LAB 5

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 is a 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.

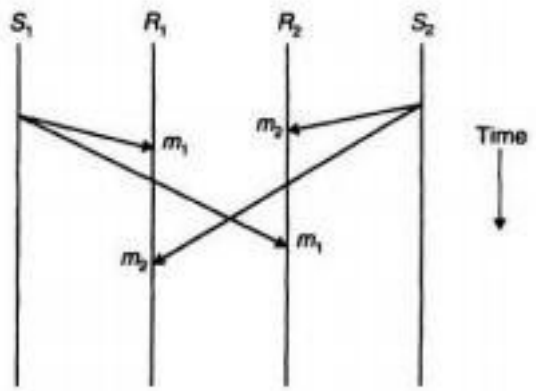


Fig. 3.14 No ordering constraint for message delivery.

➔ 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.

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.

(Code & Output:)

Single Client Server Communication Code

GossipClient.java

```
import java.io.*;
import java.net.*;
public class
GossipClient
{
  public static void main(String[] args) throws Exception
  {
    Socket sock = new Socket("127.0.0.1", 3000);

    BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));

    OutputStream ostream = sock.getOutputStream();
    PrintWriter pwrite = new PrintWriter(ostream, true);

                        InputStream istream = sock.getInputStream();
    BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));

    System.out.println("Start the chitchat, type and press Enter key");

    String receiveMessage, sendMessage;
   while(true)
    {
      sendMessage = keyRead.readLine();
     pwrite.println(sendMessage);
     pwrite.flush();
      if((receiveMessage = receiveRead.readLine()) != null)
      {
         System.out.println(receiveMessage);
      }
     }
   }
}
```

GossipServer.java

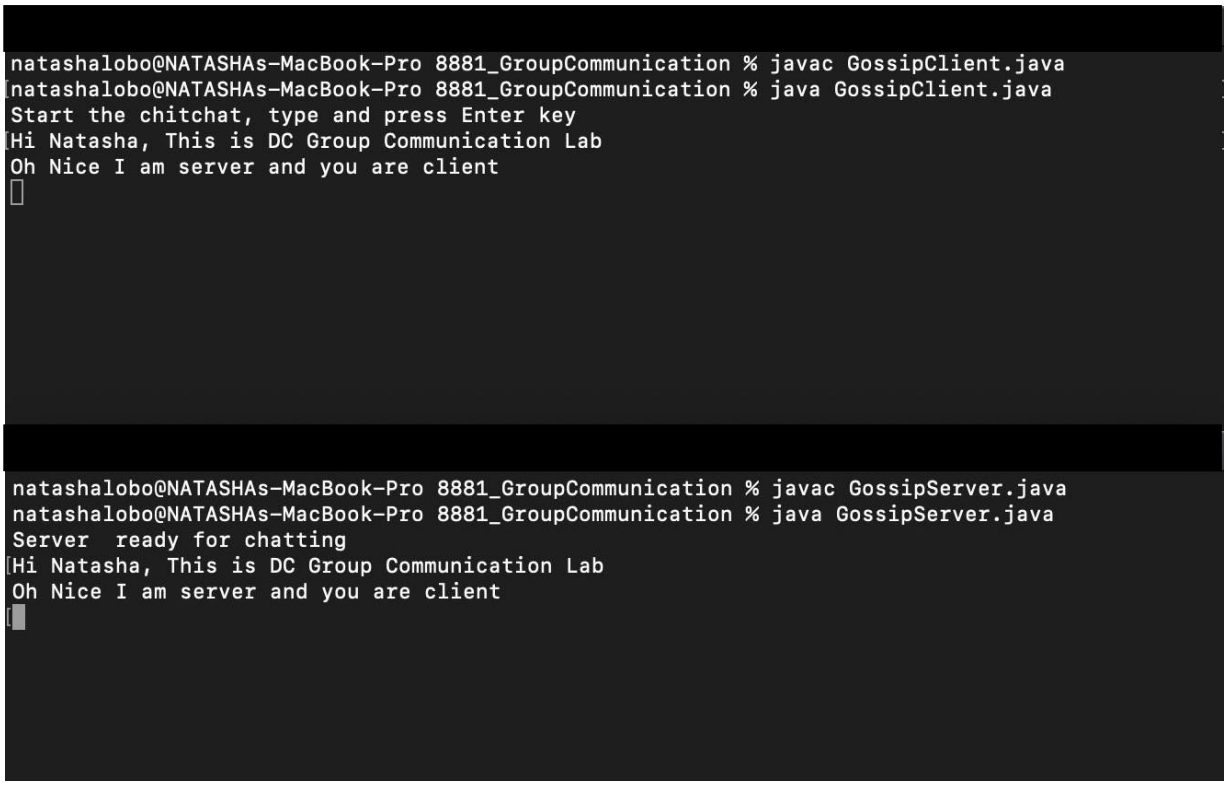
```
import java.io.*; import
java.net.*; public class
GossipServer
```

```
{
  public static void main(String[] args) throws Exception
  {
     ServerSocket sersock = new ServerSocket(3000);
     System.out.println("Server ready for chatting");
     Socket sock = sersock.accept( );
                        // reading from keyboard (keyRead object)
     BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));
                             // sending to client (pwrite object)
     OutputStream ostream = sock.getOutputStream();
     PrintWriter pwrite = new PrintWriter(ostream, true);

                        // receiving from server ( receiveRead object)
     InputStream istream = sock.getInputStream();
     BufferedReader receiveRead = new BufferedReader(new InputStreamReader(istream));

     String receiveMessage, sendMessage;
    while(true)
     {
       if((receiveMessage = receiveRead.readLine()) != null)
       {
          System.out.println(receiveMessage);
       }
       sendMessage = keyRead.readLine();
      pwrite.println(sendMessage);
      pwrite.flush();
     }
   }
}
```

Single Client Server Communication Output



Multi Client Server Communication

Code Server.java

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

public class Server {
   static ArrayList<ClientHandler> clients;

   public static void main(String args[]) throws Exception {
      // Server server = new Server();
      ServerSocket MyServer = new ServerSocket(8881);
      clients = new ArrayList<ClientHandler>();
      Socket ss = null;
      Message msg = new
      Message(); int count = 0;
      while (true) { ss = null; try {
            ss = MyServer.accept();
            DataInputStream din = new DataInputStream(ss.getInputStream());
            DataOutputStream dout = new DataOutputStream(ss.getOutputStream());
            ClientHandler chlr = new ClientHandler(ss, din, dout, msg);
            Thread t = chlr;
            if (count > 0)
               clients.add(chlr);
```

```
            count++;
            // System.out.println(threads);
            t.start();
          } catch (Exception E) {
            continue;
          }
      }
    }
}

class Message {
  String msg;

    public void set_msg(String msg) {
      this.msg = msg;
    }

    public void get_msg() {
      System.out.println("\nNEW GROUP MESSAGE: " + this.msg);
      for (int i = 0; i < Server.clients.size(); i++) {
        try {
            System.out.print("Client: " + Server.clients.get(i).ip + "; ");
            Server.clients.get(i).out.writeUTF(this.msg);
            Server.clients.get(i).out.flush();
         } catch (Exception e) {
            System.out.print(e);
         }
      }
    }
}

class ClientHandler extends Thread {
    DataInputStream in;
    DataOutputStream
   out; Socket socket; int
   sum;  float  res;
   boolean  conn;
   Message msg;
    String ip;
```

```
    public ClientHandler(Socket s, DataInputStream din, DataOutputStream dout,
Message msg) { this.socket = s; this.in = din; this.out = dout; this.conn = true;
this.msg = msg; this.ip = (((InetSocketAddress)
this.socket.getRemoteSocketAddress()).getAddress()).toString().replace("/", "");
    }

    public void run() {
      while (conn == true)
      { try {
             String input = this.in.readUTF();
             // System.out.println("From host " + this.ip + ':' +
            input); // String msg = "From host " + this.ip + ':' +
            input;  this.msg.set_msg(input);  this.msg.get_msg();
          } catch (Exception E) {
            conn = false;
            System.out.println(E);
          }
       }
       closeConn();
    }

    public void closeConn() {
      try {
          this.out.close();
          this.in.close();
          this.socket.close();
       } catch (Exception E) {
          System.out.println(E);
       }
    }
 }
```

Master.java

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

 public class Master {
    public static void main(String args[]) throws Exception {
       String send = "", r = "";
       Socket MyClient = new Socket("127.0.0.1", 8881);
       System.out.println("Connected as Master");
```

```
      DataInputStream din = new DataInputStream(MyClient.getInputStream());
      DataOutputStream dout = new
      DataOutputStream(MyClient.getOutputStream()); Scanner sc = new
      Scanner(System.in);  do {
          System.out.print("Message('close' to stop):
         "); send = sc.nextLine();
         dout.writeUTF(send);  dout.flush();
      } while
      (!send.equals("stop"));
      dout.close(); din.close();
      MyClient.close();
   }
}
```

Slave.java

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

public class Slave { public static void main(String args[])
  throws Exception {
      String r = "";
      Socket MyClient = new Socket("127.0.0.1", 8881);
      System.out.println("Connected as Slave");
      DataInputStream din = new
      DataInputStream(MyClient.getInputStream());  do {  r = din.readUTF();
          System.out.println("Master says: " + r);
      } while (!r.equals("stop"));
      din.close();
      MyClient.close();
   }
}
```

Multi Client Server Communication Output Server

```
natashalobo@NATASHAs-MacBook-Pro 8881_GrpComm % javac Server.java
natashalobo@NATASHAs-MacBook-Pro 8881_GrpComm % java Server

NEW GROUP MESSAGE: Group communication when only one user
Client: 127.0.0.1;
NEW GROUP MESSAGE: Group communication after one more slave joined.
Client: 127.0.0.1; Client: 127.0.0.1;
```

Master

```
natashalobo@NATASHAs-MacBook-Pro 8881_GrpComm % javac Master.java
natashalobo@NATASHAs-MacBook-Pro 8881_GrpComm % java Master
Connected as Master
Message('close' to stop): Group communication when only one user
Message('close' to stop): Group communication after one more slave joined.
Message('close' to stop):
```

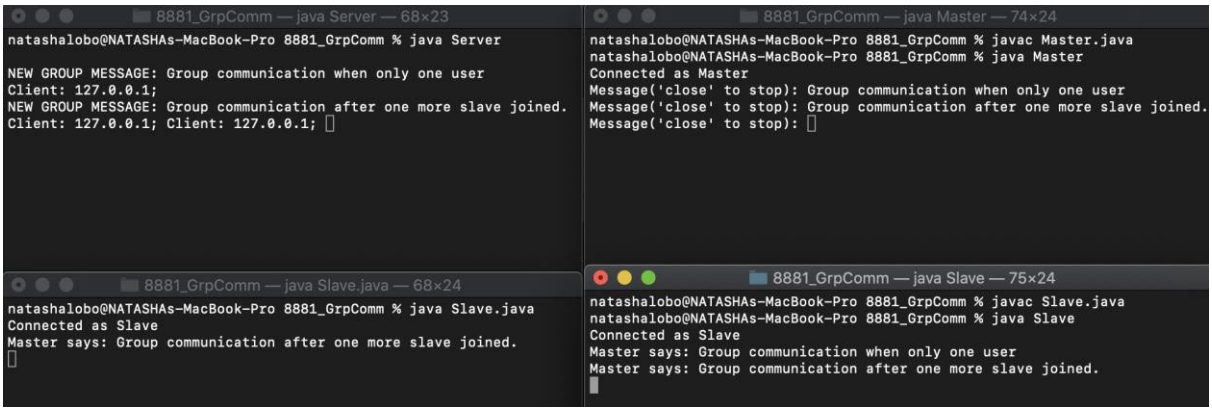
Slave

```
natashalobo@NATASHAs-MacBook-Pro 8881_GrpComm % javac Slave.java
natashalobo@NATASHAs-MacBook-Pro 8881_GrpComm % java Slave
Connected as Slave
Master says: Group communication when only one user
Master says: Group communication after one more slave joined.
```

Another Slave

```
natashalobo@NATASHAs-MacBook-Pro 8881_GrpComm % java Slave.java
Connected as Slave
Master says: Group communication after one more slave joined.
```

Group Communication Demonstration



Conclusions :

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

Postlab Questions:

1. Explain group communication.
2. Explain types of group communication.