;

while ((message = reader.readLine()) != null) { System.out.println("Message from client: " + message); broadcastMessage(message);

}

} catch (IOException e) { e.printStackTrace();

}

}

}

}

MasterClient.java

import java.io.IOException; import java.io.OutputStream; import java.net.Socket; import java.util.Scanner;

public class MasterClient {

private static final String SERVER\_HOST = "localhost"; private static final int SERVER\_PORT = 12345;

public static void main(String[] args) {

try (Socket socket = new Socket(SERVER\_HOST, SERVER\_PORT)) { OutputStream outputStream = socket.getOutputStream(); Scanner scanner = new Scanner(System.in);

while (true) {

System.out.print("Enter message to send to slaves (or 'exit' to quit): ");

String message = scanner.nextLine(); if (message.equalsIgnoreCase("exit")) {

break;

}

outputStream.write(message.getBytes());

}

} catch (IOException e) { e.printStackTrace();

}

}

}

SlaveClient.java

import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.net.Socket;

public class SlaveClient {

private static final String SERVER\_HOST = "localhost"; private static final int SERVER\_PORT = 12345;

public static void main(String[] args) {

try (Socket socket = new Socket(SERVER\_HOST, SERVER\_PORT); BufferedReader reader = new BufferedReader(new

InputStreamReader(socket.getInputStream()))) {

while (true) {

String message = reader.readLine(); if (message == null) {

break;

}

System.out.println("Received message from server: " +

message);

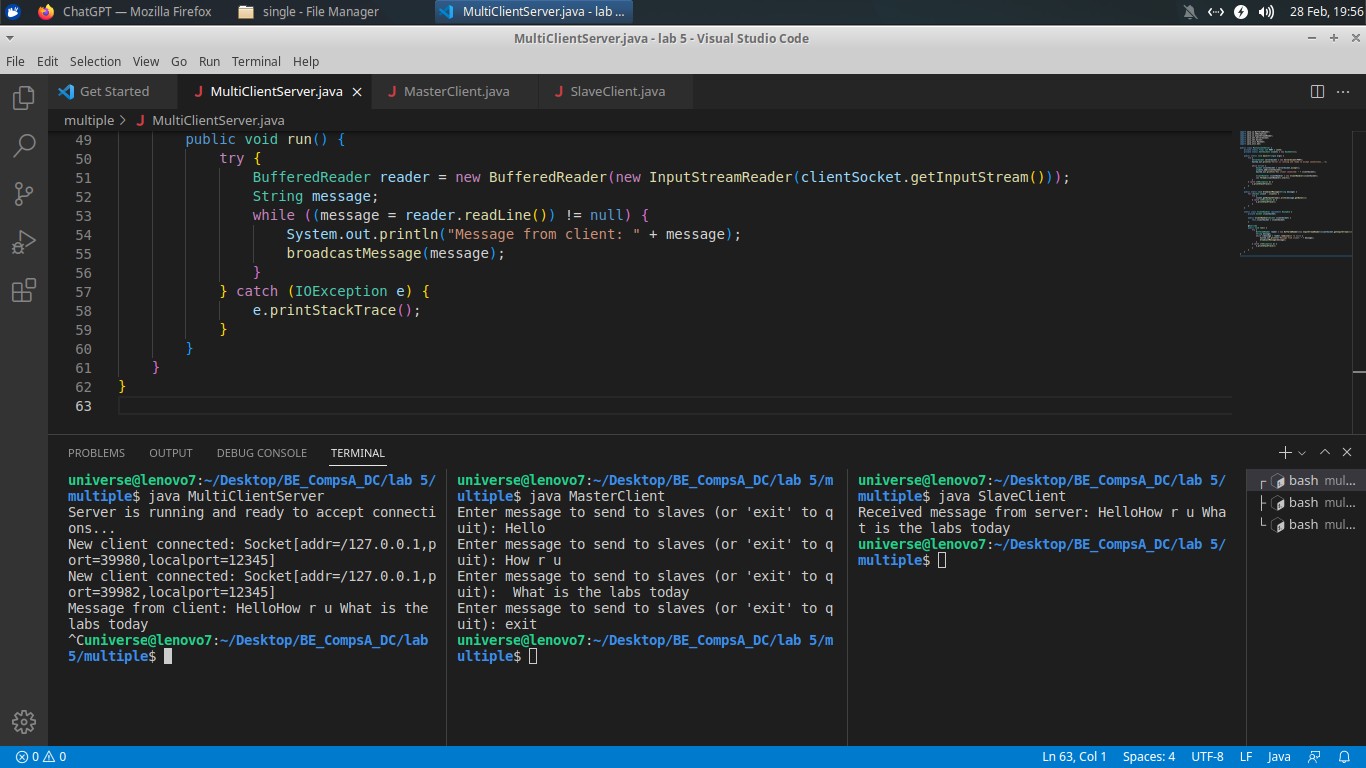
}

} catch (IOException e) { e.printStackTrace();

}

}

}

Output:

Conclusions :

1. Implemented group communication using Java.
2. Understood and learnt the three types of group communication.

Postlab Questions:

