**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) {

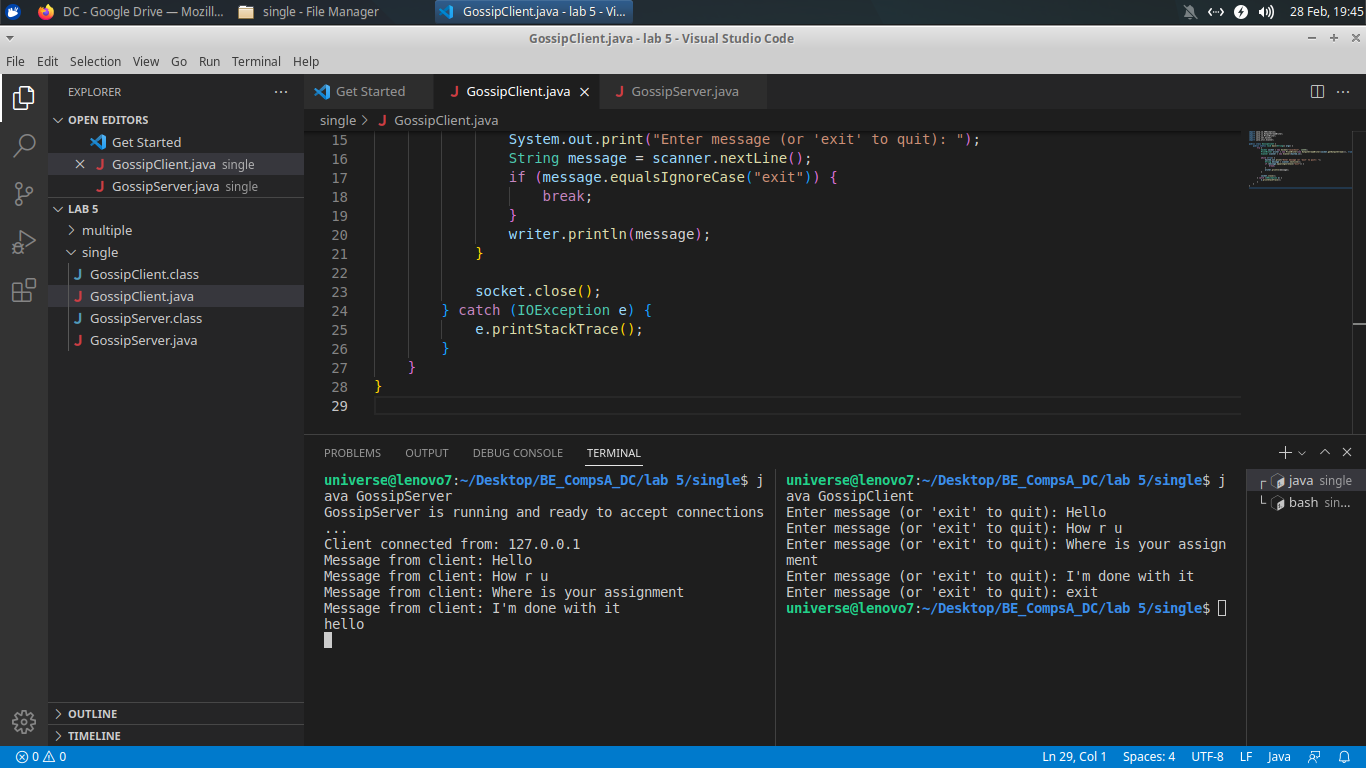
e.printStackTrace();

}

}

}

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) {

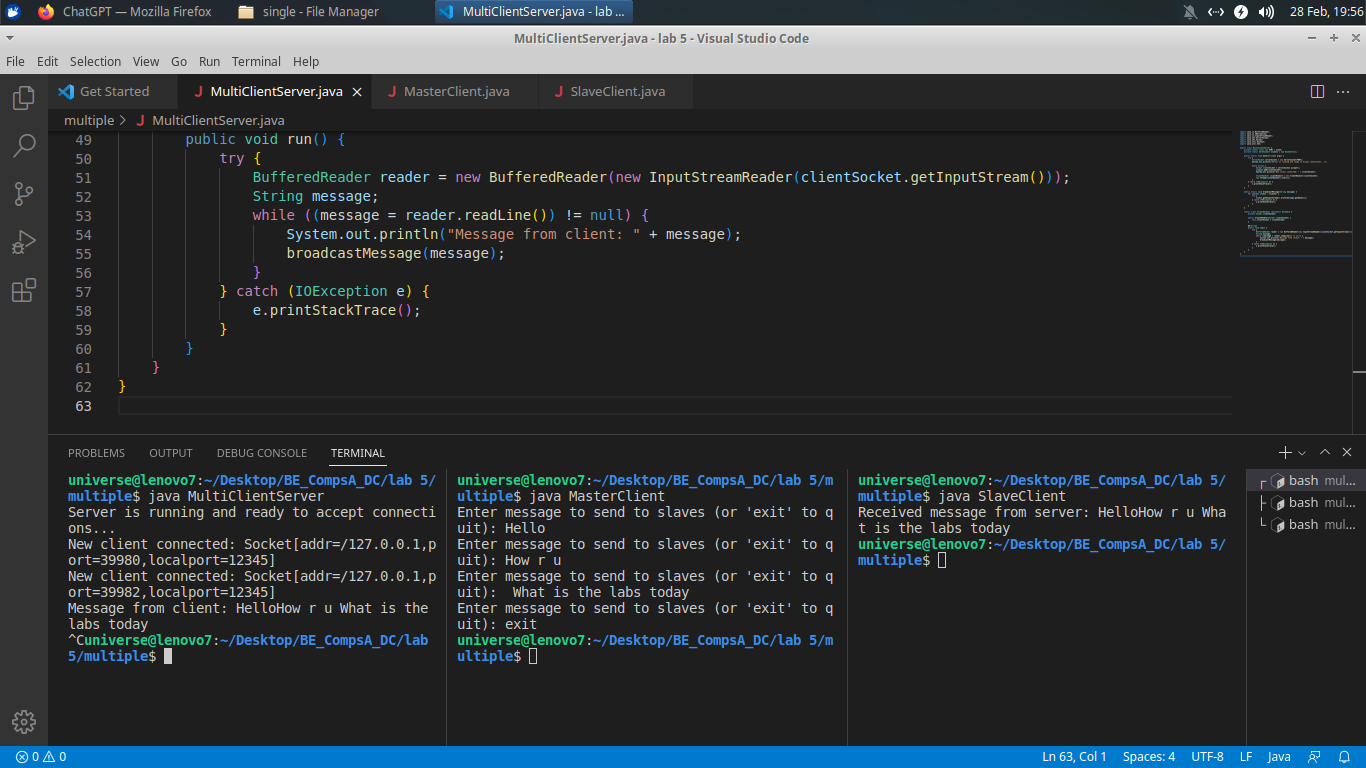
e.printStackTrace();

}

}

}

Output:



Conclusions :

1. Implemented group communication using Java.

2. Understood and learnt the three types of group communication.

Postlab Questions:

1.Explain group communication.

Group communication, also known as multicast communication, refers to the process of transmitting data from one sender to multiple receivers simultaneously within a network. In group communication, the sender sends a single message, and the message is received by all members of the group who are interested in receiving it.

Some key aspects of group communication:

1. Single Transmission, Multiple Receivers: In group communication, a single message is transmitted by the sender, and this message is received by multiple receivers simultaneously. This contrasts with unicast communication, where a separate message is sent to each individual receiver.

2. Membership and Group Management: Group communication typically involves defining and managing groups of receivers who are interested in receiving certain types of messages. Members join or leave groups dynamically, and there may be mechanisms in place to manage group membership and access control.

3. Efficiency: Group communication is often more efficient than sending individual messages to each receiver, especially when the number of receivers is large. By sending a single message to a group, network bandwidth and resources are conserved compared to sending multiple individual messages.

4. Scalability: Group communication can scale to accommodate a large number of receivers. As the number of receivers increases, the overhead associated with message delivery remains relatively constant, making group communication suitable for multicast applications such as live streaming, online gaming, and distributed simulations.

5. Reliability and Ordering: Group communication systems may provide reliability and ordering guarantees to ensure that messages are delivered to all group members and in the correct order. Techniques such as acknowledgments, retransmissions, and sequence numbers may be used to achieve these guarantees.

6. Applications: Group communication finds applications in various domains, including real-time collaboration tools, multiplayer online games, content distribution networks, distributed computing, and distributed systems management.

Overall, group communication enables efficient, scalable, and reliable communication between multiple participants or entities within a network. It facilitates collaboration, coordination, and information sharing among group members and is a fundamental building block in many distributed and networked systems.

2.Explain types of group communication.

Group communication can be categorized into several types based on various criteria. Here are some common types of group communication:

1. Unstructured Group Communication:

- In unstructured group communication, there is no predefined organization or hierarchy among group members.

- Communication flows freely among group members without any formal structure or rules.

- Examples include casual conversations among friends, social media interactions, and online forums where participants contribute without following a specific agenda.

2. Structured Group Communication:

- Structured group communication involves a predefined organization or hierarchy among group members.

- Communication follows specific rules, protocols, or formats, and there may be designated roles or positions within the group.

- Examples include formal meetings, conferences, workshops, and collaborative projects with assigned tasks and responsibilities.

3. Broadcast Communication:

- In broadcast communication, a single sender transmits a message to multiple receivers simultaneously.

- All receivers receive the same message, and there is no interaction or exchange of messages between receivers.

- Broadcast communication is commonly used in scenarios such as broadcasting television and radio programs, public announcements, and multicast messaging in computer networks.

4. Interactive Group Communication:

- Interactive group communication involves two-way communication between the sender and multiple receivers.

- Receivers have the ability to respond, interact, and engage with the sender or other group members.

- Examples include live discussions, video conferencing, online chat rooms, and multiplayer online games where participants communicate in real-time.

5. Asynchronous Group Communication:

- Asynchronous group communication allows group members to communicate and exchange messages at different times.

- Participants do not need to be present simultaneously, and messages can be posted, read, and responded to at the convenience of each member.

- Examples include email threads, discussion boards, and collaborative document editing tools where users can contribute asynchronously.

6. Synchronous Group Communication:

- Synchronous group communication requires all group members to be present and actively engaged in communication simultaneously.

- Communication occurs in real-time, and participants interact and exchange messages in a coordinated manner.

- Examples include live video conferences, instant messaging sessions, and virtual meetings where participants communicate synchronously.

7. Closed Group Communication:

- Closed group communication involves communication within a restricted or private group where membership is limited and controlled.

- Only authorized members have access to the communication channels, and messages are not accessible to outsiders.

- Examples include private chat groups, secure messaging platforms, and closed-door meetings within organizations.

8. Open Group Communication:

- Open group communication allows participation from anyone who wishes to join the group without restrictions on membership.

- Communication channels are open to the public, and messages are accessible to anyone interested in the topic or discussion.

- Examples include public forums, open-access mailing lists, and social media groups with no membership restrictions.