Sanjivani Rural Education Society's

Sanjivani College of Engineering, Kopargaon – 423603

(An Autonomous Institute Affiliated to Savitribai Phule Pune University, Pune)

NAAC 'A' Grade Accredited,

Department of Information Technology

NBA UG Program Accredited

COURSE JOURNAL

Distributed System Laboratory

Course Code: IT 407

Class: B.Tech. I.T.

Course Pattern: 2020

Semester VII A.Y. 2023-2024

ASSIGNMENT NO. 1 A

Problem Statement:

To develop any distributed application through implementing client-server communication programs based on Java Sockets.

Tools / Environment:

Java Programming Environment, rmi registry, jdk 1.8, Eclipse IDE.

Related Theory:

Socket: In distributed computing, network communication is one of the essential parts of any system, and the socket is the endpoint of every instance of network communication. In Java communication, it is the most critical and basic object involved.

A socket is a handle that a local program can pass to the networking API to connect to another machine. It can be defined as *the terminal of a communication link through which two programs /processes/threads running on the network can communicate with each other. The TCP layer can easily identify the application location and access information through the port number assigned to the respective sockets.*

During an instance of communication, a client program creates a socket at its end and tries to connect it to the socket on the server. When the connection is made, the server creates a socket at its end and then server and client communication is established.

Java Socket programming is used for communication between the applications running on different JRE. Java Socket programming can be connection-oriented or connectionless. Socket and Server Socket classes are used for connection-oriented socket programming. Datagram Socket and Datagram Packet classes are used for connectionless socket programming.

• Socket class and methods:
– A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.
– public InputStream getInputStream(): returns the InputStream attached with this socket.
– public OutputStream getOutputStream():returns the OutputStream attached with this socket.
– public synchronized void close(): closes this socket

Designing the solution:

The **java.net** package provides classes to facilitate the functionalities required for networking. The **socket** class programmed through Java using this package has the capacity of being independent of the platform of execution; also, it abstracts the calls specific to the operating system on which it is invoked from other Java interfaces. The **Server Socket** class offers to observe connection invocations, and it accepts such invocations from different clients through another socket. High-level wrapper classes, such as **URL Connection** and **URL Encoder**, are more appropriate. If you want to establish a connection to the Web using a URL, then these classes will use the socket internally.

The **java.net** package provides support for the two common network protocols −

TCP − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

UDP − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

Socket programming for TCP:

The following steps occur when establishing a TCP connection between two computers using sockets

The server instantiates a Server Socket object, denoting which port number communication is to occur on.

The server invokes the accept() method of the Server Socket class. This method waits until a client connects to the server on the given port.

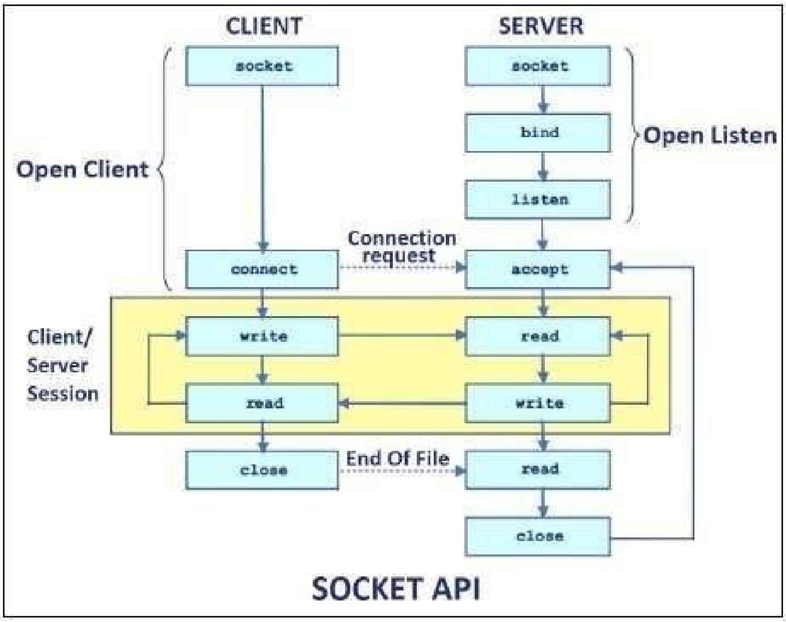
After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.

The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.

On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an Output Stream and an Input Stream. The client's Output Stream is connected to the server's Input Stream, and the client's Input Stream is connected to the server's Output Stream.

TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.



Socket programming for UDP:

UDP is used only when the entire information can be bundled into a single packet and there is no dependency on the other packet. Therefore, the usage of UDP is quite limited, whereas TCP is widely used in IP applications. UDP sockets are used where limited bandwidth is available, and the overhead associated with resending packets is not acceptable.

To connect using a UDP socket on a specific port, use the following code:

```
DatagramSocket udpSock = new DatagramSocket(3000);
```

A datagram is a self-contained, independent message whose time of arrival, confirmation of arrival over the network, and content cannot be guaranteed. At datagram Packet objects are used to send data over Datagram Socket. Every Datagram Packet object consists of a data buffer, a remote host to whom the data needs to be sent, and a port number on which the remote agent would be listened.

Implementing the solution:

Socket Programming for TCP:

Client Programming:

1. **Establish a Socket Connection:** The **java.net Socket** class represents a Socket. To open a socket:

   ```
   Socket socket = new Socket("127.0.0.1", 5000)
   ```

2. **Communication:** To communicate over a socket connection, streams are used to both input and output the data.

3. **Closing the connection:** The socket connection is closed explicitly once the message to server is sent.

Server Programming:

1. **Establish a Socket Connection:** To write a server application two sockets are needed. A Server Socket which waits for the client requests (when a client makes a new Socket()). A plain old Socket socket to use for communication with the client.

2. **Communication:** getOutputStream() method is used to send the output through the socket.

3. **Close the Connection:** After finishing, it is important to close the connection by closing the socket as well as input/output streams.

Socket Programming for UDP:

1. **Creation of Datagram Socket:-** First, a datagram Socket object is created to carry the packet to the destination and to receive it whenever the server sends any data.

2. **Creation of Datagram Packet:** In this step, the packet for sending/receiving data via a datagram Socket is created.

3. **Invoke a send() or receive() call on socket object**

TCP Socket Source code:

```
// Program for Server
import java.net.*;
import java.io.*;

public class Server
{
      //initialize socket and input stream
      private Socket         socket =
      null; private ServerSocket server =
      null; private DataInputStream in  =
      null;

      // constructor with port
      public Server(int port)
      {
            // starts server and waits for a connection
            try
            {
                  server = new ServerSocket(port);
                  System.out.println("Server started");

                  System.out.println("Waiting for a client ...");

                  socket = server.accept();
                  System.out.println("Client accepted");

                  // takes input from the client socket
                  in = new DataInputStream(new
                  BufferedInputStream(socket.getInputStream()));

                  String line = "";

                  // reads message from client until "Over" is sent
                  while (!line.equals("Over"))
                  {
                        try   line = in.readUTF();
                        {     System.out.println(line);



                        }
                        catch(IOException i)
                        {
                              System.out.println(i);
                        }
                  }
                  System.out.println("Closing connection");

                  // close connection
                  socket.close();
                  in.close();
            }
            catch(IOException i)
            {
                  System.out.println(i);
            }
      }
public static void main(String args[])
      {
            Server server = new Server(5000);
      }
```

```
}

// Program for a Client
import  java.net.*;
import java.io.*;

public class Client
{
    // initialize socket and input output
    streams private Socket socket    = null;
    private DataInputStream  input   = null;
    private DataOutputStream out     = null;

    // constructor to put ip address and port
    public Client(String address, int port)
    {
        // establish a connection
        try
        {
            socket = new Socket(address, port);
            System.out.println("Connected");

            // takes input from terminal
            input = new DataInputStream(System.in);

            // sends output to the socket
            out = new DataOutputStream(socket.getOutputStream());
        }
        catch(UnknownHostException u)
        {
            System.out.println(u);
        }
        catch(IOException i)
        {
            System.out.println(i);
        }

        // string to read message from input
        String line = "";

        // keep reading until "Over" is input
        while (!line.equals("Over"))
        {
            try line = input.readLine();
            {   out.writeUTF(line);


            }



            catch(IOException i)
            {
                System.out.println(i);
            }
        }
// close the connection
        try
        {
            input.close();
            out.close();
            socket.close();
```

```
        }
        catch(IOException i)
        {
            System.out.println(i);
        }
    }

    public static void main(String args[])
    {
        Client client = new Client("127.0.0.1", 5000);
    }
}
```

Compilation and Execution:

1. Create **Client.java**
2. Create **Server.java**
3. Compile both files using **javac *.java** , it will generate **.class** files
4. Open two terminal windows one for Server and another for Client.
5. Run the server first in one terminal window using **java Server** it will show - Server started Waiting for a client …
6. Run the client in another windows using **java Client**
7. Then you can start typing messages in the Client window, Which the Server simultaneously receives and shows
8. Sending "Over" closes the connection between the Client and the Server.

Output:

***FORM SERVER SIDE:**

dell@dell-Vostro-3546:~/Desktop$ cd ass1a

dell@dell-Vostro-3546:~/Desktop/ass1a$ ls

Client.java readme.odt Server.java

dell@dell-Vostro-3546:~/Desktop/ass1a$ javac *.java

Note: Client.java uses or overrides a deprecated API.

Note: Recompile with -Xlint: deprecation for details.

dell@dell-Vostro-3546:~/Desktop/ass1a$ java Server

Server started

Waiting for a client...

Client accepted

hi, **StudentName**

is here.. Over

Closing connection

dell@dell-Vostro-3546:~/Desktop/ass1a$

*****FOR CLIENTSIDE:**

dell@dell-Vostro-3546:~$ cd Desktop

dell@dell-Vostro-3546:~/Desktop$ cd ass1a

dell@dell-Vostro-3546:~/Desktop/ass1a$ javac *.java

Note: Client.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

dell@dell-Vostro-3546:~/Desktop/ass1a$ java Client

Connected

hi, **Student Name** is here.

Over

dell@dell-Vostro-3546:~/Desktop/ass1a$

//Program for Server(converting to Upper Case)

```
// A Java program for a Server
import java.net.*;
import java.io.*;

public class Server
{
	//initialize socket and input stream
	private Socket		socket =
	null; private ServerSocket server =
	null; private DataInputStream in =
	null;

	// constructor with port
	public Server(int port)
	{
		// starts server and waits for a connection
		try
		{
		   server = new ServerSocket(port);
		   System.out.println("Server started");

		   System.out.println("Waiting for a client ...");

		   socket = server.accept();
		   System.out.println("Client accepted");

		   // takes input from the client socket
		   in = new DataInputStream(
		     new BufferedInputStream(socket.getInputStream()));

			String line = "";
			// reads message from client until "Over" is sent
			while (!line.equals("Over"))
			{
				try
				{	line = in.readUTF();
					System.out.println(line.toUpperCase());

				}


				catch(IOException i)
				{
					System.out.println(i);
				}
			}
			System.out.println("Closing connection");
			// close connection
			socket.close();
			in.close();
		}
		catch(IOException i)
		{
			System.out.println(i);
		}
	}
	public static void main(String args[])
	{
		Server server = new Server(5000);
```

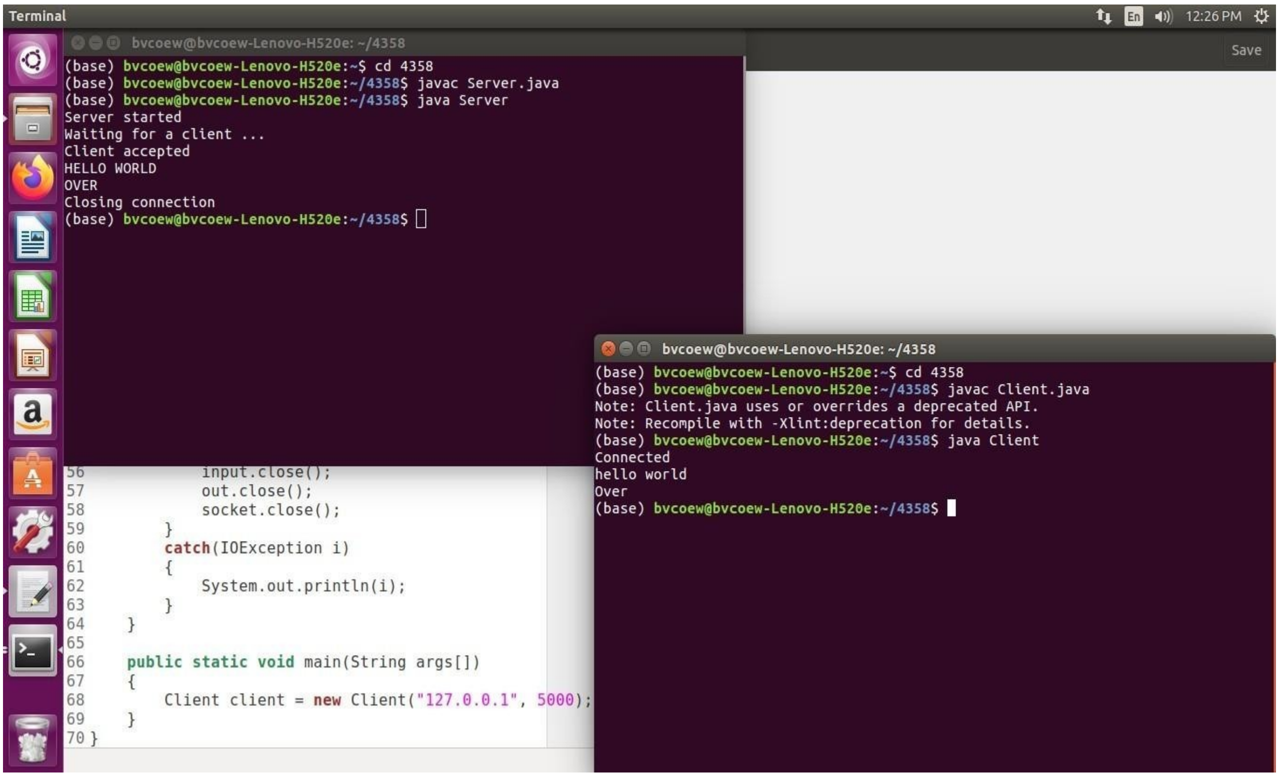
```
      }
}
```

//Program for Client

```
// A Java program for a Client
import java.net.*;
import java.io.*;

public class Client
{
    // initialize socket and input output
    streams private Socket socket    = null;
    private DataInputStream  input   = null;
    private DataOutputStream out     = null;

    // constructor to put ip address and port
    public Client(String address, int port)
    {
        // establish a connection
        try
        {
            socket = new Socket(address, port);
            System.out.println("Connected");

            // takes input from terminal
            input  = new DataInputStream(System.in);

            // sends output to the socket
            out    = new DataOutputStream(socket.getOutputStream());
        }
        catch(UnknownHostException u)
        {
            System.out.println(u);
        }
        catch(IOException i)
        {
            System.out.println(i);
        }

        // string to read message from input
        String line = "";
        // keep reading until "Over" is input
        while (!line.equals("Over"))
        {
            try
            {   line = input.readLine();
                out.writeUTF(line);

            }


            catch(IOException i)
            {
                System.out.println(i);
            }
        }
                // close the
                connection
                try
                {
                 input.close(); out.close(); socket.close();
```

```
            }
            catch(IOException i)
            {
                System.out.println(i);
            }
        }

        public static void main(String args[])
        {
            Client client = new Client("127.0.0.1", 5000);
        }
    }
```

OUTPUT:



UDP Socket Source code:

```
// A Java program for Server
import java.io.*;
import java.net.*;
import java.util.*;


public class QuoteServer {
    private DatagramSocket socket;
    private List<String> listQuotes = new ArrayList<String>();
    private Random random;

    public QuoteServer(int port) throws SocketException {
        socket = new DatagramSocket(port);
        random = new Random();
    }

    public static void main(String[] args) {
        if (args.length < 2) {
```

```
            System.out.println("Syntax: QuoteServer <file> <port>");
            return;
        }

        String quoteFile = args[0];
        int port = Integer.parseInt(args[1]);

        try {
            QuoteServer server = new QuoteServer(port);
            server.loadQuotesFromFile(quoteFile);
            server.service();
        } catch (SocketException ex) {
            System.out.println("Socket error: " + ex.getMessage());
        } catch (IOException ex) {
            System.out.println("I/O error: " + ex.getMessage());
        }
    }

    private void service() throws IOException {
        while (true) {
            DatagramPacket request=new DatagramPacket(new byte[1], 1);
            socket.receive(request);

            String quote = getRandomQuote();
            byte[] buffer = quote.getBytes();

            InetAddress clientAddress = request.getAddress();
            int clientPort = request.getPort();

            DatagramPacket response = new DatagramPacket(buffer,
            buffer.length, clientAddress, clientPort);
            socket.send(response);
        }
    }
private void loadQuotesFromFile(String quoteFile) throws IOException {
        BufferedReader reader = new BufferedReader(new
        FileReader(quoteFile));
        String aQuote;

        while ((aQuote = reader.readLine()) != null) {
            listQuotes.add(aQuote);
        }

        reader.close();
    }

    private String getRandomQuote() {
        int randomIndex = random.nextInt(listQuotes.size());
        String randomQuote = listQuotes.get(randomIndex);
        return randomQuote;
    }
}
```

```
// A Java program for a Client
import java.io.*;
import java.net.*;


public class QuoteClient {
```

```
public static void main(String[] args) {
    if (args.length < 2) {
        System.out.println("Syntax: QuoteClient <hostname> <port>");
        return;
    }

    String hostname = args[0];
    int port = Integer.parseInt(args[1]);

    try {
        InetAddress address = InetAddress.getByName(hostname);
        DatagramSocket socket = new DatagramSocket();

        while (true) {
             DatagramPacket request = new DatagramPacket(new byte[1],
             1, address, port);
             socket.send(request);

            byte[] buffer = new byte[512];
            DatagramPacket response = new DatagramPacket(buffer,
            buffer.length);
            socket.receive(response);

            String quote=new String(buffer, 0, response.getLength());

            System.out.println(quote);
            System.out.println();

            Thread.sleep(3000);
        }

    } catch (SocketTimeoutException ex) {
        System.out.println("Timeout error: " + ex.getMessage());
        ex.printStackTrace();
```

```
        } catch (IOException ex) {
            System.out.println("Client error: " + ex.getMessage());
            ex.printStackTrace();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
}
```

1. Create **QuoteClient.java**
2. Create **QuoteServer.java** & **Quotes.txt**
3. **C**ompile both files using **javac *.java** , it will generate **.class** files
4. Open two terminal windows one for Server and another for Client.
5. Run the server first in one terminal window using **java QuoteServer Quotes.txt 2525**
6. Run the client in another windows using **java QuoteClient localhost 2525**

Both the client and server run in an infinite loop as seen in client window. Press Ctrl + C to terminate.

Output:

*****SERVERSIDE:**

dell@dell-Vostro-3546:~$ cd Desktop
dell@dell-Vostro-3546:~/Desktop$ cd ass1b
dell@dell-Vostro-3546:~/Desktop/ass1b$ javac *.java
dell@dell-Vostro-3546:~/Desktop/ass1b$ java QuoteServer Quotes.txt 2525
dell@dell-Vostro-3546:~/Desktop/ass1b$

*****CLIENTSIDE:**

dell@dell-Vostro-3546:~$ cd Desktop
dell@dell-Vostro-3546:~/Desktop$ cd ass1b
dell@dell-Vostro-3546:~/Desktop/ass1b$ javac *.java
dell@dell-Vostro-3546:~/Desktop/ass1b$ java QuoteClient localhost 2525
Whether you think you can or you think you can't, you're right - Henry Ford

There are no traffic jams along the extra mile - Roger Staubach

Build your own dreams, or someone else will hire you to build theirs - Farrah Gray
Whether you think you can or you think you can't, you're right - Henry Ford
Whether you think you can or you think you can't, you're right - Henry Ford
dell@dell-Vostro-3546:~/Desktop/ass1b$
Conclusion:

In this assignment, we have learnt to develop a client/server distributed application relying on TCP and UDP protocol. We also learned about Java support through the socket API for TCP and UDP programming.

Dr.R.D.Chintamani **Dr.M.A.Jawale**

Practical Lab I/C **Head,Department of IT**

ASSIGNMENT NO. 1B

Problem Statement:

To develop any distributed application through implementing client-server communication programs based on Java RMI.

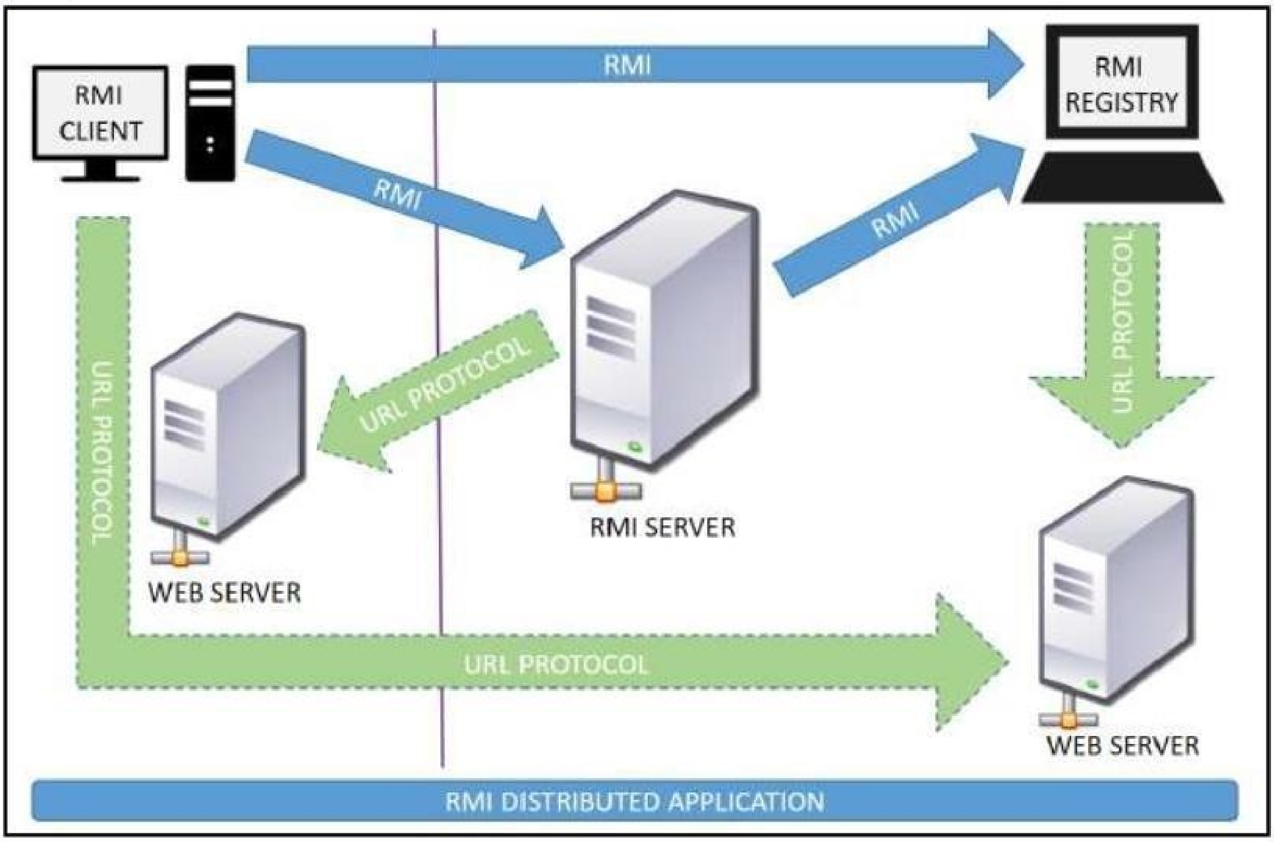
Tools / Environment:

Java Programming Environment, jdk 1.8, rmiregistry

Related Theory:

RMI provides communication between java applications that are deployed on different servers and connected remotely using objects called **stub** and **skeleton**. This communication architecture makesa distributed application seem like a group of objects communicating across a remote connection. These objects are encapsulated by exposing an interface, which helps access the private state and behavior of an object through its methods.

The following diagram shows how RMI happens between the RMI client and RMI server with the help of the RMI registry:



RMI REGISTRY is a remote object registry, a Bootstrap naming service that is used by **RMI SERVER** on the same host to bind remote objects to names. Clients on local and remote hosts then look up the remote objects and make remote method invocations.

Key terminologies of RMI:

The following are some of the important terminologies used in a Remote Method Invocation.

Remote object: This is an object in a specific JVM whose methods are exposed so they could be invoked by another program deployed on a different JVM.

Remote interface: This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.

RMI: This is a way of invoking a remote object's methods with the help of a remote interface. It can be carried with a syntax that is similar to the local method invocation.

Stub: This is a Java object that acts as an entry point for the client object to route anyoutgoing requests. It exists on the client JVM and represents the handle to the remote object. If any object invokes a method on the stub object, the stub establishes RMI by following thesesteps:

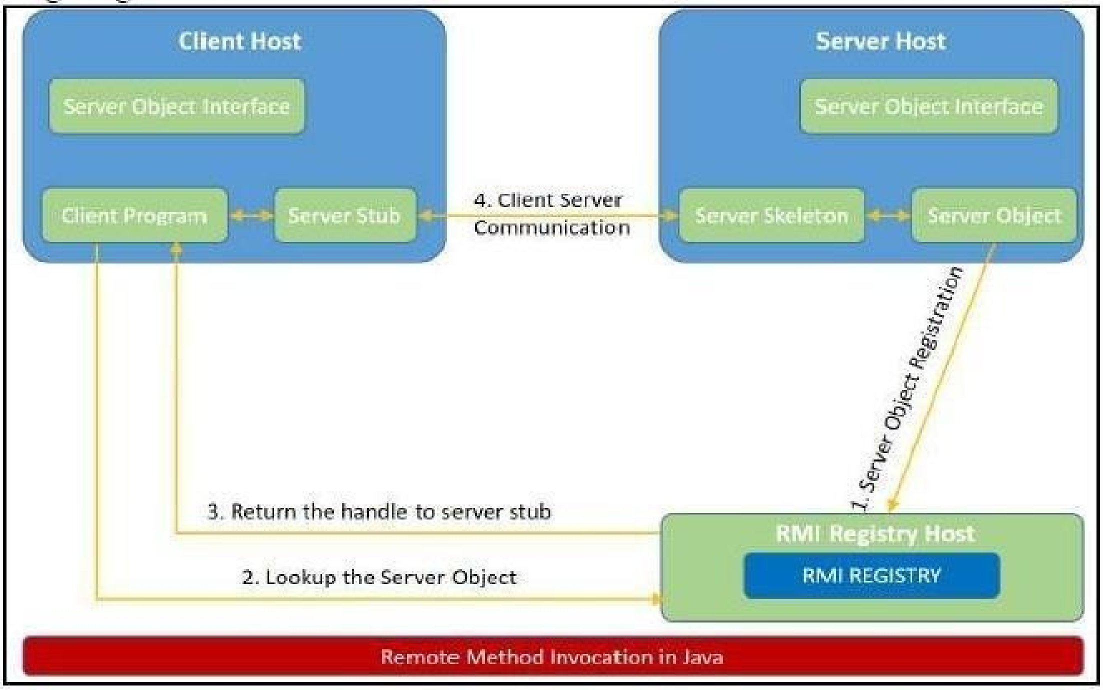
1. It initiates a connection to the remote machine JVM.
2. It marshals (write and transmit) the parameters passed to it via the remote JVM.
3. It waits for a response from the remote object and unmarshals (read) the returned value or exception, then it responds to the caller with that value or exception.

Skeleton: This is an object that behaves like a gateway on the server side. It acts as a remote object with which the client objects interact through the stub. This means that any requests coming from the remote client are routed through it. If the skeleton receives a request, it establishes RMI through these steps:

1. It reads the parameter sent to the remote method.
2. It invokes the actual remote object method.
3. It marshals (writes and transmits) the result back to the caller (stub).

The following diagram demonstrates RMI communication with stub and skeleton involved:

Designing the solution:



The essential steps that need to be followed to develop a distributed application with RMI are as follows:

1. Design and implement a component that should not only be involved in the distributed application, but also the local components.
2. Ensure that the components that participate in the RMI calls are accessible across networks.
3. Establish a network connection between applications that need to interact using the RMI.

1. **Remote interface definition**: The purpose of defining a remote interface is to declare the methods that should be available for invocation by a remote client. Programming the interface instead of programming the component implementation is an essential design principle adopted byall modern Java frameworks, including spring. In the same pattern, the definition of a remote interface takes importance in RMI design as well.

2. **Remote object implementation**: Java allows a class to implement more than one interface at a time. This helps remote objects implement one or more remote interfaces. The remote objectclass may have to implement other local interfaces and methods that it is responsible for. Avoid adding complexity to this scenario, in terms of how the arguments or return parameter values of such component methods should be written.

3. **Remote client implementation**: Client objects that interact with remote server objects can be written once the remote interfaces are carefully defined even after the remote objects are deployed.

Implementing the solution:

Consider building an application to perform diverse mathematical operations.

The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

1. Creating remote interface, implement remote interface, server-side and client-side program and Compile the code.

This application uses four source files. The first file, **AddServerIntf.java**, defines the remote interface that is provided by the server. It contains one method that accepts two **double** arguments and returns their sum. All remote interfaces must extend the **Remote** interface, which is part of **java.rmi**. **Remote** defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a **RemoteException**.

The second source file, **AddServerImpl.java**, implements the remote interface. The implementation of the **add()** method is straightforward. All remote objects must extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.

The third source file, **AddServer.java**, contains the main program for the server machine. Its primary function is **to update the RMI registry on that machine**. This is done by using the **rebind()** method of the **Naming** class (found in **java.rmi**). That method associates a name with an object reference. The first argument to the **rebind()** method is a string that names the server as “AddServer”. Its second argument is a reference to an instance of **AddServerImpl**.

The fourth source file, **AddClient.java**, implements the client side of this distributed application. **AddClient.java** requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the **rmi** protocol. The string includes the IP address or name of the server and the string "AddServer". The program then invokes the **lookup( )** method of the **Naming** class. This method accepts one argument, the **rmi** URL, and returns a reference to an object of type **AddServerIntf**. All remote method invocations can then be directed to this object. The program continues by displaying its arguments and then invokes the remote **add()** method. The sum is returned from this method and is then printed. Use **javac** to compile the four source files that are created.

2. Generate a Stub
Before using client and server, the necessary stub must be generated. In the context of RMI, a *stub* is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.

All of this information must be sent to the remote machine. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine. If a response must be returned to the client, the process works in reverse. **The serialization and deserialization facilities are also used if objects are returned to a client.**

To generate a stub the command is RMIcompiler is invoked as follows:
rmic AddServerImpl.
This command generates the file **AddServerImpl_Stub.class**.

3. Install Files on the Client and Server Machines
Copy **AddClient.class**, **AddServerImpl_Stub.class**, **AddServerIntf.class** to a directory on the client machine.
Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_Stub.class**, and **AddServer.class** to a directory on the server machine.

4. Start the RMI Registry on the Server Machine
Java provides a program called **rmiregistry**, which executes on the server machine. It maps names to object references. Start the RMI Registry from the command line.
start rmiregistry

5. Start the Server
The server code is started from the command line: java AddServer

The **AddServer** code instantiates **AddServerImpl** and registers that object with the name "AddServer".

6. Start the Client
The **AddClient** software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together: java AddClient 192.168.13.14 7 8

Source code:

// Program for AddClient

```
import  java.rmi.*;
public class AddClient {
      public static void main(String args[]) {
            try {
                  String addServerURL = "rmi://" + args[0] + "/AddServer";
                  AddServerIntf addServerIntf =
                     (AddServerIntf)Naming.lookup(addServerURL);
                  System.out.println("The first number is: " + args[1]);
                  double d1 = Double.valueOf(args[1]).doubleValue();
                  System.out.println("The second number is: " + args[2]);
                  double d2 = Double.valueOf(args[2]).doubleValue();
                  System.out.println("The sum is: " + addServerIntf.add(d1,
                     d2));
            }
            catch(Exception e) {
                  System.out.println("Exception: " + e);
            }
      }
}
```

//Program for AddServer

```
import java.net.*;
import java.rmi.*;
public class AddServer {
      public static void main(String args[]) {
            try {
                  AddServerImpl addServerImpl = new AddServerImpl();
                  Naming.rebind("AddServer", addServerImpl);
                  System.out.println("in server side");
            }
            catch(Exception e) {
                  System.out.println("Exception: " + e);
            }
      }
}
```

//Program for AddServerImpl

```
import  java.rmi.*;
import
java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject implements
AddServerIntf {
      public AddServerImpl() throws RemoteException {
      }
      public double add(double d1, double d2) throws RemoteException {
            return d1 + d2;
      }
}
```

//Program for AddServerIntf

```
import java.rmi.*;
public interface AddServerIntf extends Remote {
      double add(double d1, double d2) throws RemoteException;
}
```

Compilation and Executing the solution:

1. Create all java files and compile using **javac** command , it will generate **.class** files.
2. Generate stubs invoking **rmic AddServerImpl** it will generate **AddServerImpl_Stub.class** file.
3. Copy **AddClient.class**, **AddServerImpl_Stub.class**, and **AddServerIntf.class** to a directory on the client machine/folder.
4. Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_ Stub.class**, and **AddServer.class** to a directory on the server machine/folder.
5. Start the RMI Registry on the Server Machine using **rmiregistry**
6. In new terminal start the Server using **java AddServer**
7. In another new terminal **s**tart the Client **java AddClient servername/ip_address 8 9** where servername is first argument and 8 , 9 are second & third arguments respectively.
   e.g **java AddClient 127.0.0.1 8 9** for localhost (when client and server on same machine)
   e.g **java AddClient 172.16.86.80 8 9** (when client and server on different machine, specify IP address of server machine)

OUTPUT:

dell@dell-Vostro-3546:~$ cd Desktop

dell@dell-Vostro-3546:~/Desktop$ cd ass1b

dell@dell-Vostro-3546:~/Desktop/ass1b$ javac *.java

dell@dell-Vostro-3546:~/Desktop/ass1b$ rmic AddServerImpl

Warning: generation and use of skeletons and static stubs for JRMP

is deprecated. Skeletons are unnecessary, and static stubs have been

uperseded by dynamically generated stubs. Users are

encouraged to migrate away from using rmic to generate skeletons and static

stubs. See the documentation for java.rmi.server.UnicastRemoteObject.

dell@dell-Vostro-3546:~/Desktop/ass1b$ rmiregistry

*****SERVER SIDE:**

dell@dell-Vostro-3546:~$ cd Desktop

dell@dell-Vostro-3546:~/Desktop$ cd ass1b

dell@dell-Vostro-3546:~/Desktop/ass1b$ cd Server

dell@dell-Vostro-3546:~/Desktop/ass1b/Server$ javac *.java

dell@dell-Vostro-3546:~/Desktop/ass1b/Server$ java AddServer in server side

*****CLIENT SIDE:**

dell@dell-Vostro-3546:~$ cd Desktop

dell@dell-Vostro-3546:~/Desktop$ cd ass1b

dell@dell-Vostro-3546:~/Desktop/ass1b$ cd Client

dell@dell-Vostro-3546:~/Desktop/ass1b/Client$ javac *.java

dell@dell-Vostro-3546:~/Desktop/ass1b/Client$ java AddClient localhost 4 5

The first number is: 4

The second number is: 5

The sum is: 9.0

dell@dell-Vostro-3546:~/Desktop/ass1b/Client$

Conclusion:

In this assignment, we have studied how Remote Method Invocation (RMI) allows us to build Java applications that are distributed among several machines. Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows us to build distributed applications.

Dr.R.D.Chintamani	**Dr.M.A.Jawale**
Practical Lab I/C	**Head,Department of IT**

ASSIGNMENT NO. 2

Problem Statement:

To develop any distributed application using Message Passing Interface (MPI).

Tools / Environment:

Java Programming Environment, JDK1.8 or higher, MPI Library (mpi.jar), MPJ Express Software (Version 0.44).

Theory:

Message passing is a popularly renowned mechanism to implement parallelism in applications; it is also called MPI. The MPI interface for Java has a technique for identifying the user and helping in lower startup overhead. It also helps in collective communication and could be executed on both **shared memory and distributed systems**. MPJ is a familiar Java API for MPI implementation. mpiJava is the near flexible Java binding for MPJ standards. Currently developers can produce more efficient and effective parallel applications using messagepassing.

A basic prerequisite for message passing is a good communication API. Java comes with various ready-made packages for communication, notably an interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. The parallel computing world is mainly concerned with `Symmetric' communication, occurring in groups of interacting peers. This symmetric model of communication is captured in the successful Message Passing Interface standard (MPI).

Message-Passing Interface Basics:

Group is the set of processes that communicate with one another.
Communicator is the central object for communication in MPI. There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**.

Every MPI program must contain the preprocessor directive:

```
#include <mpi.h>
```

The `mpi.h` file contains the definitions and declarations necessary for compiling an MPI program.

`MPI_Init` initializes the execution environment for MPI. It is a “share nothing” modality in which the outcome of any one of the concurrent processes can in no way be influenced by the intermediate results of any of the other processes. Command has to be called before any other MPI call is made, and it is an error to call it more than a single time within the program.

`MPI_Finalize` cleans up all the extraneous mess that was first put into place by `MPI_Init`.

The principal weakness of this limited form of processing is that the processes on different nodes run entirely independent of each other. It cannot enable capability or coordinated computing. **To get the different processes to interact, the concept of communicators is needed.** MPI programs are made up of concurrent processes executing at the same time that in almost all cases are also communicating with each other. To do this, an object called the "communicator" is provided by MPI. Thus the user may specify any number of communicators within an MPI program, each with its own set of processes. "**`MPI_COMM_WORLD`**" communicator contains all the concurrent processes making up an MPI program.

The size of a communicator is the number of processes that makes up the particular communicator. The following function call provides the value of the number of processes of the specified communicator:

```
int MPI_Comm_size(MPI_Comm comm, int _size).
```

The function "`MPI_Comm_size`" required to return the number of processes; int size. `MPI_Comm_size(MPI_COMM_WORLD,&size)`; This will put the total number of processes in the MPI_COMM_WORLD communicator in the variable size of the process data context. Every process within the communicator has a unique ID referred to as its "rank". MPI system automatically and arbitrarily assigns a unique positive integer value, starting with 0, to all the processes within the communicator. The MPI command to determine the process rank is:

```
int MPI_Comm_rank (MPI_Comm comm, int _rank).
```

The `send` function is used by the source process to define the data and establish the connection of the message. The send construct has the following syntax:

```
int MPI_Send (void _message, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

The first three operands establish the data to be transferred between the source and destination processes. The first argument points to the message content itself, which may be a simple scalar or a group of data. The message data content is described by the next two arguments. The second operand specifies the number of data elements of which the message is composed. The third operand indicates the data type of the elements that make up the message.

The `receive` command (`MPI_Recv`) describes both the data to be transferred and the connection to be established. The MPI_Recv construct is structured as follows:

```
int MPI_Recv (void _message, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status _status)
```

The source field designates the rank of the process sending the message.

Communication Collectives: Communication collective operations can dramatically expand interprocess communication from point-to-point to n-way or all-way data exchanges.

The scatter operation: The scatter collective communication pattern, like broadcast, shares data of one process (the root) with all the other processes of a communicator. But in this case it partitions a set of data of the root process into subsets and sends one subset to each of the processes. Each receiving process gets a different subset, and there are as many subsets as there are processes. In this example the send array is A and the receive array is B. B is initialized to 0. The root process (process 0 here) partitions the data into subsets of length 1 and sends each subset to a separate process.

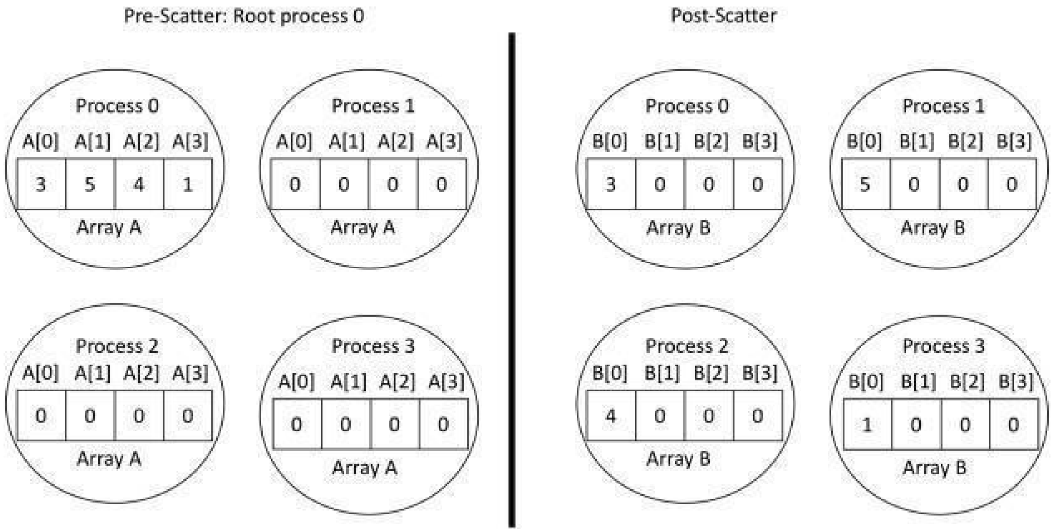


Fig: The Scatter Process

MPJ Express is an open source Java message passing library that allows application developers to write and execute parallel applications **for multicore processors and compute clusters / clouds.** The software is distributed under the MIT (a variant of the LGPL) license. MPJ Express is a message passing library that can be used by application developers to execute their parallel Java applications on compute clusters or network of computers.

MPJ Express is essentially a middleware that supports communication between individual processors of clusters. **The programming model followed by MPJ Express is Single Program**

Multiple Data (SPMD).

The multicore configuration is meant for users who plan to write and execute parallel Java applications using MPJ Express on their desktops or laptops which contains shared memory and multicore processors. In this configuration, users can write their message passing parallel application using MPJ Express and it will be ported automatically on multicore processors. We except that users can first develop applications on their laptops and desktops using multicore configuration, and then take the same code to distributed memory platforms.

Designing the solution:

While designing the solution, we have considered the multi-core architecture as per shown in the diagram below. The communicator has processes as per input by the user. MPI program will execute the sequence as per the supplied processes and the number of processor cores available for the execution.

Implementing the solution:

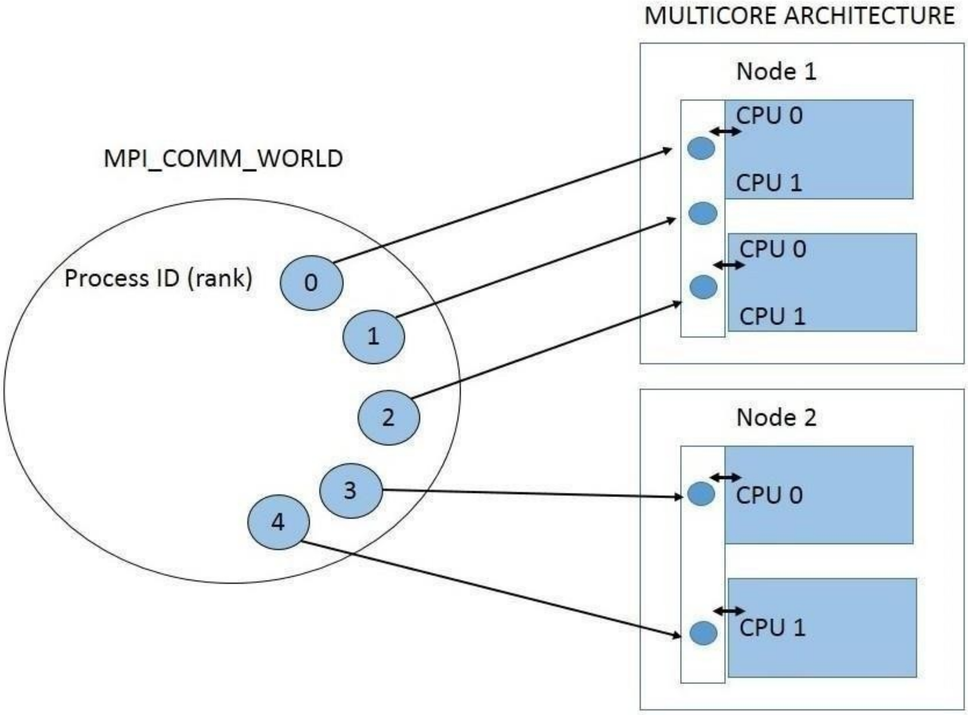


Fig: MPI on Multicore Architecture

Code:

```
import mpi.MPI;

public class ScatterGather {
	public static void main(String args[]){
		//Initialize MPI execution environment
		MPI.Init(args);
		//Get the id of the process
		int rank = MPI.COMM_WORLD.Rank();
		//total number of processes is stored in size
		int size = MPI.COMM_WORLD.Size();
		int root=0;
		//array which will be filled with data by root process
		int sendbuf[]=null;

		sendbuf= new int[size];

		//creates data to be scattered
		if(rank==root){
			sendbuf[0] = 10;
			sendbuf[1] = 20;
			sendbuf[2] = 30;
			sendbuf[3] = 40;

			//print current process number
			System.out.print("Processor "+rank+" has data: ");
			for(int i = 0; i < size; i++){
				System.out.print(sendbuf[i]+" ");
			}
			System.out.println();
		}
		//collect data in recvbuf
		int recvbuf[] = new  int[1];

		//following are the args of Scatter method
		//send, offset, chunk_count, chunk_data_type, recv, offset,
		//chunk_count, chunk_data_type, root_process_id
		MPI.COMM_WORLD.Scatter(sendbuf, 0, 1, MPI.INT, recvbuf,  0,
1, MPI.INT, root);
		System.out.println("Processor "+rank+" has data:
"+recvbuf[0]);
		System.out.println("Processor "+rank+" is doubling the
data");	recvbuf[0]=recvbuf[0]*2;
		//following are the args of Gather method
		//Object sendbuf, int sendoffset, int sendcount, Datatype
		//sendtype, Object recvbuf, int recvoffset, int  recvcount,
		//Datatype recvtype,
		//int root)
		MPI.COMM_WORLD.Gather(recvbuf, 0, 1, MPI.INT, sendbuf, 0,
1, MPI.INT, root);
		//display the gathered result
		if(rank==root){
			System.out.println("Process 0 has data: ");
			for(int i=0;i<4;i++){


				System.out.print(sendbuf[i]+ " ");
			}
		}
```

```
            //Terminate MPI execution
            environment MPI.Finalize();
        }
  }
```

For implementing the MPI program in multi-core environment, we need to **install MPJ express**
library.

Download MPJ Express (mpj.jar, Version 0.44) and unpack it.

Compiling and Executing:

1. **Set MPJ_HOME and PATH environment variables**: export
   MPJ_HOME=/path/to/mpj/
   export PATH=$MPJ_HOME/bin:$PATH
   (These above two lines can be added to ~/.bashrc)
2. **Compile ScatterGather.java:**
   javac -cp $MPJ_HOME/lib/mpj.jar
   ScatterGather.java (mpj.jar is inside lib folder in the
   downloaded MPJ Express)
3. **Execute:**
   $MPJ_HOME/bin/mpjrun.sh -np **4** ScatterGather
   Note: the number 4 above indicates the no. of processes.

Output:

```
bvcoew@bvcoew-Lenovo-Product:~$ export
MPJ_HOME=/home/bvcoew/Desktop/4346/2/mpj-v0_44
bvcoew@bvcoew-Lenovo-Product:~$ cd Desktop/4345/2
bvcoew@bvcoew-Lenovo-Product:~/Desktop/4346/2$ javac -cp
$MPJ_HOME/lib/mpj.jar ScatterGather.java
bvcoew@bvcoew-Lenovo-Product:~/Desktop/4346/2$  $MPJ_HOME/bin/mpjrun.sh -np
4 ScatterGather
bash: /home/bvcoew/Desktop/4345/2/mpj-v0_44/bin/mpjrun.sh: Permission
denied
bvcoew@bvcoew-Lenovo-Product:~/Desktop/4346/2$ chmod 777 mpj-
v0_44/bin/mpjrun.sh
bvcoew@bvcoew-Lenovo-Product:~/Desktop/4346/2$  $MPJ_HOME/bin/mpjrun.sh -np
4 ScatterGather
MPJ Express (0.44) is started in the multicore configuration
Processor 0 has data: 10 20 30 40
Processor 0 has data: 10
Processor 2 has data: 30


Processor 1 has data: 20
Processor 3 has data: 40
Processor 2 is doubling the
data Processor 1 is doubling
the data Processor 3 is
doubling the data Processor 0
is doubling the data Process 0
has data:
20 40 60 80
```

*****SYSTEM MONITOR*****

Conclusion:

There has been a large amount of interest in parallel programming using Java. mpj is an MPI binding with Java along with the support for multi core architecture so that user can develop the code on his/her own laptop or desktop. This is an effort to develop and run parallel programs according to MPI standard.

Dr.R.D.Chintamani **Dr.M.A.Jawale**

Practical Lab I/C **Head,Department of IT**

ASSIGNMENT NO. 3

Problem Statement:

To develop any distributed application with CORBA program using JAVA IDL.

Tools / Environment:

Java Programming Environment, JDK 1.8

Related Theory:

Common Object Request Broker Architecture (CORBA):
CORBA is an acronym for Common Object Request Broker Architecture. It is an open source, vendor-independent architecture and infrastructure developed by the **Object Management Group** (**OMG**) to integrate enterprise applications across a distributed network. CORBA specifications provide guidelines for such integration applications, based on the way they want to interact, irrespective of the technology; hence, all kinds of technologies can implement these standards using their own technical implementations.

When two applications/systems in a distributed environment interact with each other, there are quite a few unknowns between those applications/systems, including the technology they are developed in (such as Java/ PHP/ .NET), the base operating system they are running on (such as Windows/Linux), or system configuration (such as memory allocation). **They communicate mostly with the help of each other's network address or through a naming service.** Due to this, these applications end up with quite a few issues in integration, including content (message) mapping mismatches.

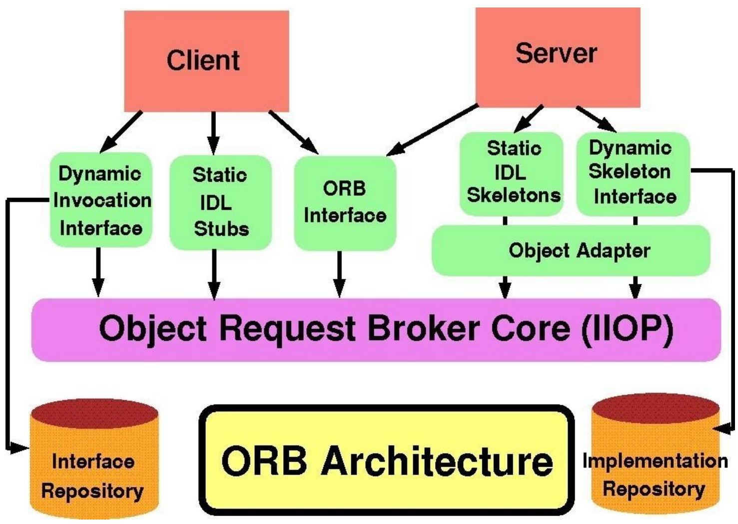
An application developed based on CORBA standards with standard **Internet Inter-ORB Protocol** (**IIOP**), irrespective of the vendor that develops it, should be able to smoothly integrate and operate with another application developed based on CORBA standards through the same or different vendor.
Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR & Benefits" maintain an object model with details of the organization, employees with demographic information, benefits, payroll, and deductions. They are only different in the way they handle the details, based on the country and region they are operating for. For each object type, similar to the HR & Benefits systems, we can define an interface using the **Interface Definition Language** (**OMG IDL**).

The contract between these applications is defined in terms of an interface for the server objects that the clients can call. This IDL interface is used by each client to indicate when they should call any particular method to marshal (read and send the arguments).

The target object is going to use the same interface definition when it receives the request from the client to unmarshal (read the arguments) in order to execute the method that was requested by the client operation. Again, during response handling, the interface definition is helpful to marshal (send from the server) and unmarshal (receive and read the response) arguments on the client side once received.

The IDL interface is a design concept that works with multiple programming languages including C, C++, Java, Ruby, Python, and IDL script. This is close to writing a program to an interface, a concept we have been discussing that most recent programming languages and frameworks, such as Spring. The interface has to be defined clearly for each object. The systems encapsulate the actual implementation along with their respective data handling and processing, and only the methods are available to the rest of the world through the interface. Hence, the clients are forced to develop their invocation logic for the IDL interface exposed by the application they want to connect to with the method parameters (input and output) advised by the interface operation.

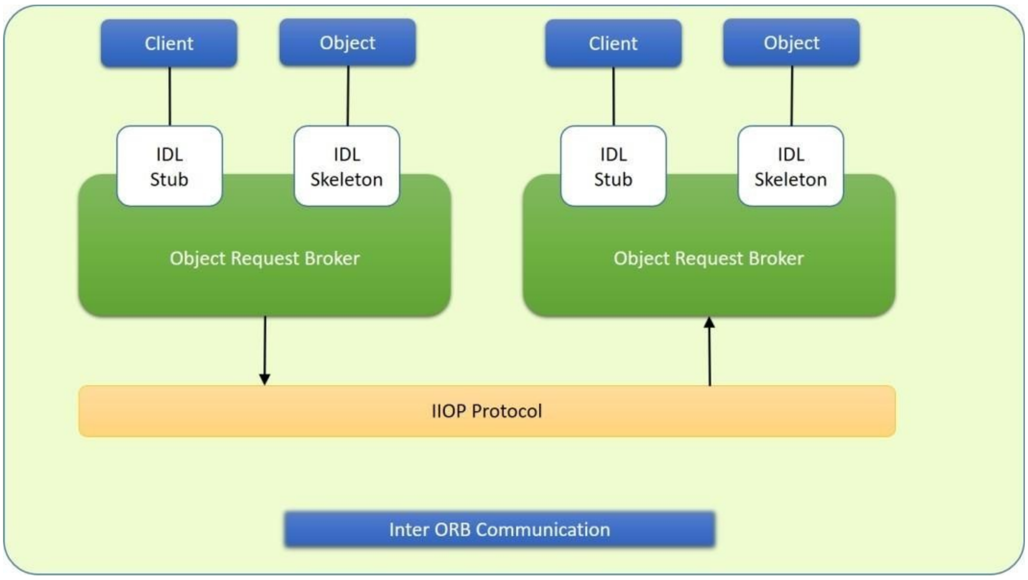
The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons, The objects are written (on the right) and a client for it (on the left), as represented in the diagram. The client and server use stubs and skeletons as proxies, respectively. The IDL interface follows a strict definition, and even though the client and server are implemented in different technologies, they should integrate smoothly with the interface definition strictly implemented.



In CORBA, each object instance acquires an object reference for itself with the electronic token identifier. Client invocations are going to use these object references that have the ability to figure out which ORB instance they are supposed to interact with. The stub and skeleton represent the client and server, respectively, to their counterparts. They help establish this communication through ORB and pass the arguments to the right method and its instance during the invocation.

Inter-ORB communication

The following diagram shows how remote invocation works for inter-ORB communication. It shows that the clients that interacted have created **IDL Stub** and **IDL Skeleton** based on **Object Request Broker** and communicated through **IIOP Protocol**.



To invoke the remote object instance, the client can get its object reference using a naming service. Replacing the object reference with the remote object reference, the client can make the invocation of the remote method with the same syntax as the local object method invocation. ORB keeps the responsibility of recognizing the remote object reference based on the client object invocation through a naming service and routes it accordingly.

Java Support for CORBA

CORBA complements the Java™ platform by providing a distributed object framework, services to support that framework, and interoperability with other languages. The Java platform complements CORBA by providing a portable, highly productive implementation environment, and a very robust platform. By combining the Java platform with CORBA and other key enterprise technologies, the Java Platform is the ultimate platform for distributed technology solutions.

CORBA standards provide the proven, interoperable infrastructure to the Java platform. IIOP (Internet Inter-ORB Protocol) manages the communication between the object components that power the system. The Java platform provides a portable object infrastructure that works on every major operating system. CORBA provides the network transparency, Java provides the implementation transparency. **An *Object Request Broker (ORB)* is part of the Java Platform. The ORB is a runtime component that can be used for distributed computing using IIOP communication. Java IDL is a Java API for interoperability and integration with CORBA.**

Java IDL included both a Java-based ORB, which supported IIOP, and the **IDL-to-Java compiler**, for generating client-side stubs and server-side code skeletons. J2SE v.1.4 includes an **Object Request Broker Daemon (ORBD), which is used to enable clients to transparently locate and invoke persistent objects on servers in the CORBA environment.**

When using the **IDL programming model,** the interface is everything! It defines the points of entry that can be called from a remote process, such as the types of arguments the called procedure will accept, or the value/output parameter of information returned. Using IDL, the programmer can make the entry points and data types that pass between communicating processes act like a standard language.

CORBA is a language-neutral system in which the argument values or return values are limited to what can be represented in the involved implementation languages. In CORBA, object orientation is limited only to objects that can be passed by reference (the object code itself cannot be passed from machine-to-machine) or are predefined in the overall framework. Passed and returned types must be those declared in the interface.

With RMI, the interface and the implementation language are described in the same language, so you don't have to worry about mapping from one to the other. Language-level objects (the code itself) can be passed from one process to the next. Values can be returned by their actual type, not the declared type. Or, you can compile the interfaces to generate IIOP stubs and skeletons which allow your objects to be accessible from other CORBA- compliant languages.

The IDL Programming Model:

The IDL programming model, known as Java™ IDL, consists of both the Java CORBA ORB and the `idlj` compiler that maps the IDL to Java bindings that use the Java CORBA ORB, as well as a set of APIs, which can be explored by selecting the `org.omg` prefix from the Package section of the API index.

Java IDL adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standards-based interoperability and connectivity. Runtime components include a Java ORB for distributed computing using IIOP communication.

To use the IDL programming model, define remote interfaces using OMG Interface Definition Language (IDL), then compile the interfaces using `idlj` compiler. When you run the `idlj` compiler over your interface definition file, it generates the Java version of the interface, as well as the class code files for the stubs and skeletons that enable applications to hook into the ORB.

Portable Object Adapter (POA): An *object adapter* is the mechanism that connects a request using an object reference with the proper code to service that request. The Portable Object Adapter, or POA, is a particular type of object adapter that is defined by the CORBA specification. The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities.

Designing the solution:

Here the design of how to create a complete CORBA (Common Object Request Broker Architecture) application using IDL (Interface Definition Language) to define interfaces and Java IDL compiler to generate stubs and skeletons. You can also create CORBA application by defining the interfaces in the Java programming language.

The server-side implementation generated by the `idlj` compiler is the *Portable Servant Inheritance Model*, also known as the POA (Portable Object Adapter) model. This document presents a sample application created using the default behavior of the `idlj` compiler, which uses a POA server-side model.

1. Creating CORBA Objects using Java IDL:

In order to distribute a Java object over the network using CORBA, one has to define its own CORBA-enabled interface and it implementation. This involves doing the following:

- Writing an interface in the CORBA Interface Definition Language
- Generating a Java base interface, plus a Java stub and skeleton class, using an IDL-to-Java compiler
- Writing a server-side implementation of the Java interface in

Java Interfaces in IDL are declared much like interfaces in Java.

Modules

Modules are declared in IDL using the module keyword, followed by a name for the module and an opening brace that starts the module scope. Everything defined within the scope of this module (interfaces, constants, other modules) falls within the module and is referenced in other IDL modules using the syntax *modulename*::x. e.g.

```
// IDL
module jen {
module corba {
interface NeatExample ...
};
};
```

Interfaces

The declaration of an interface includes an interface header and an interface body. The header specifies the name of the interface and the interfaces it inherits from (if any). Here is an IDL interface header:

```
Interface PrintServer: Server {...
```

This header starts the declaration of an interface called `PrintServer` that inherits all the methods and data members from the `Server` interface.

Data members and methods

The interface body declares all the data members (or attributes) and methods of an interface. Data members are declared using the `attribute` keyword. At a minimum, the declaration includes a name and a type.

```
readonly attribute string myString;
```

The method can be declared by specifying its name, return type, and parameters, at a minimum.

```
string parseString(in string buffer);
```

This declares a method called `parseString()` that accepts a single `string` argument and returns a `string` value.

A complete IDL example

Here's a complete IDL example that declares a module within another module, which itself contains several interfaces:

```
module OS {
     module services {
          interface Server {
               readonly attribute string serverName;
               boolean init(in string sName);
          };
          interface Printable {
               boolean print(in string header);
          };
          interface PrintServer : Server {
               boolean printThis(in Printable p);
          };
     };
};
```

The first interface, `Server`, has a single read-only `string` attribute and an `init()` method that accepts a `string` and returns a `boolean`. The `Printable` interface has a single `print()` method that accepts a string header. Finally, the `PrintServer` interface extends the `Server` interface and adds a `printThis()` method that accepts a `Printable` object and returns a `boolean`. In all cases, we've declared the method arguments as input-only (i.e., pass-by-value), using the `in` keyword.

2. Turning IDL into Java

Once the remote interfaces in IDL are described, you need to generate Java classes that act as a starting point for implementing those remote interfaces in Java using an IDL-to- Java compiler. Every standard IDL-to-Java compiler generates the following 3 Java classes from an IDL interface:

- A Java interface with the same name as the IDL interface. This can act as the basis for a Java implementation of the interface (but you have to write it, since IDL doesn't provide any details about method implementations).

- A *helper* class whose name is the name of the IDL interface with "Helper" appended to it (e.g., `ServerHelper`). The primary purpose of this class is to provide a static `narrow()` method that can safely cast CORBA `Object` references to the Java interface type. The helper class also provides other useful static methods, such as `read()` and `write()` methods that allow you to read and write an object of the corresponding type using I/O streams.

- A *holder* class whose name is the name of the IDL interface with "Holder" appended to it (e.g., `ServerHolder`). This class is used when objects with this interface are used as `out` or `inout` arguments in remote CORBA methods. Instead of being passed directly into the remote method, the object is wrapped with its holder before being passed. When a remote method has parameters that are declared as `out` or `inout`, the method has to be able to update the argument it is passed and return the updated value. The only way to guarantee this, even for primitive Java data types, is to force `out` and `inout` arguments to be wrapped in Java holder classes, which are filled with the output value of the argument when the method returns.

The `idltoj` tool generate 2 other classes:

- **A client *stub* class,** called _*interface-name*`Stub`, that acts as a client-side implementation of the interface and knows how to convert method requests into ORB requests that are forwarded to the actual remote object. The stub class for an interface named `Server` is called
`_ServerStub`.

- **A server *skeleton* class,** called _*interface-name*`ImplBase`, that is a base class for a serverside implementation of the interface. The base class can accept requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named `Server` is called `_ServerImplBase`.

So, in addition to generating a Java mapping of the IDL interface and some helper classes for the Java interface, the *idltoj* compiler also creates subclasses that act as an interface between a CORBA client and the ORB and between the server-side implementation and the ORB.

This creates the five Java classes: a Java version of the interface, a helper class, a holder class, a client stub, and a server skeleton.

3. Writing the Implementation

The IDL interface is written and generated the Java interface and support classes for it, including the client stub and the server skeleton. Now, concrete server-side implementations of all of the methods on the interface needs to be created.

Implementing the solution:

Here, we are demonstrating the "Hello World" Example. **To create this example, create a directory named hello/ where you develop sample applications and create the files in this directory.**

1. Defining the Interface (Hello.idl)

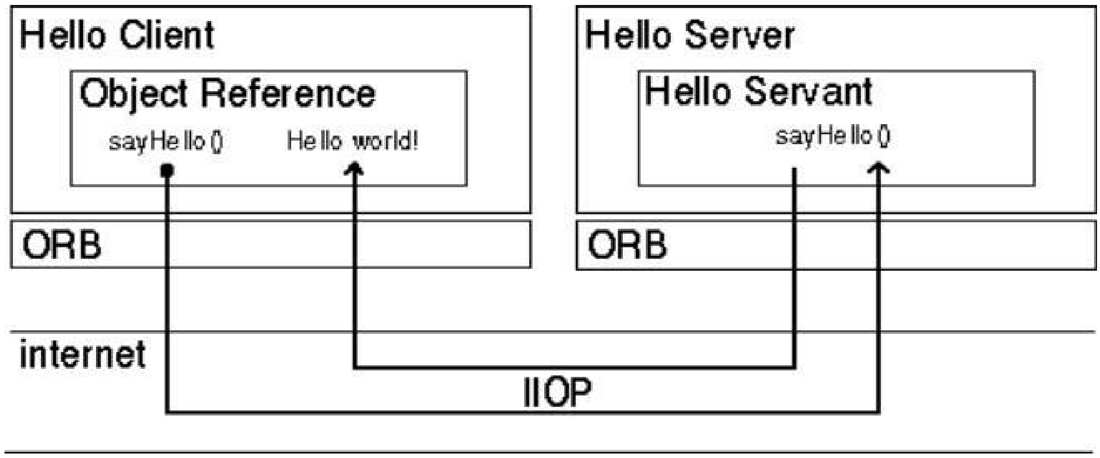
The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL).To complete the application, you simply provide the server **(HelloServer.java)** and client **(HelloClient.java)** implementations.

2. Implementing the Server (HelloServer.java)

The example server consists of two classes, the servant and the server. The servant, `HelloImpl`, is the implementation of the `Hello` IDL interface; each `Hello` instance is implemented by a `HelloImpl` instance. The servant is a subclass of `HelloPOA`, which is generated by the `idlj` compiler from the example IDL. The servant contains one method for each IDL operation, in this example, the `sayHello()` and `shutdown()` methods. Servant methods are just like ordinary Java methods; the extra code to deal with the ORB, with marshaling arguments and results, and so on, is provided by the skeleton.

The `HelloServer` class has the server's `main()` method, which:

- Creates and initializes an ORB instance
- Gets a reference to the root POA and activates the `POAManager`
- Creates a servant instance (the implementation of one CORBA `Hello` object) and tells the ORB about it
- Gets a CORBA object reference for a naming context in which to register the new CORBA object
- Gets the root naming context
- Registers the new object in the naming context under the name "Hello"
- Waits for invocations of the new object from the client.



3. Implementing the Client Application (HelloClient.java)

The example application client that follows:

- Creates and initializes an ORB
- Obtains a reference to the root naming context
- Looks up "Hello" in the naming context and receives a reference to that CORBA object
- Invokes the object's `sayHello()` and `shutdown()` operations and prints the result.

Building and executing the solution:

The Hello World program lets you learn and experiment with all the tasks required to develop almost any CORBA program that uses static invocation, which uses a client stub for the invocation and a server skeleton for the service being invoked and is used when the interface of the object is known at compile time.

This example requires a naming service, which is a CORBA service that allows **CORBA objects** to be named by means of binding a name to an object reference. The **name binding** may be stored in the naming service, and a client may supply the name to obtain the desired object reference. The two options for Naming Services with Java include **orbd**, a daemon process containing a Bootstrap Service, a Transient Naming Service,

To run this client-server application on the development machine:

1. Change to the directory that contains the file `Hello.idl`.
2. Run the IDL-to-Java compiler, `idlj`, on the IDL file to create stubs and skeletons.

This step assumes that you have included the path to the `java/bin` directory in your path.

idlj -fall Hello.idl

You must use the `-fall` option with the `idlj` compiler to generate both client and serverside bindings. This command line will generate the default server-side bindings, which assumes the POA Inheritance server-side model.

The files generated by the idlj compiler for Hello.idl, with the -fall command line option, are:

- HelloPOA.java:

This abstract class is the stream-based server skeleton, providing basic CORBA functionality for the server. It extends org.omg.PortableServer.Servant, and implements the InvokeHandler interface and the HelloOperations interface. The server class HelloImpl extends HelloPOA.

- HelloStub.java:

This class is the client stub, providing CORBA functionality for the client. It extends org.omg.CORBA.portable.ObjectImpl and implements the Hello.java interface.

- Hello.java:

This interface contains the Java version of IDL interface written. The Hello.java interface extends org.omg.CORBA.Object, providing standard CORBA object functionality. It also extends the HelloOperations interface and org.omg.CORBA.portable.IDLEntity.

- HelloHelper.java

This class provides auxiliary functionality, notably the narrow() method required to cast CORBA object references to their proper types. **The Helper class is responsible for reading and writing the data type to CORBA streams, and inserting and extracting the data type from AnyS.** The Holder class delegates to the methods in the Helper class for reading and writing.

- HelloHolder.java

This final class holds a public instance member of type Hello. Whenever the IDL type is an out or an inout parameter, the Holder class is used. It provides operations for org.omg.CORBA.portable.OutputStream and org.omg.CORBA.portable.InputStream arguments, which CORBA allows, but which do not map easily to Java's semantics. The Holder class delegates to the methods in the Helper class for reading and writing. It implements org.omg.CORBA.portable.Streamable.

- HelloOperations.java

This interface contains the methods sayHello() and shutdown(). The IDL-to-Java mapping puts all of the operations defined on the IDL interface into this file, which is shared by both the stubs and skeletons.

3. Compile the .java files, including the stubs and skeletons (which are in the directory directory HelloApp). This step assumes the java/bin directory is included in your path.

```
javac *.java HelloApp/*.java
```

4. Start `orbd.`

To start `orbd` from a UNIX command shell, enter:

```
orbd -ORBInitialPort 1050&
```

Note that `1050` is the port on which you want the name server to run. The `-ORBInitialPort` argument is a required command-line argument.

5. Start the `HelloServer`:

To start the `HelloServer` from a UNIX command shell, enter:

```
java HelloServer -ORBInitialPort 1050 -ORBInitialHost localhost&
```

You will see `HelloServer` ready and waiting... when the server is started.

6. Run the client application:

```
java HelloClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

When the client is running, you will see a response such as the following on your terminal:

```
Obtained a handle on server object: IOR: (binary code)
Hello World! HelloServer exiting...
After completion kill the name server (orbd).
```

Code for ReverseString:

ReverseServer.java

```
import ReverseModule.Reverse;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;

class ReverseServer
{
public static void main(String[] args)
{
try
{
// initialize the ORB
      org.omg.CORBA.ORB orb=org.omg.CORBA.ORB.init(args,null);

// initialize the BOA/POA
            POA rootPOA=
            POAHelper.narrow(orb.resolve_initial_references("RootPOA"
            ));
rootPOA.the_POAManager().activate();

// creating the calculator object
ReverseImpl rvr = new ReverseImpl();
```

```
// get the object reference from the servant class
          org.omg.CORBA.Object
          ref=rootPOA.servant_to_reference(rvr);

System.out.println("Step1");
Reverse h_ref = ReverseModule.ReverseHelper.narrow(ref);
System.out.println("Step2");

          org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");

System.out.println("Step3");
          NamingContextExt ncRef =
          NamingContextExtHelper.narrow(objRef);
System.out.println("Step4");

String name = "Reverse";
NameComponent path[] = ncRef.to_name(name);
ncRef.rebind(path,h_ref);

          System.out.println("Reverse Server reading and
          waiting...");
orb.run();
}
catch(Exception e)
{
e.printStackTrace();
}
}
}
```

ReverseClient.java

```
import ReverseModule.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.io.*;

class ReverseClient
{

public static void main(String args[])
{
Reverse ReverseImpl=null;

try
{
// initialize the ORB
          org.omg.CORBA.ORB orb =
          org.omg.CORBA.ORB.init(args,null);

          org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
```

```
            NamingContextExt ncRef =
            NamingContextExtHelper.narrow(objRef);

            String name = "Reverse";
            ReverseImpl =
            ReverseHelper.narrow(ncRef.resolve_str(name));

System.out.println("Enter String=");
            BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
String str= br.readLine();

String tempStr= ReverseImpl.reverse_string(str);

System.out.println(tempStr);
}
catch(Exception e)
{
e.printStackTrace();
}
}
}
```

ReverseImpl.java

```
import ReverseModule.ReversePOA;
import java.lang.String;
class ReverseImpl extends ReversePOA
{
      ReverseImpl()
      {
      super();
      System.out.println("Reverse Object Created");
      }

      public String reverse_string(String name)
      {
            StringBuffer str=new StringBuffer(name);
            str.reverse();
      return (("Server Send "+str));
      }
}
```

ReverseModule.idl

```
module ReverseModule
{
interface Reverse
{
string reverse_string(in string str);
};
};
```

Compiling and Executing:

1. Create the all **ReverseServer.java** , **ReverseClient.java** , **ReverseImpl.java&ReverseModule.idl** files.

2. Run the IDL-to-Java compiler idlj, on the IDL file to create stubs and skeletons. Thisstep assumes that you have included the path to the java/bin directory in your path.

idlj -fall ReverseModule.idl

The idlj compiler generates a number of files.

3. Compile the **.java files**, including the stubs and skeletons (which are in the directory newly created directory). This step assumes the java/bin directory is included in your path.

javac *.java ReverseModule/*.java

4. Start orbd. To start orbd from a UNIX command shell, enter :

orbd -ORBInitialPort 1050&

5. Start the server. To start the server from a UNIX command shell, enter :

java ReverseServer -ORBInitialPort 1050& -ORBInitialHost localhost&

6. Run the client application :

java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost

Output:

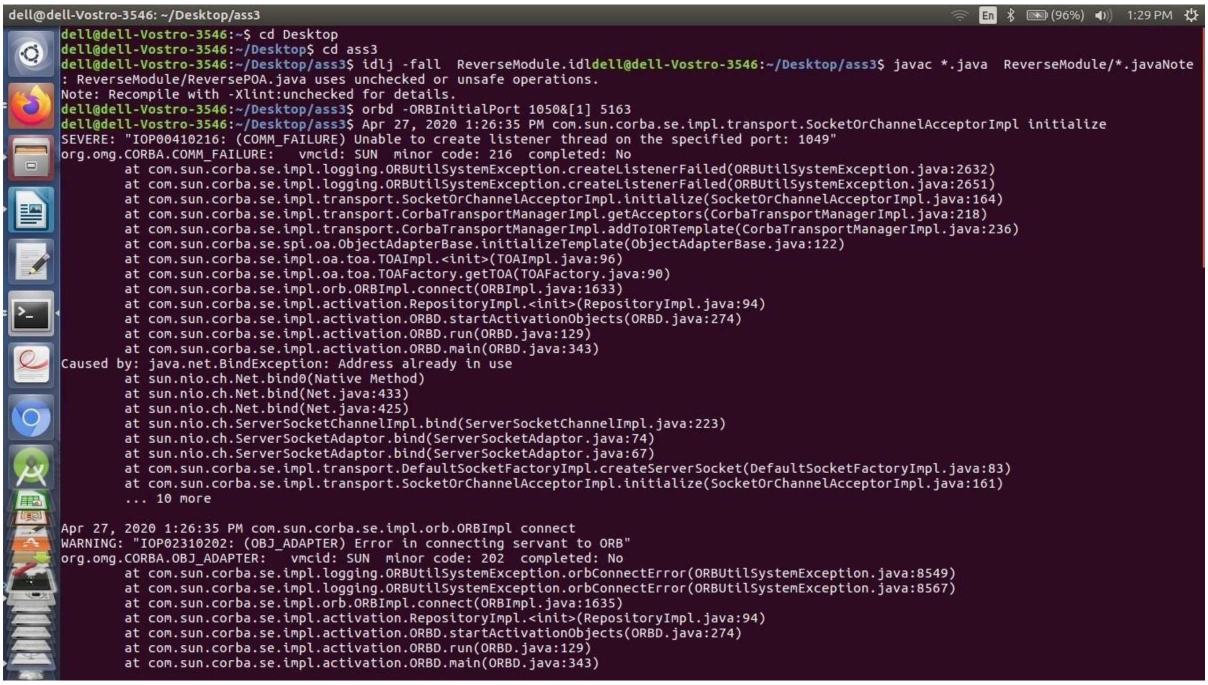
*****SERVER SIDE:**

```
dell@dell-Vostro-3546:~$ cd Desktop
dell@dell-Vostro-3546:~/Desktop$ cd ass3
dell@dell-Vostro-3546:~/Desktop/ass3$        idlj        -fall
ReverseModule.idldell@dell-Vostro-3546:~/Desktop/ass3$    javac
*.java ReverseModule/*.javaNote: ReverseModule/ReversePOA.java
uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
dell@dell-Vostro-3546:~/Desktop/ass3$   orbd   -ORBInitialPort
1050&[1] 5163
dell@dell-Vostro-3546:~/Desktop/ass3$  java  ReverseServer  -
ORBInitialPort 1050& -ORBInitialHost localhost&
[1] 4933
[2] 4934
dell@dell-Vostro-3546:~/Desktop/ass3$ -ORBInitialHost: command
not found
Reverse Object Created
Step1
Step2
```

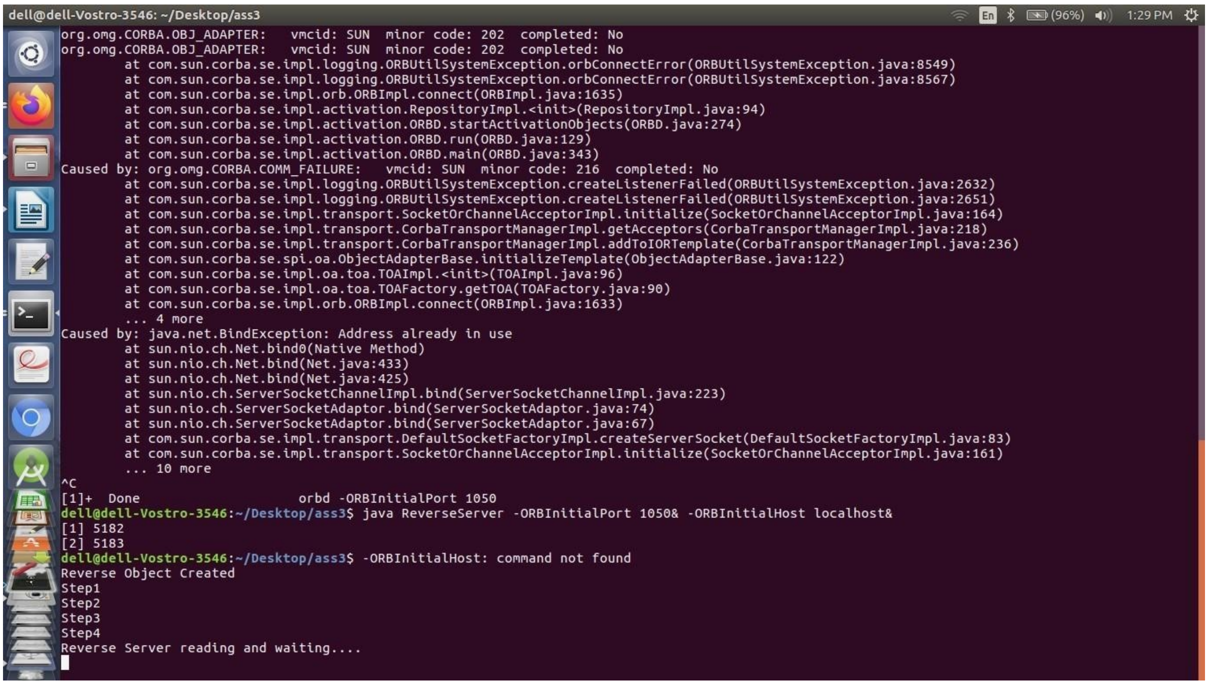
Step3

Step4

Reverse Server reading and waiting....



```
dell@dell-Vostro-3546: ~/Desktop/ass3
dell@dell-Vostro-3546:~$ cd Desktop
dell@dell-Vostro-3546:~/Desktop$ cd ass3
dell@dell-Vostro-3546:~/Desktop/ass3$ idlj -fall  ReverseModule.idldell@dell-Vostro-3546:~/Desktop/ass3$ javac *.java  ReverseModule/*.javaNote
: ReverseModule/ReversePOA.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
dell@dell-Vostro-3546:~/Desktop/ass3$ orbd -ORBInitialPort 1050&[1] 5163
dell@dell-Vostro-3546:~/Desktop/ass3$ Apr 27, 2020 1:26:35 PM com.sun.corba.se.impl.transport.SocketOrChannelAcceptorImpl initialize
SEVERE: "IOP00410216: (COMM_FAILURE) Unable to create listener thread on the specified port: 1049"
org.omg.CORBA.COMM_FAILURE:   vmcid: SUN  minor code: 216  completed: No
        at com.sun.corba.se.impl.logging.ORBUtilSystemException.createListenerFailed(ORBUtilSystemException.java:2632)
        at com.sun.corba.se.impl.logging.ORBUtilSystemException.createListenerFailed(ORBUtilSystemException.java:2651)
        at com.sun.corba.se.impl.transport.SocketOrChannelAcceptorImpl.initialize(SocketOrChannelAcceptorImpl.java:164)
        at com.sun.corba.se.impl.transport.CorbaTransportManagerImpl.getAcceptors(CorbaTransportManagerImpl.java:218)
        at com.sun.corba.se.impl.transport.CorbaTransportManagerImpl.addToIORTemplate(CorbaTransportManagerImpl.java:236)
        at com.sun.corba.se.spi.oa.ObjectAdapterBase.initializeTemplate(ObjectAdapterBase.java:122)
        at com.sun.corba.se.impl.oa.toa.TOAImpl.<init>(TOAImpl.java:96)
        at com.sun.corba.se.impl.oa.toa.TOAFactory.getTOA(TOAFactory.java:90)
        at com.sun.corba.se.impl.orb.ORBImpl.connect(ORBImpl.java:1633)
        at com.sun.corba.se.impl.activation.RepositoryImpl.<init>(RepositoryImpl.java:94)
        at com.sun.corba.se.impl.activation.ORBD.startActivationObjects(ORBD.java:274)
        at com.sun.corba.se.impl.activation.ORBD.run(ORBD.java:129)
        at com.sun.corba.se.impl.activation.ORBD.main(ORBD.java:343)
Caused by: java.net.BindException: Address already in use
        at sun.nio.ch.Net.bind0(Native Method)
        at sun.nio.ch.Net.bind(Net.java:433)
        at sun.nio.ch.Net.bind(Net.java:425)
        at sun.nio.ch.ServerSocketChannelImpl.bind(ServerSocketChannelImpl.java:223)
        at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:74)
        at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:67)
        at com.sun.corba.se.impl.transport.DefaultSocketFactoryImpl.createServerSocket(DefaultSocketFactoryImpl.java:83)
        at com.sun.corba.se.impl.transport.SocketOrChannelAcceptorImpl.initialize(SocketOrChannelAcceptorImpl.java:161)
        ... 10 more

Apr 27, 2020 1:26:35 PM com.sun.corba.se.impl.orb.ORBImpl connect
WARNING: "IOP02310202: (OBJ_ADAPTER) Error in connecting servant to ORB"
org.omg.CORBA.OBJ_ADAPTER:   vmcid: SUN  minor code: 202  completed: No
        at com.sun.corba.se.impl.logging.ORBUtilSystemException.orbConnectError(ORBUtilSystemException.java:8549)
        at com.sun.corba.se.impl.logging.ORBUtilSystemException.orbConnectError(ORBUtilSystemException.java:8567)
        at com.sun.corba.se.impl.orb.ORBImpl.connect(ORBImpl.java:1635)
        at com.sun.corba.se.impl.activation.RepositoryImpl.<init>(RepositoryImpl.java:94)
        at com.sun.corba.se.impl.activation.ORBD.startActivationObjects(ORBD.java:274)
        at com.sun.corba.se.impl.activation.ORBD.run(ORBD.java:129)
        at com.sun.corba.se.impl.activation.ORBD.main(ORBD.java:343)
```



```
dell@dell-Vostro-3546: ~/Desktop/ass3
org.omg.CORBA.OBJ_ADAPTER:   vmcid: SUN  minor code: 202  completed: No
org.omg.CORBA.OBJ_ADAPTER:   vmcid: SUN  minor code: 202  completed: No
        at com.sun.corba.se.impl.logging.ORBUtilSystemException.orbConnectError(ORBUtilSystemException.java:8549)
        at com.sun.corba.se.impl.logging.ORBUtilSystemException.orbConnectError(ORBUtilSystemException.java:8567)
        at com.sun.corba.se.impl.orb.ORBImpl.connect(ORBImpl.java:1635)
        at com.sun.corba.se.impl.activation.RepositoryImpl.<init>(RepositoryImpl.java:94)
        at com.sun.corba.se.impl.activation.ORBD.startActivationObjects(ORBD.java:274)
        at com.sun.corba.se.impl.activation.ORBD.run(ORBD.java:129)
        at com.sun.corba.se.impl.activation.ORBD.main(ORBD.java:343)
Caused by: org.omg.CORBA.COMM_FAILURE:   vmcid: SUN  minor code: 216  completed: No
        at com.sun.corba.se.impl.logging.ORBUtilSystemException.createListenerFailed(ORBUtilSystemException.java:2632)
        at com.sun.corba.se.impl.logging.ORBUtilSystemException.createListenerFailed(ORBUtilSystemException.java:2651)
        at com.sun.corba.se.impl.transport.SocketOrChannelAcceptorImpl.initialize(SocketOrChannelAcceptorImpl.java:164)
        at com.sun.corba.se.impl.transport.CorbaTransportManagerImpl.getAcceptors(CorbaTransportManagerImpl.java:218)
        at com.sun.corba.se.impl.transport.CorbaTransportManagerImpl.addToIORTemplate(CorbaTransportManagerImpl.java:236)
        at com.sun.corba.se.spi.oa.ObjectAdapterBase.initializeTemplate(ObjectAdapterBase.java:122)
        at com.sun.corba.se.impl.oa.toa.TOAImpl.<init>(TOAImpl.java:96)
        at com.sun.corba.se.impl.oa.toa.TOAFactory.getTOA(TOAFactory.java:90)
        at com.sun.corba.se.impl.orb.ORBImpl.connect(ORBImpl.java:1633)
        ... 4 more
Caused by: java.net.BindException: Address already in use
        at sun.nio.ch.Net.bind0(Native Method)
        at sun.nio.ch.Net.bind(Net.java:433)
        at sun.nio.ch.Net.bind(Net.java:425)
        at sun.nio.ch.ServerSocketChannelImpl.bind(ServerSocketChannelImpl.java:223)
        at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:74)
        at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:67)
        at com.sun.corba.se.impl.transport.DefaultSocketFactoryImpl.createServerSocket(DefaultSocketFactoryImpl.java:83)
        at com.sun.corba.se.impl.transport.SocketOrChannelAcceptorImpl.initialize(SocketOrChannelAcceptorImpl.java:161)
        ... 10 more
^C
[1]+  Done                    orbd -ORBInitialPort 1050
dell@dell-Vostro-3546:~/Desktop/ass3$ java ReverseServer -ORBInitialPort 1050& -ORBInitialHost localhost&
[1] 5182
[2] 5183
dell@dell-Vostro-3546:~/Desktop/ass3$ -ORBInitialHost: command not found
Reverse Object Created
Step1
Step2
Step3
Step4
Reverse Server reading and waiting....
```

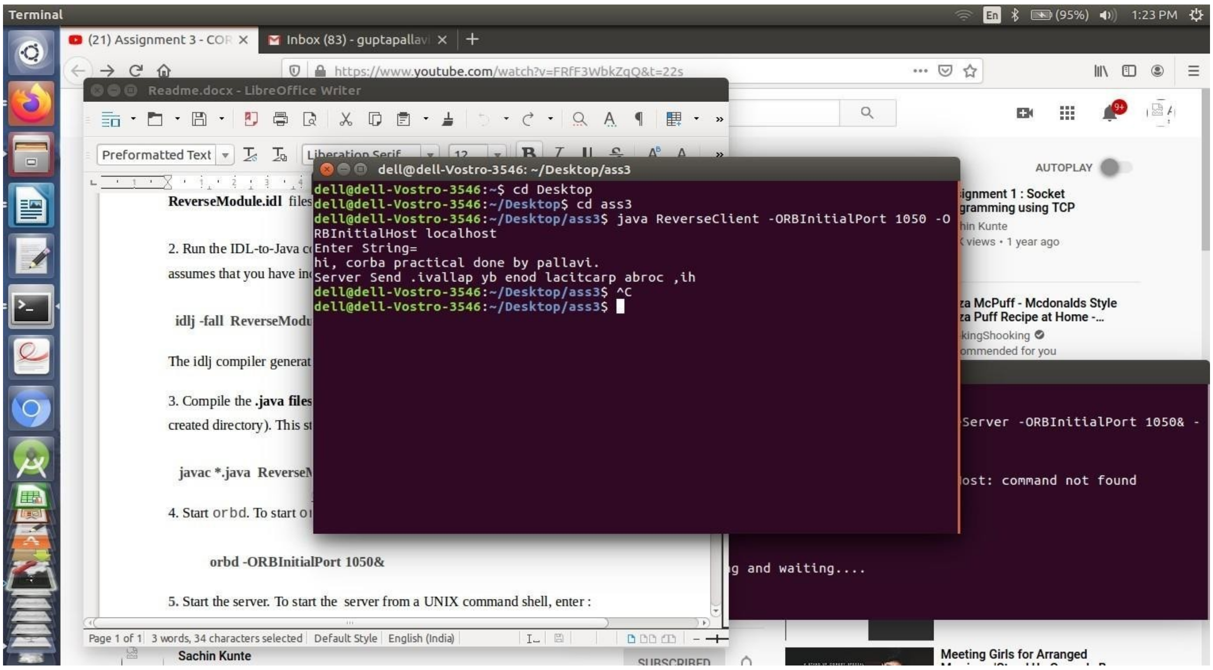
*****CLIENT SIDE:**

dell@dell-Vostro-3546:~$ cd Desktop

dell@dell-Vostro-3546:~/Desktop$ cd ass3

dell@dell-Vostro-3546:~/Desktop/ass3$ java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost

```
Enter String=

hi, corba practical done by pallavi.

Server Send .ivallap yb enod lacitcarp abroc ,ih

dell@dell-Vostro-3546:~/Desktop/ass3$
```



Conclusion:

CORBA provides the network transparency, Java provides the implementation transparency. CORBA complements the Java™ platform by providing a distributed object framework, services to support that framework, and interoperability with other languages. The Java platform complements CORBA by providing a portable, highly productive implementation environment. The combination of Java and CORBA allows you to build more scalable and more capable applications than can be built using the JDK alone.

Dr.R.D.Chintamani | **Dr.M.A.Jawale**

Practical Lab I/C | **Head,Department of IT**

ASSIGNMENT NO.4

Problem Statement:

To develop any distributed algorithm for leader election.

Tools / Environment:

Java Programming Environment, JDK 1.8, Eclipse Neon(EE).

Related Theory:

Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role. In general, it does not matter which process takes on this special responsibility, but one of them has to do it. If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process P has a unique identifier id(P). In general, election algorithms attempt to locate the process with the highest identifier and designate it as coordinator.

We also assume that every process knows the identifier of every other process. In other words, each process has complete knowledge of the process group in which a coordinator must be elected. What the processes do not know is which ones are currently up and which ones are currently down. The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

There are two types of Distributed Algorithms:
1. Bully Algorithm
2. Ring Algorithm

Bully Algorithm:

A. When a process, P, notices that the coordinator is no longer responding to requests, it initiates an election.
1. P sends an ELECTION message to all processes with higher numbers.
2. If no one responds, P wins the election and becomes a coordinator.
3. If one of the higher-ups answers, it takes over. P's job is done.

B. When a process gets an ELECTION message from one of its lower-numbered colleagues:
1. Receiver sends an OK message back to the sender to indicate that he is alive and will take over.
2. Eventually, all processes give up a part of one, and that one is the new coordinator.
3. The new coordinator announces *its* victory by sending all processes a **CO-ORDINATOR** Message telling them that it is the new coordinator.

C. If a process that *was* previously down comes back:
1. It holds an election.
2. If it happens to be the highest process currently running, it will win the election and takeoverthe coordinators job.

“Biggest guy" always wins and hence the name bully algorithm.

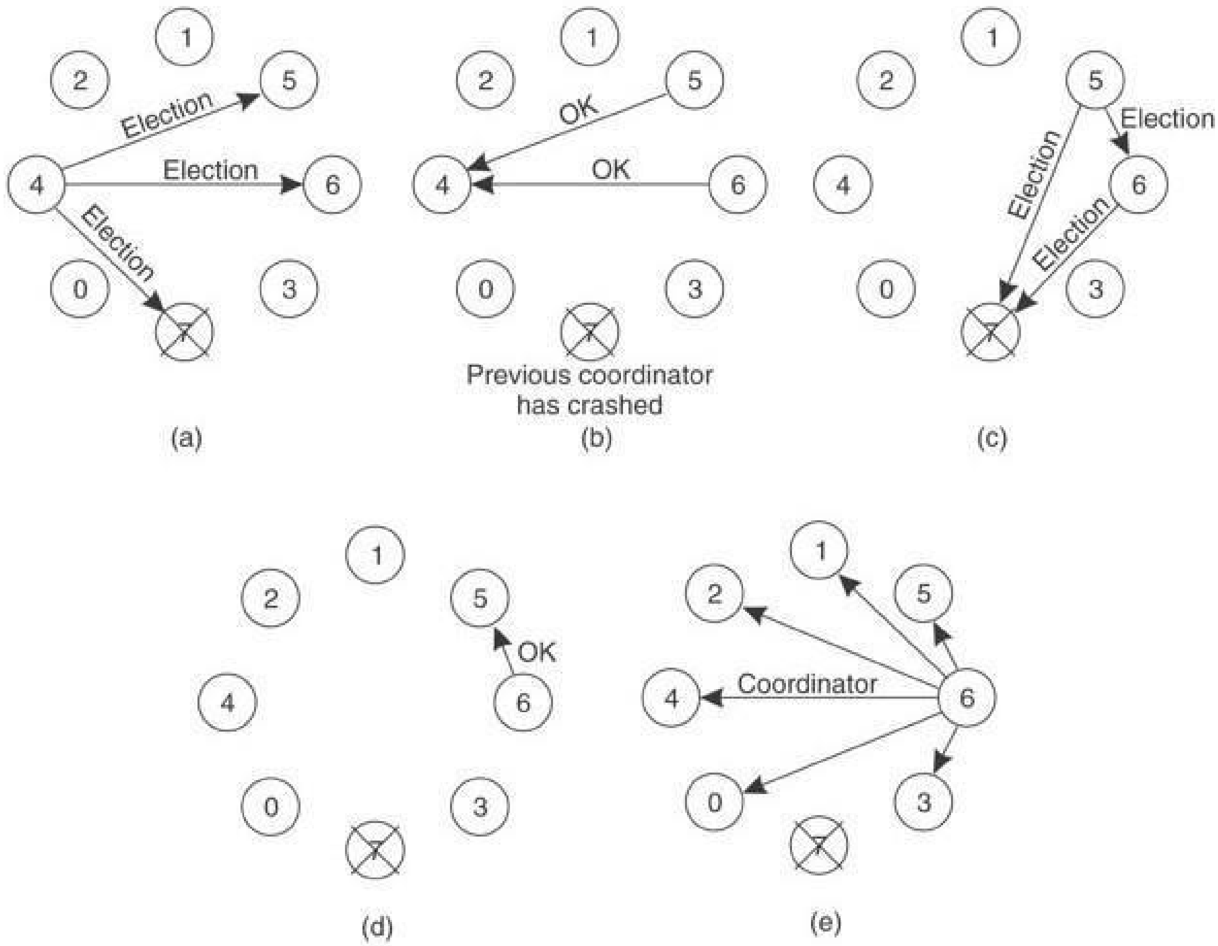


Figure 1: Bully Algorithm

Ring Algorithm:

Initiation:

1. A process notices that coordinator is not functioning:
2. Another process (initiator) initiates the election by sending "ELECTION" message(containing its own process number)

Leader Election:

3. Initiator sends the message to its successor (if successor is down, sender skips over it and goesto the next member along the ring, or the one after that, until a running process is located).
4. At each step, sender adds its own process number to the list in the message.
5. When the message gets back to the process that started it all i.e Message comes back to initiator, the **process with maximum ID Number** in the queue **wins the Election**.
6. Initiator announces the winner by sending another message (Coordinator message) around the ring.

Implementing the solution:

For Ring Algorithm:

1. Creating Class for Process which includes
    i) State: Active / Inactive

ii) Index: Stores index of process.

iii) ID: Process ID

2. Import Scanner Class for getting input from Console
3. Getting input from User for number of Processes and store them into object of classes.
4. Sort these objects on the basis of process id.
5. Make the last process id as "inactive".
6. Ask for menu

    1.Election

    2.Quit

7. Ask for initializing election process.
8. These inputs will be used by Ring Algorithm.

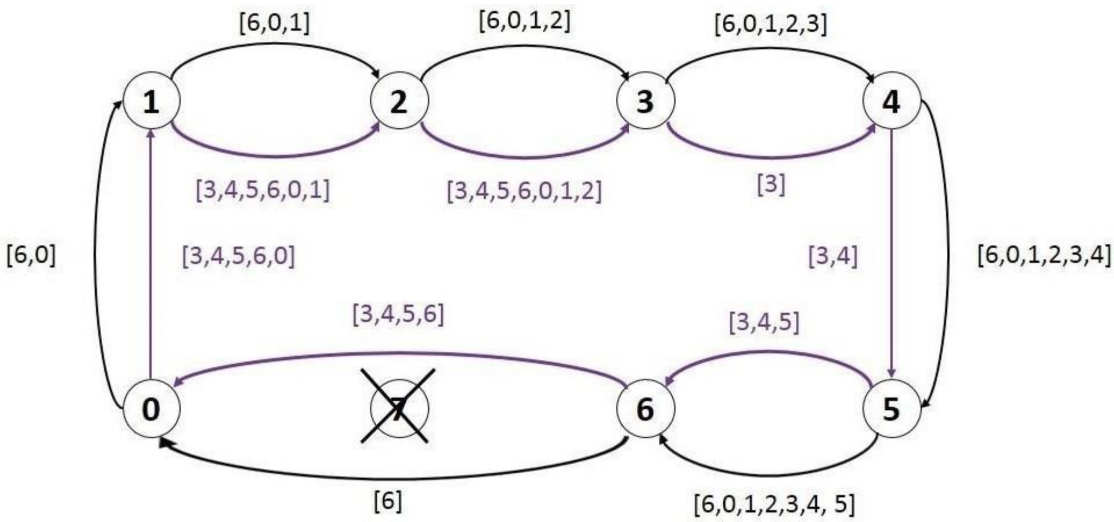


Figure 2: Ring Algorithm

Source code:

Ring.java

```
import java.util.Scanner;

public class Ring {

      public static void main(String[] args) {

            // TODO Auto-generated method stub

            int temp, i, j;
            char str[] = new char[10];
            Rr proc[] = new Rr[10];

// object initialisation
            for (i = 0; i < proc.length; i++)
                  proc[i] = new Rr();

// scanner used for getting input from console
            Scanner in = new Scanner(System.in);
            System.out.println("Enter the number of process : ");
            int num = in.nextInt();

// getting input from users
            for (i = 0; i < num; i++) {
                  proc[i].index = i;
                  System.out.println("Enter the id of process : ");
                  proc[i].id = in.nextInt();
                  proc[i].state = "active";
                  proc[i].f = 0;
            }


// sorting the processes from on the basis of id
            for (i = 0; i < num - 1; i++) {
                  for (j = 0; j < num - 1; j++) {
                        if (proc[j].id > proc[j + 1].id) {
                              temp =  proc[j].id;
                              proc[j].id = proc[j + 1].id;
                              proc[j + 1].id = temp;
                        }
                  }
            }


            for (i = 0; i < num; i++) {
                  System.out.print("  [" + i + "]" + " " + proc[i].id);
            }

            int init;
            int ch;
            int
            temp1;
            int
            temp2;
            int ch1;
            int arr[] = new int[10];

            proc[num - 1].state = "inactive";
```

```
            System.out.println("\n process " + proc[num - 1].id + "select
as co-ordinator");

            while (true) {
                  System.out.println("\n 1.election 2.quit ");
                  ch = in.nextInt();

                  for (i = 0; i < num; i++) {
                        proc[i].f = 0;
                  }

                  switch (ch) {
                  case 1:
                        System.out.println("\n Enter the Process number who
initialsied election : ");
                        init = in.nextInt();
                        init--;
                        temp2 = init;
                        temp1 = init + 1;

                        i = 0;

                        while (temp2 != temp1) {
                              if ("active".equals(proc[temp1].state) &&
proc[temp1].f == 0) {

                                    System.out.println("\nProcess " +
proc[init].id + " send message to " + proc[temp1].id);
                                    proc[temp1].f = 1;
                                    init = temp1;
                                    arr[i] = proc[temp1].id;
                                    i++;
                              }
                              if (temp1 == num) {
                                    temp1 = 0;
                              } else {
                                    temp1++;
                              }
                        }

                        System.out.println("\nProcess " + proc[init].id + "
send message to " + proc[temp1].id);
                        arr[i] = proc[temp1].id;
                        i++;
                        int max = -1;


// finding maximum for co-ordinator selection
                        for (j = 0; j < i; j++) {
                              if (max < arr[j]) {
                                    max = arr[j];
                              }
                        }

// co-ordinator is found then printing on console
                        System.out.println("\n process " + max + "select as
co-ordinator");


                        for (i = 0; i < num; i++) {
```

```
                                        if (proc[i].id == max) {
                                                proc[i].state = "inactive";
                                        }
                                }
                                break;
                        case 2:
                System.out.println("Program terminated ...");
                return ;
                        default:
                                System.out.println("\n invalid response \n");
                                break;
                        }


                }
        }

}

class Rr {

        public int index;    // to store the index of process
        public int id;       // to store id/name of process
        public int f;
        String state;        // indiactes whether active or inactive state of
node

}
```

Compiling and Executing the solution:

1. Create Java Project in Eclipse
2. Create Package
3. Add class in package Ring.java.
4. Compile and Execute in Eclipse.

Output:

```
Enter the number of process :
4
Enter the id of process :
1
Enter the id of process :
2
Enter the id of process :
3
Enter the id of process :
4
   [0] 1   [1] 2   [2] 3   [3] 4
 process 4select as co-ordinator

 1.election 2.quit
1

 Enter the Process number who initialsied election :
2

Process 2 send message to 3

Process 3 send message to 1

Process 1 send message to 2

 process 3select as co-ordinator
```

```
 1.election 2.quit
2
Program terminated ...
```

Bully.java

```
import java.io.InputStream;
import java.io.PrintStream;
import java.util.Scanner;

public class Bully {
    static boolean[] state = new boolean[5];
    int coordinator;

    public static void up(int up) {
        if (state[up - 1]) {
            System.out.println("process" + up + "is already up");
        } else {
            int i;
            Bully.state[up - 1] = true;
            System.out.println("process " + up + "held election");
            for (i = up; i < 5; ++i) {
                System.out.println("election message sent from process" +
up + "to process" + (i + 1));
            }
            for (i = up + 1; i <= 5; ++i) {
                if (!state[i - 1]) continue;
                System.out.println("alive message send from process" + i
+ "to process" + up);
                break;
            }
        }
    }

    public static void down(int down) {
        if (!state[down - 1]) {
            System.out.println("process " + down + "is already dowm.");
        } else {
            Bully.state[down - 1] = false;
        }
    }

    public static void mess(int mess) {
        if (state[mess - 1]) {
            if (state[4]) {
                System.out.println("0K");
            } else if (!state[4]) {
                int i;
                System.out.println("process" + mess + "election");
                for (i = mess; i < 5; ++i) {
                    System.out.println("election send from process" +
mess + "to process " + (i + 1));
```

```
                }
                for (i = 5; i >= mess; --i) {
                    if (!state[i - 1])
                    continue;
                    System.out.println("Coordinator message send from
process" + i + "to all");
                    break;
                }
            }
        } else {
            System.out.println("Prccess" + mess + "is down");
        }
    }

    public static void main(String[] args) {
        int choice;
        Scanner sc = new Scanner(System.in);
        for (int i = 0; i < 5; ++i) {
            Bully.state[i] = true;
        }
        System.out.println("5 active process are:");
        System.out.println("Process up = p1 p2 p3 p4 p5");
        System.out.println("Process 5 is coordinator");
        do {
            System.out.println(".........");
            System.out.println("1 up a process.");
            System.out.println("2.down a process");
            System.out.println("3 send a message");
            System.out.println("4.Exit");
            choice = sc.nextInt();
            switch (choice) {
                case 1: {
                    System.out.println("bring proces up");
                    int up = sc.nextInt();
                    if (up == 5) {
                        System.out.println("process 5 is co-ordinator");
                        Bully.state[4] = true;
                        break;
                    }
                    Bully.up(up);
                    break;
                }
                case 2: {
                    System.out.println("bring down any process.");
                    int down = sc.nextInt();
                    Bully.down(down);
                    break;
                }
                case 3: {
                    System.out.println("which process will send
message");
                    int mess = sc.nextInt();
                    Bully.mess(mess);
                }
            }
        } while (choice != 4);
    }
}
```

Output:

```
5 active process are:
Process up  = p1 p2 p3 p4 p5
Process 5 is coordinator
```

```
.........
1 up a process.
2.down a process
3 send a message
4.Exit
2
bring down any process.
5
.........
1 up a process.
2.down a process
3 send a message
4.Exit
3
which process will send message
2
process2election
election send from process2 to process 3
election send from process2 to process 4
election send from process2 to process 5
Coordinator message send from process4to all
.........
1 up a process.
2.down a process
3 send a message
4.Exit
4
```

Conclusion:

Election algorithms **are designed to choose a coordinator.** We have two election algorithms for two different configurations of distributed system. **The Bully** algorithm applies to system where every process can send a message to every other process in the system and **The Ring** algorithm m applies to systems organized as a ring (logically or physically). In this algorithm we assume that the link between the process are unidirectional and every process can message to the process on its right only.

Dr.R.D.Chintamani **Dr.M.A.Jawale**

Practical Lab I/C **Head,Department of IT**

ASSIGNMENT NO. 5

Problem Statement:

To create a simple web service and write any distributed application to consume the web service.

Tools / Environment:

Java Programming Environment, JDK 8, Netbeans IDE with GlassFish Server

Related Theory:

Web Service:
A web service can be defined as a collection of open protocols and standards for exchanging information among systems or applications.
A service can be treated as a web service if:

- The service is discoverable through a simple lookup
- It uses a standard XML format for messaging
- It is available across internet/intranet networks.
- It is a self-describing service through a simple XML syntax
- The service is open to, and not tied to, any operating system/programming language

Types of Web Services:
There are two types of web services:

1. **SOAP**: SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it's XML based, it's platform and language independent. So, our server can be based on JAVA and client can be on .NET, PHP etc. and vice versa.

2. **REST**: REST (Representational State Transfer) is an architectural style for developing web services. It's getting popularity recently because it has small learning curve when compared to SOAP. Resources are core concepts of Restful web services and they are uniquely identified by their URIs.

Web service architectures:
As part of a web service architecture, there exist three major roles.
Service Provider is the program that implements the service agreed for the web service andexposes the service over the internet/intranet for other applications to interact with.

Service Requestor is the program that interacts with the web service exposed by the ServiceProvider. It makes an invocation to the web service over the network to the Service Provider andexchanges information.

Service Registry acts as the directory to store references to the web services.
The following are the steps involved in a basic SOAP web service operational behavior:

1. The client program that wants to interact with another application prepares its requestcontent as a SOAP message.

2. Then, the client program sends this SOAP message to the server web service as an HTTPPOST request with the content passed as the body of the request.

3. The web service plays a crucial role in this step by understanding the SOAP request andconverting it into a set of instructions that the server program can understand.

4. The server program processes the request content as programmed and prepares the outputas the response to the SOAP request.

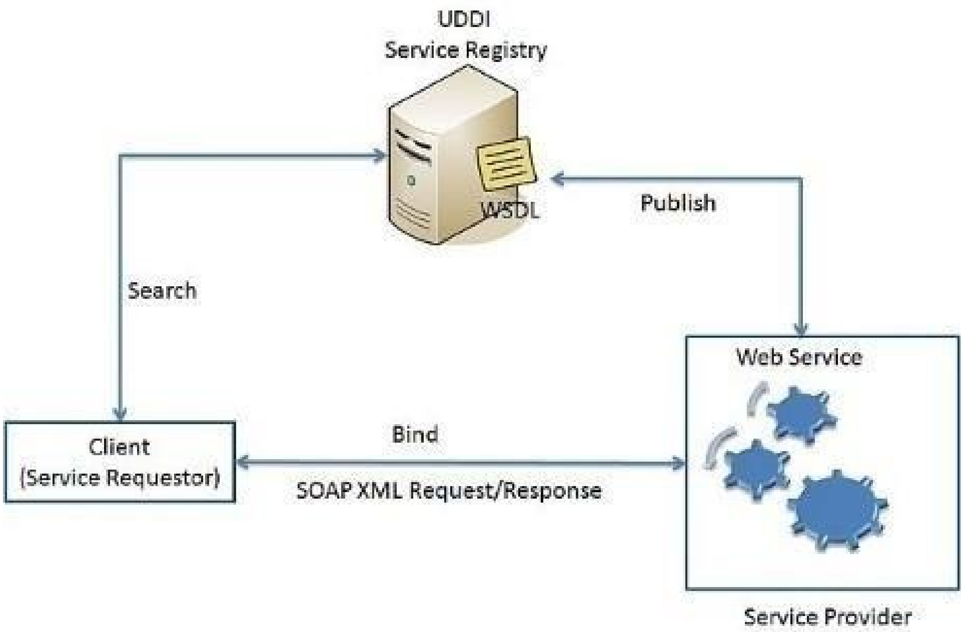
5. Then, the web service takes this response content as a SOAP message and reverts to theSOAP HTTP request invoked by the client program with this response.

6. The client program web service reads the SOAP response message to receive theoutcome of the server program for the request content it sent as a request.

SOAP web services:
Simple Object Access Protocol (**SOAP**) is an XML-based protocol for accessing web services.
It is a W3C recommendation for communication between two applications, and it is a platformandlanguage-independent technology in integrated distributed applications.

While XML and HTTP together make the basic platform for web services, the following are thekey components of standard SOAP web services:
Universal Description, Discovery, and Integration (**UDDI**)*:* UDDI is an XMLbasedframework for describing, discovering, and integrating web services. It acts as a directory of webservice interfaces described in the WSDL language.

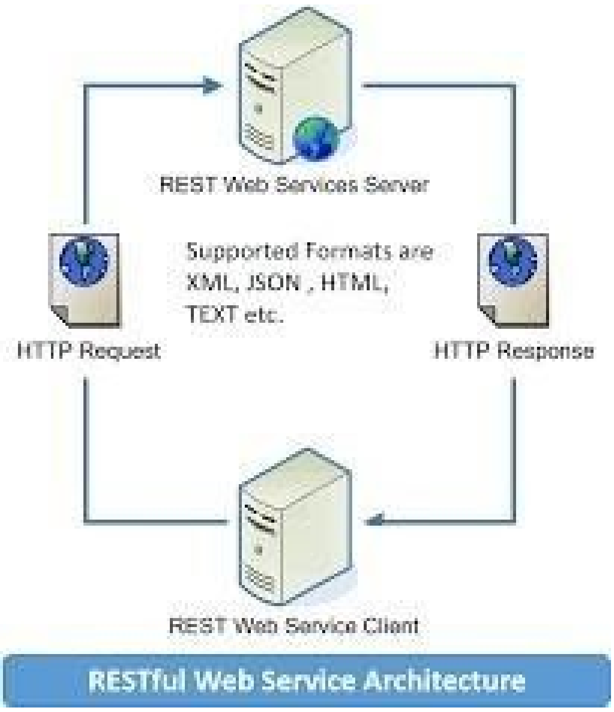
Web Services Description Language (**WSDL**)*:* WSDL is an XML document containinginformation about web services, such as the method name, method parameters, and how toinvoke the service. WSDL is part of the UDDI registry. It acts as an interface betweenapplications that want to interact based on web services. The following diagram shows theinteraction between the UDDI, Service Provider, and service consumer in SOAP web services:



SOAP Web Services Architecture

RESTful web services

REST stands for **Representational State Transfer**. RESTful web services are considered aperformance-efficient alternative to the SOAP web services. REST is an architectural style, not aprotocol. Refer to the following diagram:



While both SOAP and RESTful support efficient web service development, the differencebetween these two technologies can be checked out in the following table:

	SOAP	REST
Bandwith usage	Uses more bandwith over the internet	Uses less bandwith
Client-server coupling	Tighter client-server coupling	Looser client server coupling
Security	Built in mechanism for security	No built in security
Data formats	Supports only XML	Supports multiple formats
Exposing business logic	Service interfaces	URIs
Failure handling	Retry logic built-in	Expects clients to retry
Caching data	Cannot be cached	Can be cached
Java API	JAX-WS	JAX-RS

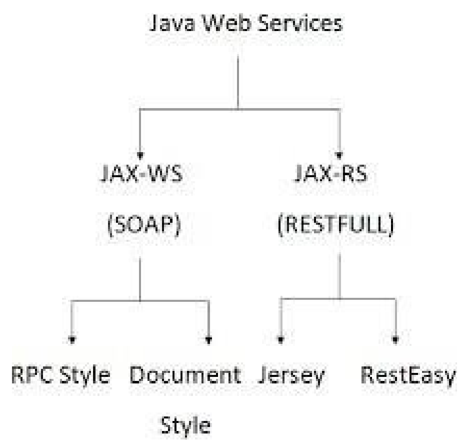
Designing the solution:

Java provides it's own API to create both SOAP as well as RESTful web services.

1. **JAX-WS**: JAX-WS stands for Java API for XML Web Services. JAX-WS is XML basedJava API to build web services server and client application.

2. **JAX-RS**: Java API for RESTful Web Services (JAX-RS) is the Java API for creatingREST web services. JAX-RS uses annotations to simplify the development anddeployment of web services.

Both of these APIs are part of standard JDK installation, sono need to add any jars towork with them.



Implementing the solution:

Step 1: Choosing a container:

You can either deploy your web service in a web container.

- Choose **File** >**New Project** (Ctrl-Shift-N on Linux and Windows).
- Select **Web Application** from the Java **Web** category.
- Name the project *CalculatorWSApplication*.
- Select a location for the project. Click **Next**.
- Select the server **[Glassfish / Tomcat]** and **Java EE** version and click **Finish.**

Step 2: Creating a Web Service from a Java Class:

- Right-click the *CalculatorWSApplication* node and **choose New** >**Web Service**.
- Name the web service *CalculatorWS* and type *org.me.calculator* in Package.
- Keep "**Create Web Service from Scratch**" check box selected.
- If you are creating a Java EE project on GlassFish, select "**Implement Web Service as a Stateless Session Bean**".
- Click **Finish**. The Projects window displays the structure of the new web service and the source code is shown in the editor area.

Step 3: Adding an Operation to the Web Service:

- Find the web service's node in the Projects window. Right-click that node. A context menu opens.
- Click **Add Operation** in either the visual designer or the context menu. The Add Operation dialog opens.
- In the upper part of the **Add Operation dialog box**, type **add** in **Name** and type **int**in the **Return Type** drop-down list.
- In the lower part of the Add Operation dialog box, click **Add** and create a parameterof type **int**named **i**. Click **Add** again and create a parameter of type **int**called **j**.
- Click **OK** at the bottom of the Add Operation dialog box. You return to the editor.
- Remove the **default hello operation**, either by deleting the hello() method in the **source code** or by selecting the hello operation in the **visual designer** and clicking **RemoveOperation**.
- Click **Source** menu and view the generated code.
- In the editor, extend the skeleton **add operation.** Add the following:

```
int k = i + j;
return k; (instead of return 0)
```

Step 4: Deploying and Testing the Web Service:
Once you deploy a webservice to a server, you can use the IDE to open the server's test client. TheGlassFish server provide test clientswhereas in Tomcat Web Server, thereis no test client.

- Right-click the **project** and choose **Deploy**. The IDE starts the applicationserver, builds the application, and deploys the application to the server.
- In the IDE's **Projects** tab, expand the **Web Services** node of the*CalculatorWSApplication* project. Right-click the *CalculatorWS*node, and choose **Test Web Service.**
- The IDE opens the tester page in the browser, if you deployed a web application to the GlassFish server.

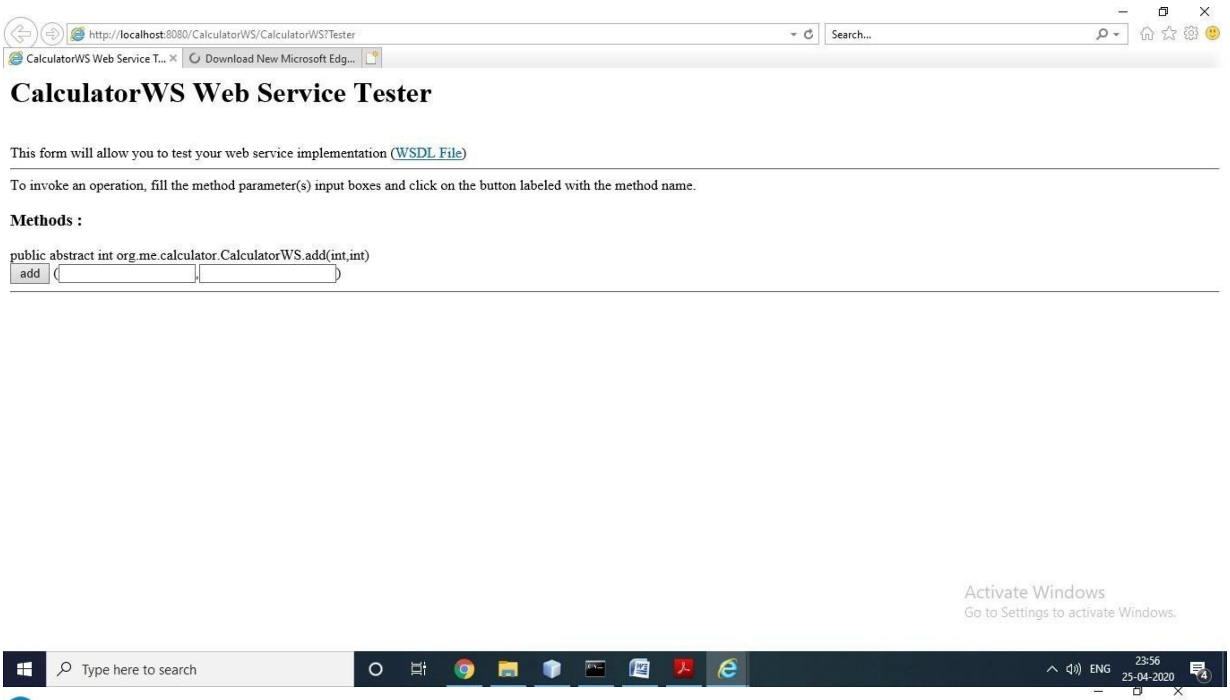
Step 5: Consuming the Web Service:
Once the web service is deployed, youneed to **create a client** to make use of the web service's **add** method. Here,you can create three types of clients: a Java class in a Java SE application, aservlet, and a JSP page in a web application.
Client 1: Java Class in Java SE Application

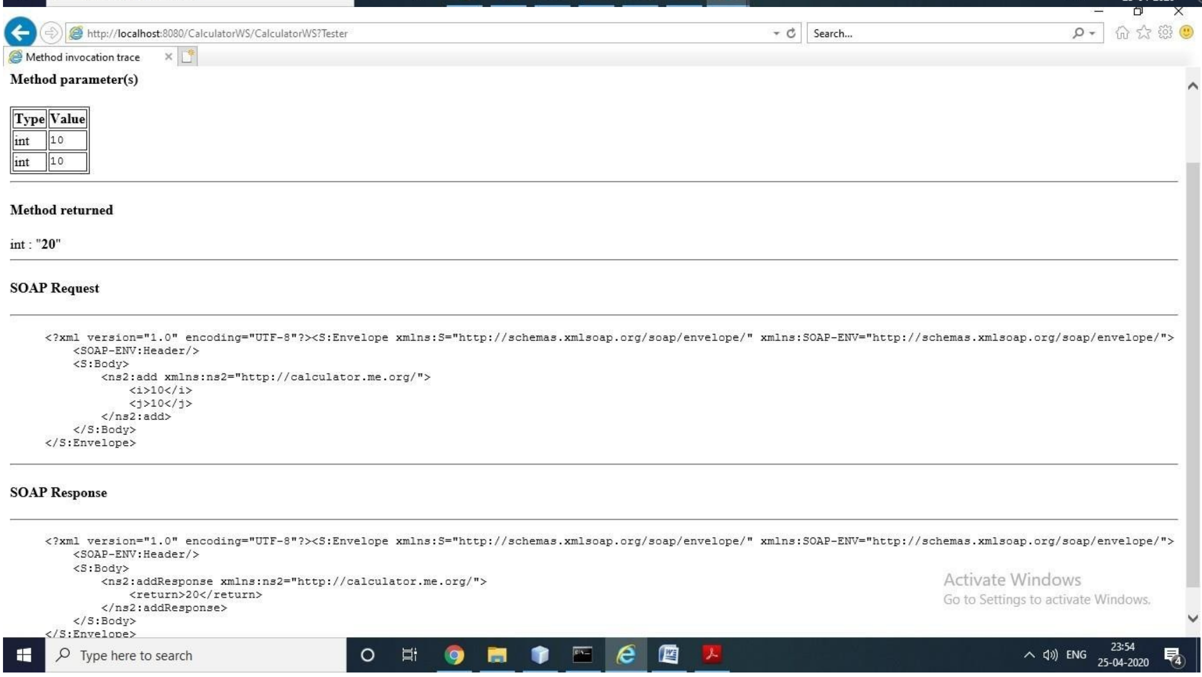
- Choose **File** >**New Project**.
- Select **Java Application** from the **Java** category.
- Name the project CalculatorWS_Client_Application.
- Keep "**Create Main Class selected**" and accept all other default settings. Click **Finish**.
- Right-click the CalculatorWS_Client_Application node and choose **New** >**Web Service Client**. The New Web Service Client wizard opens.
- Select "Project as the **WSDL…** source. Click on **Browse** button. Browse to the CalculatorWS web service in the **CalculatorWSApplication**project.
- When you have selected the web service, click **OK**.
- Do not select **a package** name. Leave this field empty. Keep the other settings at default and click **Finish**.
- The Projects window displays the new **web service client**, with a node for the **add** method that is created
- Double-click your **main class** so that it opens in the Source Editor. Drag the **add** node below the main() method.
- In the main() method body, **write the** code that initializes values for i and j, calls add(), and prints the result.

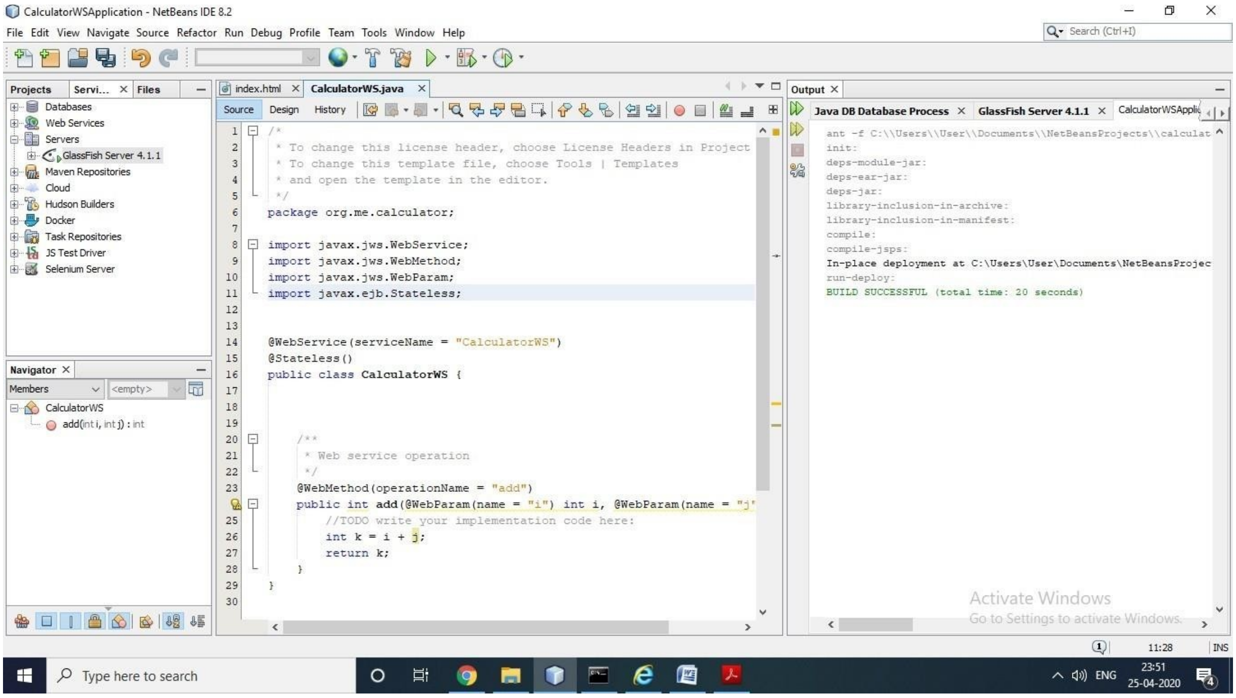
```
try
{
        inti=3;
        int j=4;
        int result = add(i,j);
        System.out.println("Result =" + result);
} catch (Exception ex) {
                System.out.println("Exception =" + ex);
```

Compiling and Executing the solution:
Right Click on the Project node and Choose Run.

Output:


http://localhost:8080/CalculatorWS/CalculatorWS?Tester
CalculatorWS Web Service T...
Download New Microsoft Edg...
CalculatorWS Web Service Tester
This form will allow you to test your web service implementation (WSDL File)
To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.
Methods :
public abstract int org.me.calculator.CalculatorWS.add(int,int)
add
Activate Windows
Go to Settings to activate Windows.
Type here to search
ENG
23:56
25-04-2020


http://localhost:8080/CalculatorWS/CalculatorWS?Tester
Method invocation trace
Method parameter(s)
Type Value
int 10
int 10
Method returned
int : "20"
SOAP Request
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<S:Body>
<ns2:add xmlns:ns2="http://calculator.me.org/">
<i>10</i>
<j>10</j>
</ns2:add>
</S:Body>
</S:Envelope>
SOAP Response
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<S:Body>
<ns2:addResponse xmlns:ns2="http://calculator.me.org/">
<return>20</return>
</ns2:addResponse>
</S:Body>
</S:Envelope>
Activate Windows
Go to Settings to activate Windows.
Type here to search
ENG
23:54
25-04-2020



Conclusion:

This assignment, described the Web services approach to the Service Oriented Architecture concept. Also, described the Java APIs for programming Web services and demonstrated examples of their use by providing detailed step-by-step examples of how to program Web services in Java.

Dr.R.D.Chintamani | **Dr.M.A.Jawale**

Practical Lab I/C | **Head,Department of IT**

ASSIGNMENT NO. 6

Problem Statement:

To develop any distributed application using Messaging System in Publisher-Subscriber paradigm.

Tools / Environment:

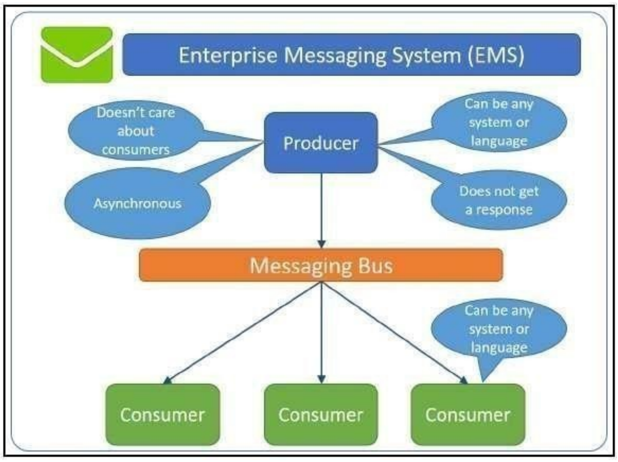
Java Programming Environment, JDK 8, Eclipse IDE, Apache ActiveMQ 4.1.1, JMS

Theory:

Large distributed systems are often overwhelmed with complications caused by heterogeneity and interoperability. Heterogeneity issues may arise due to the use of different programming languages, hardware platforms, operating systems, and data representations. Interoperability denotes the ability of heterogeneous systems to communicate meaningfully and exchange data or services. With the introduction of middleware, heterogeneity can be alleviated and interoperability can be achieved.

Middleware is a layer of software between the distributed application and the operating system and consists of a set of standard interfaces that help the application use networked resources and services.

Enterprise Messaging System:



EMS, or the messaging system, defines system standards for organizations so they can define their enterprise application messaging process with a semantically precise messaging structure. EMS encourages you to define a loosely coupled application architecture in order to define an industry-accepted message structure; this is to ensure that published messages would be persistently consumed by subscribers. Common formats, such as XML or JSON, are used to do this. EMS recommends these messaging protocols: DDS, MSMQ, AMQP, or SOAP web services. Systems designed with EMS are termed **Message-Oriented Middleware (MOM)**. An asynchronous communication is used while messaging in EMS.

Java Messaging Service:

Java's implementation of an EMS in the Application Programming Interface (API) format is known as JMS.

JMS allows distributed Java applications to communicate with applications developed in any other technology that understands messaging through asynchronous messages. JMS applications contain a provider, clients, messages, and administrated objects.

JMS providing a standard, portable way for Java programs to send/receive messages through a MOM product. Any application written in JMS can be executed on any MOM that implements the JMS API standards. The JMS API is specified as a set of interfaces as part of the Java API. Hence, all the products that intend to provide JMS behavior will have to deliver the provider to implement JMS-defined interfaces. With programming patterns that allow a program to interface, you should be able to construct a Java application in line with the JMS standards by defining the messaging programs with client applications to exchange information through JMS messaging.

The publish/subscribe messaging paradigm:

The publish/subscribe messaging paradigm is built with the concept of a topic, which behaves like an announcement board. Consumers subscribe to receiving messages that belong to a topic, and publishers report messages to a topic. The JMS provider retains the responsibility for distributing the messages that it receives from multiple publishers to many other subscribers based on the topic they subscribe to. A subscriber receives messages that it subscribes to based on the rules it defines and the messages that are published after the subscription is registered; they do not receive any messages that are already published, as shown in the following diagram:

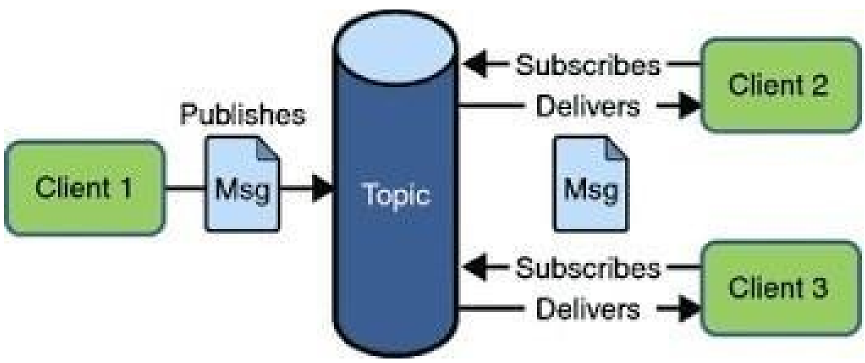


Fig: Publish / Subscribe Messaging Paradigm

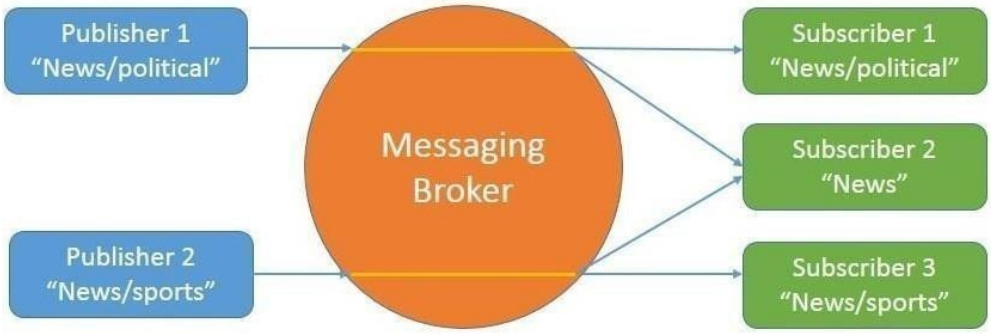


Fig: Publish / Subscribe Routing Model

JMS interfaces

JMS defines a set of high-level interfaces that encapsulate several messaging concepts. These high- level interfaces are further extended for the Point-To-Point and publish / subscribe messaging domains:

`ConnectionFactory:` This is an administered object with the ability to create a connection.

`Connection:` This is an active connection handle to the provider.

`Destination:` This is an administered object that encapsulates the identity of a message destination where messages are sent to/received from.

`Session:` This is a single-threaded context for sending/receiving messages. To ensure a simple session-based transaction, concurrent access to a message by multiple threads is restricted. We can use multiple sessions for a multithreaded application.

`MessageProducer:` This is used to send messages.

`MessageConsumer:` This is used to receive messages.

The following table shows interfaces specific to publish/subscribe paradigms enhanced from their corresponding high-level interface:

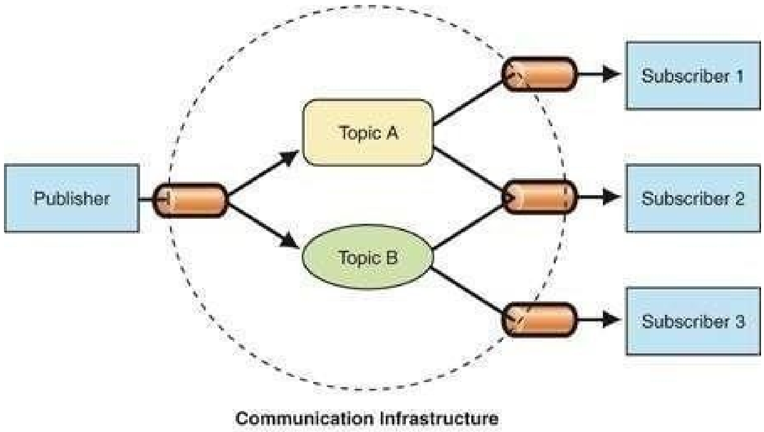
High Level Interface	Publish Subscribe Model Interface
ConnectionFactory	TopicConnectionFactory
Connection	TopicConnection
Destination	Topic
Session	TopicSession
MessageProducer	TopicPublisher
MessageConsumer	TopicSubscriber

Designing the solution:

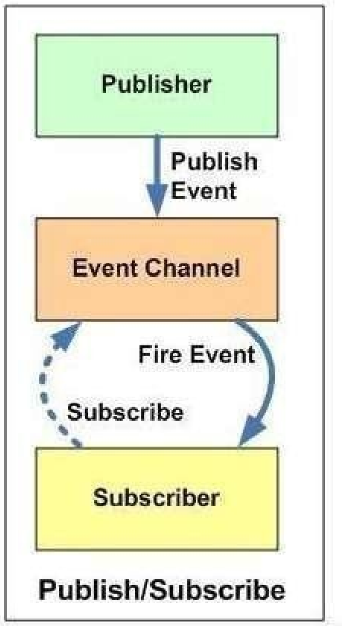
In ‘Publisher-Subscriber’ pattern, senders of messages, called **publishers**, do not program the messages to be sent directly to specific receivers, called **subscribers.**

For example, consider there is a publisher publishes news (topics) related to politics and sports; they publish to the Messaging Broker, as shown in the following diagram. While Subscriber 1 receives news related to politics and Subscriber 3 receives news related to sports, Subscriber 2 will receive both political and sports news as it subscribed to the common topics.

In designing our solution, we have created one publisher and subscriber wherein the publisher creates topic.



The **Publisher/Subscriber** pattern is mostly implemented in an ***asynchronous*** way (using message queue).



Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

JMS is a Java API that allows applications to create, send, receive, and read messages. The JMS API enables communication that is loosely coupled, asynchronous and reliable.

To use JMS, we need to have a JMS provider that can manage the sessions, queues, and topics. Some examples of known JMS providers are Apache ActiveMQ, WebSphere MQ from IBM or SonicMQ from Aurea Software. Starting from Java EE version 1.4, a JMS provider has to be contained in all Java EE application servers.

Refer to the following diagram:

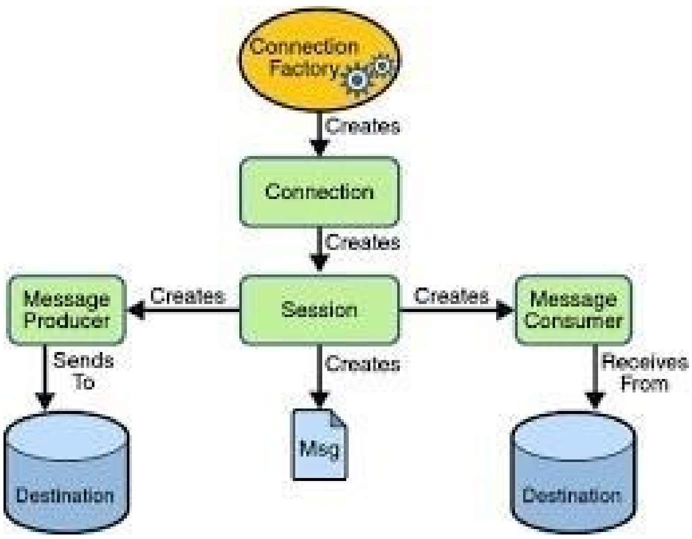


Fig: Java Messaging Service

A JMS provider is a messaging server that supports the creation of connections (multithreaded virtual links to the provider) and sessions (single-threaded contexts for producing and consuming messages). A JMS client is a Java program that either produces or consumes messages.

JMS messages are objects that communicate information between JMS clients and are composed of a header, some optional properties, and an optional body.

Administered objects are preconfigured JMS objects, such as a connection factory (the object a client uses to create a connection to a provider) and a destination (the object a client uses to specify a target for its messages).

JMS applications are usually developed in either the publish/subscribe or Point-To-Point paradigm.

The following are the objectives of JMS, as highlighted in its specification:

- Defining a common collection of messaging concepts and features
- Minimizing the number of concepts a developer should learn to develop applicationsas EMS's
- Improving the application messaging portability
- Reducing the effort involved in implementing a provider
- Providing client interfaces for both Point-To-Point and pub/sub domains

Implementing the solution:

1. To execute the pub-sub programs, you need the message queue environment.

   The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. It is a Java API that allows applications to create, send, receive, and read messages. The JMS API enables communication that is loosely coupled, asynchronous and reliable.

   To use JMS, we need to have a JMS provider that can manage the sessions, queues, and topics. Some examples of known JMS providers are Apache ActiveMQ, WebSphere MQ from IBM or SonicMQ from Aurea Software. Starting from Java EE version 1.4, a JMS provider has to be contained in all Java EE application servers.

   Here we are implementing the JMS concepts and illustrates them with a JMS Hello World example using ActiveMQ.

   Interfaces extending core JMS interfaces for Topic help build publish-subscribe components.

2. The Publisher.java program to publish messages to the Publish-Subscribe topic. The code for which is shown in the below section.

3. While the preceding program helps publish messages to the Publish-Subscribe Topic, the Subscribe.java program is used to subscribe to the Publish-Subscribe Topic, which keeps receiving messages related to the Topic until the quit command is given.

Source code:

Publisher.java

```
package pubsub;

import javax.jms.*;
import   org.apache.activemq.ActiveMQConnection;
import  org.apache.activemq.ActiveMQConnectionFactory;

public class Publisher {
      private static String url =
      ActiveMQConnection.DEFAULT_BROKER_URL;
      public static void main(String[] args) throws JMSException {
            ConnectionFactory connectionFactory = new
            ActiveMQConnectionFactory(url);
            Connection connection=connectionFactory.createConnection();
            connection.start();
// JMS messages are sent and received using a Session. We  willcreate
// here a non-transactional session object. If you want to  use
// transactions you should set the first parameter to 'true'
            Session  session  =  connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
            Topic topic =   session.createTopic("CL9");
            MessageProducer producer =  session.createProducer(topic);
// We will send a small text message saying 'Hello'
            TextMessage message =  session.createTextMessage();
            message.setText("This is a new message from publisher");
// Here we are sending the message!
            producer.send(message);
            System.out.println("Sent message '" + message.getText() +
            "'");
            connection.close();
      }
}
```

Subscriber.java

```
package pubsub;

import java.io.IOException;
import javax.jms.*;
import  org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;

public class Subscriber {
// URL of the JMS server
     private static String url=ActiveMQConnection.DEFAULT_BROKER_URL;

// Name of the topic from which we will receive messages from = " CL9"
     public static void main(String[] args) throws JMSException {
// Getting JMS connection from the server
          ConnectionFactory connectionFactory = new
          ActiveMQConnectionFactory(url);
          Connection connection=connectionFactory.createConnection();
          connection.start();
          Session session = connection.createSession(false,
          Session.AUTO_ACKNOWLEDGE);
          Topic topic =  session.createTopic("CL9");
          MessageConsumer consumer = session.createConsumer(topic);
          MessageListener listner = new MessageListener() {
               public void onMessage(Message message) {
                    try {
                         if (message instanceof TextMessage) {
                              TextMessage textMessage =
                              TextMessage) message;
                              System.out.println("Received message" +
                              textMessage.getText() + "'");
                         }
                    } catch (JMSException e) {
                         System.out.println("Caught:" + e);
                         e.printStackTrace();
                    }
               }
          };
          consumer.setMessageListener(listner);
          try {
               System.in.read();
          } catch (IOException e) {
               e.printStackTrace();
          }
          connection.close();
     }
}
```

Compilation and Executing the solution:

For Setting up an environment:

1. Download the 2 Jar files javax.jms.jar for JMS and apache-activemq-4.1.1.jar for Apache ActiveMQ.

2. Download Apache MQ and Install it using the Apache MQ InstallationLink

Steps to execute:

1. Create project in eclipse with package named **pubsub**.
2. Create class **Publisher** and **Subscriber**.
3. Add th jar files **activemq-all-5.15.8.jar**

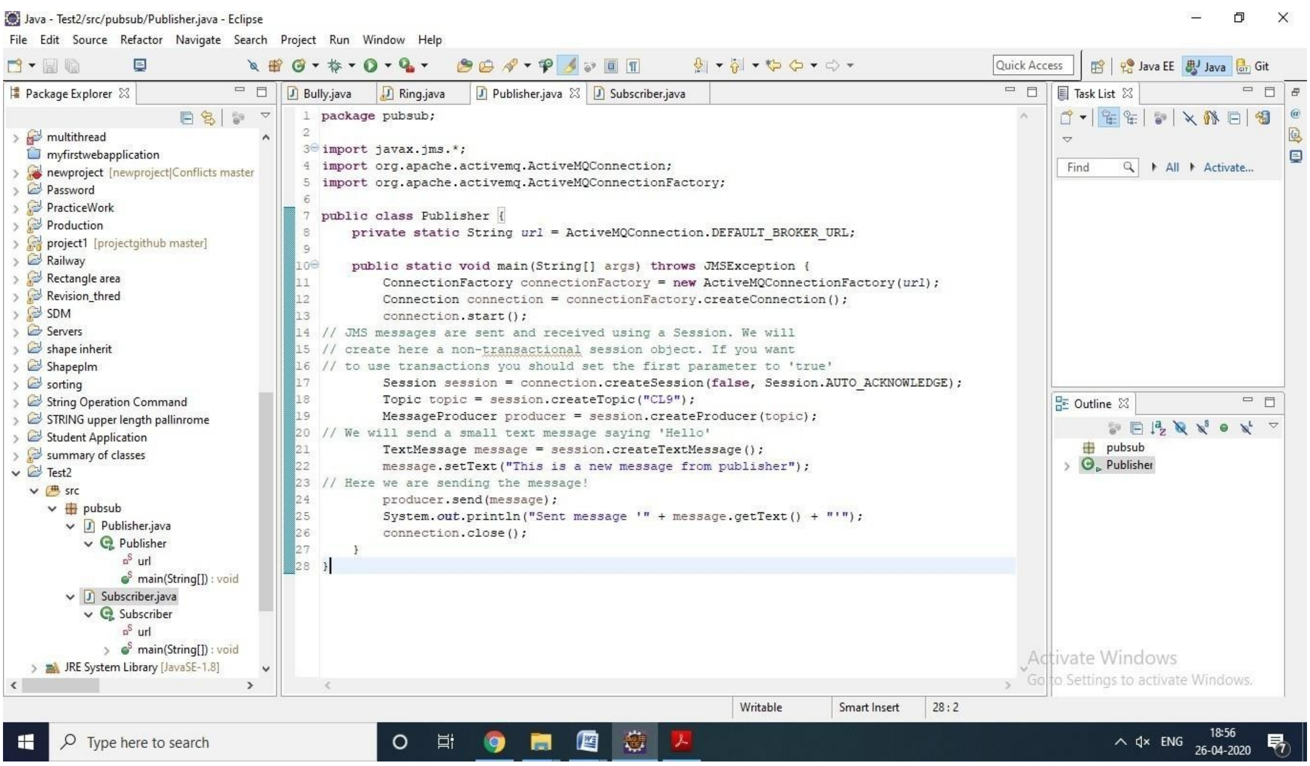
   a. Right Click on Project in eclipse package explorer
   b. Go to Build Path
   c. Select Configure Build Path
   d. Add external jars
   e. Select both the downloaded jars from the first step

4. Run activemq with the following command:

   ```
   sudo sh activemq start
   ```

5. Run Subscriber and Run the Publisher

Output:



***Subscriber**
INFO | Successfully connected to tcp://localhost:61616
Received message ‘This is a new message from publisher'

***Publisher**
INFO | Successfully connected to tcp://localhost:61616
Sent message 'This is a new message from publisher'

Conclusion:

This assignment includes study of Publish-Subscribe model of Communication which is implemented using JMS and Apache ActiveMQ. The topic based filtering requires the messages to be broadcasted into logical channels, the subscribers only receives messages from logic channels they are subscribed.

Dr.R.D.Chintamani	**Dr.M.A.Jawale**
Practical Lab I/C	**Head,Department of IT**

ASSIGNMENT NO. 5

Problem Statement:

Implement Berkeley algorithm for clock synchronization.

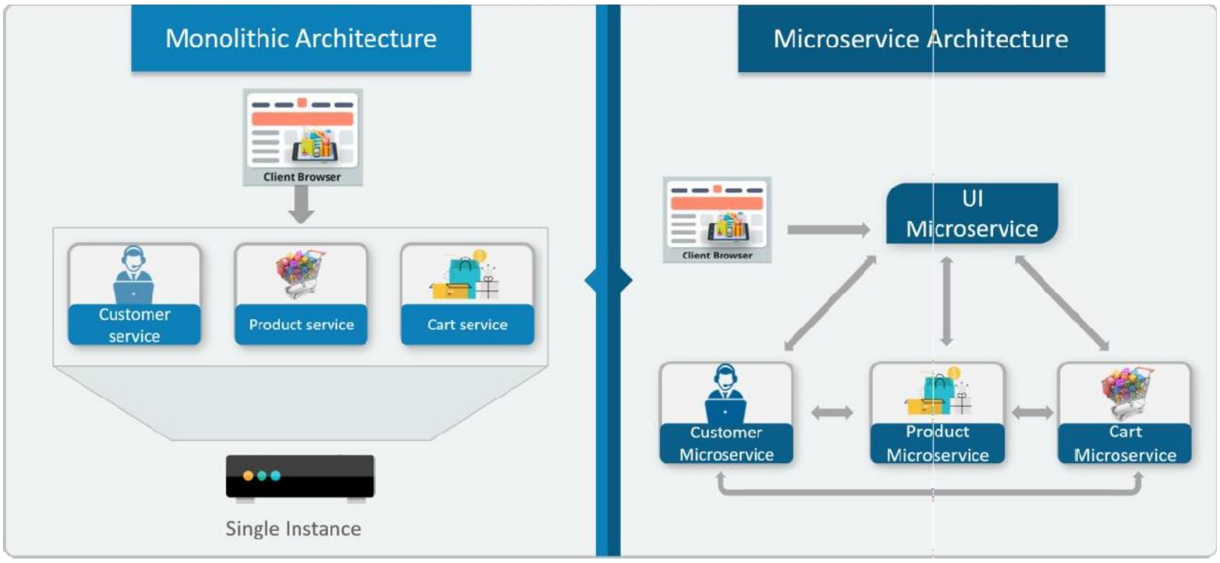
Tools / Environment:

Java Programming Environment, jdk 1.8, Eclipse IDE.

Related Theory:

1. Micro services:

Traditional application design is often called "monolithic" because the whole thing is developed in one piece. Even if the logic of the application is modular it's deployed as one group, like a Java application as a JAR file for example. This monolith eventually becomes so difficult to manage as the larger applications require longer and longer deployment time frames. In contrast with the monolith type application, here's what an app developed with a micro services focus might look like:



A team designing a micro services architecture for their application will split all of the major Functions of an application into independent services. Each independent service is usually Packaged as an API so it can interact with the rest of the application elements.

Micro services - also known as the micro service architecture - is an architectural style that structures an application as a collection of services that are:

- Highly maintainable and testable

- Loosely coupled
- Independently deployable
- Organized around business capabilities.

The micro service architecture enables the continuous delivery/deployment of large, complex applications. It also enables an organization to evolve its technology stack.

2. Web frameworks encapsulate what developers have learned over the past twenty years while programming sites and applications for the web. Frameworks make it easier to reuse code for common HTTP operations and to structure projects so other developers with knowledge of the framework can quickly build and maintain the application.

Common web framework functionality: Frameworks provide functionality in their code or through extensions to perform common operations required to run web applications. These common operations include:
1. URL routing
2. Input form handling and validation
3. HTML, XML, JSON, and other output formats with a templating engine
4. Database connection configuration and persistent data manipulation through an object relational mapper (ORM)
5. Web security against Cross-site request forgery (CSRF), SQL Injection, Cross-site Scripting(XSS) and other common malicious attacks
6. Session storage and retrieval.

3. Flask (source code) is a Python web framework built with a small core and easy-to-extend philosophy. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine.

4. WSGI: Web Server Gateway Interface (WSGI) has been adopted as a standard for Pythonweb application development. WSGI is a specification for a universal interface between the web server and the web applications.

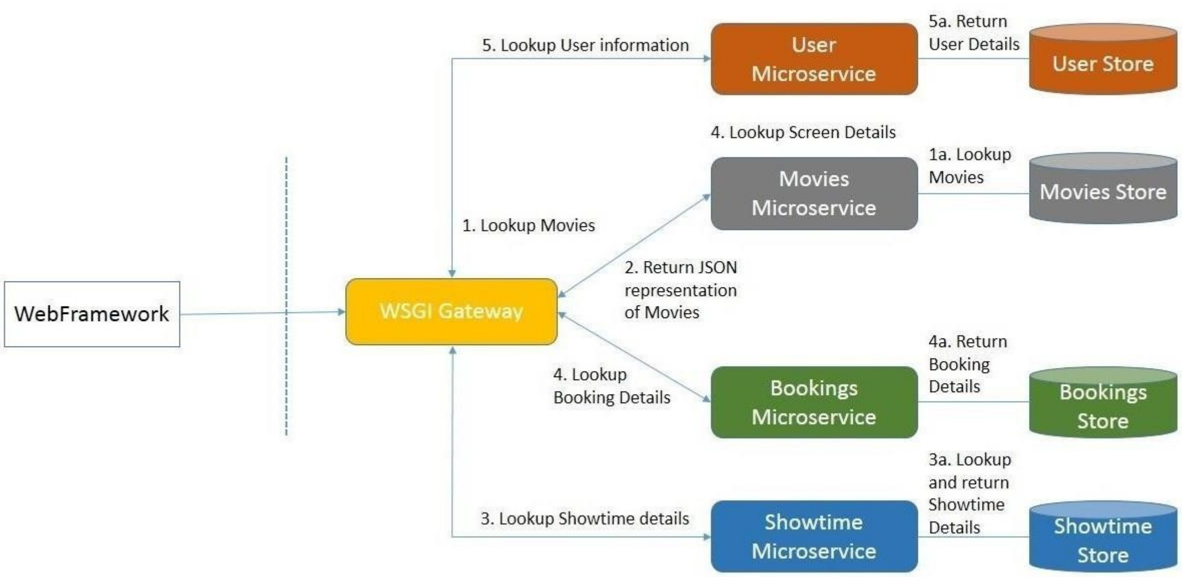
5. Werkzeug: It is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.

6. Virtual Environment:
In Python, by default, every project on the system will use the same directories to store and retrieve **site packages** (third party libraries). And **system packages** (packages that are part of the standard Python library). Consider the a scenario where there are two projects: *Project A* and *Project B*, both of which have a dependency on the same library, *Project C*. The problem becomes

apparent when we start requiring different versions of *Project C*. Maybe *Project A* needs v1.0.0, while *Project B* requires the newer v2.0.0, for example.

Since projects are stored in site-packages directory according to just their name and can't differentiate between versions, both projects, *Project A* and *Project B*, would be required to use the same version which is unacceptable in many cases and hence the virtual environment. The main purpose of Python virtual environments is to create an isolated environment for Python projects. This means that each project can have its own dependencies, regardless of what dependencies every other project has. There are no limits to the number of environments you can have since they're just directories containing a few scripts. Plus, they're easily created using the virtual env or pyenv command line tools.

Designing the solution:



Here, we are attempting to develop an micro service based architecture for Movie ticket Booking web application. The services are being implemented using python and JSON is used as for Data Store.

Implementing the solution:

1. Using Virtual Environments:
Install virtual env for development environment.virtualenv is a virtual Python environment builder. It helps a user to create multiple Python environments side-by-side. Thereby, it can avoid compatibility issues between the different versions of the libraries.

The following command installs virtualenv:

sudo apt-get install virtualenv

2. Flask Module:

Importing flask module in the project is mandatory. An object of Flask class is our WSGI application. Flask constructor takes the name of current module (nam_e) as_argument. The route() function of the Flask class is a decorator, which tells the application which URL should call the associated function.

Route decorator:

The route() decorator in Flask is used to bind URL to a function.

For example −

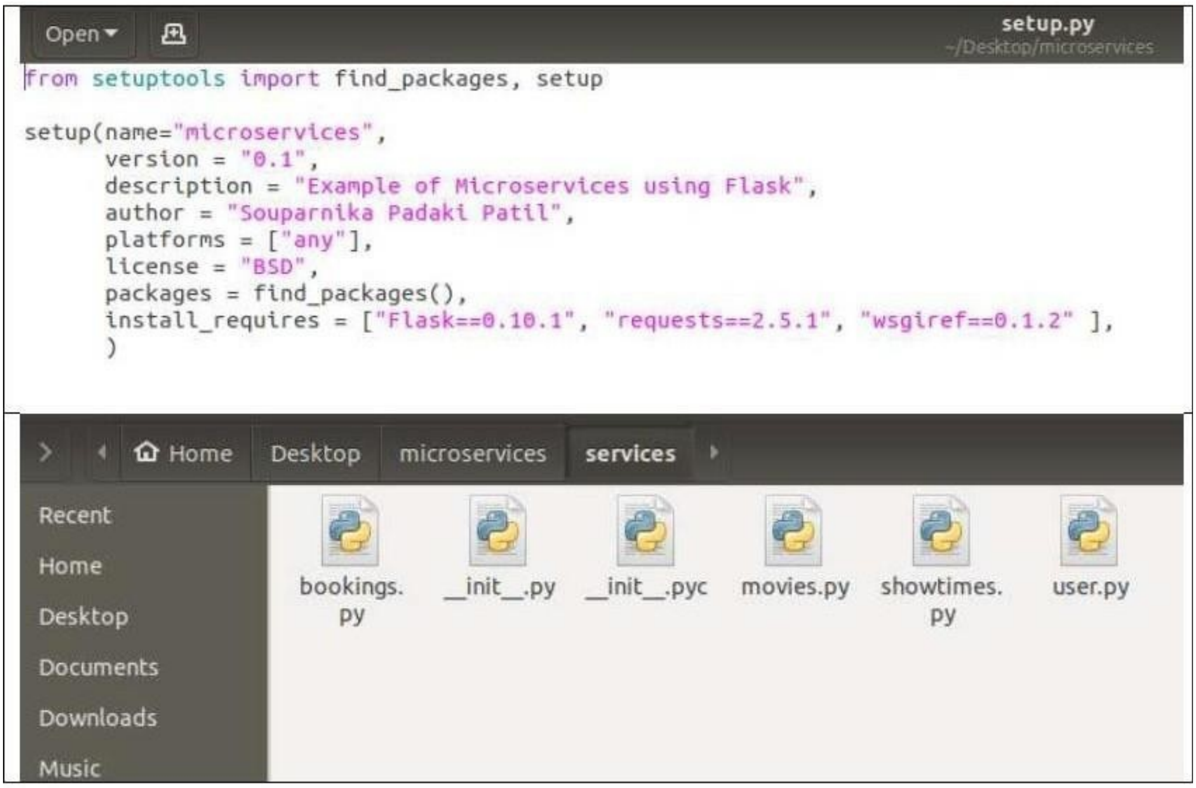
```
@app.route('/hello')
defhello_world():
return 'hello world'
```

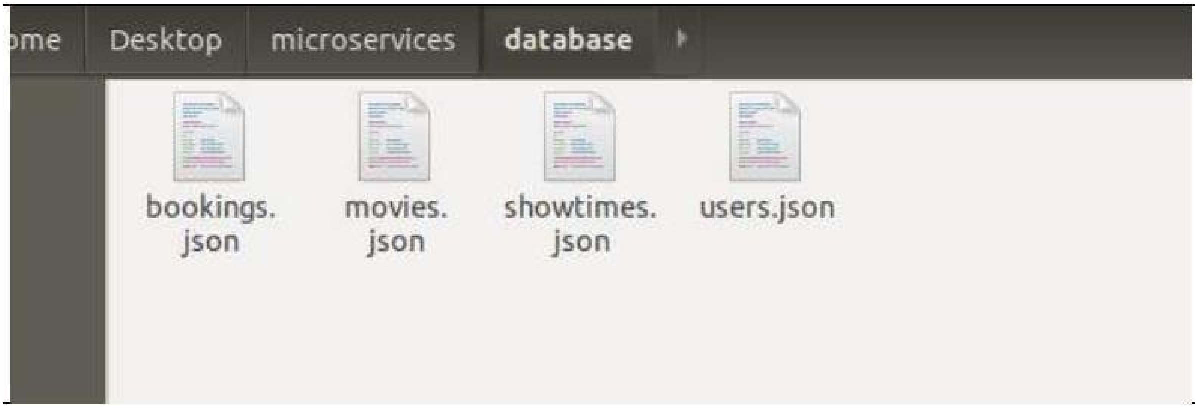
Here, URL '/hello' rule is bound to the hello_world() function. As a result, if a user visitshttp://localhost:5000/hello URL, the output of the hello_world() function will be rendered inthe browser.

3. Writing the subroutine for the four micro services:

There are four micro services viz., User, Show times, Bookings and Movies for which micro services are to be implemented.

Writing the source code:


Open
setup.py
~/Desktop/microservices
from setuptools import find_packages, setup
setup(name="microservices",
version = "0.1",
description = "Example of Microservices using Flask",
author = "Souparnika Padaki Patil",
platforms = ["any"],
license = "BSD",
packages = find_packages(),
install_requires = ["Flask==0.10.1", "requests==2.5.1", "wsgiref==0.1.2"],
)
Home
Desktop
microservices
services
Recent
Home
Desktop
Documents
Downloads
Music
bookings.py
__init__.py
__init__.pyc
movies.py
showtimes.py
user.py


Desktop
microservices
database
bookings.json
movies.json
showtimes.json
users.json

Open movies.json ~/Desktop/microservices/database


```json
{
  "720d006c-3a57-4b6a-b18f-9b713b073f3c": {
    "title": "Happy Phirr Bhag Jayegi",
    "rating": 7.4,
    "director": "Mudassar Aziz",
    "id": "720d006c-3a57-4b6a-b18f-9b713b073f3c"
  },
  "a8034f44-aee4-44cf-b32c-74cf452aaaae": {
    "title": "Stree",
    "rating": 9.2,
    "director": "Amar Kaushik",
    "id": "a8034f44-aee4-44cf-b32c-74cf452aaaae"
  },
  "96798c08-d19b-4986-a05d-7da856efb697": {
    "title": "Gold",
    "rating": 7.4,
    "director": "Reema Kagdi",
    "id": "96798c08-d19b-4986-a05d-7da856efb697"
  },
  "267eedb8-0f5d-42d5-8f43-72426b9fb3e6": {
    "title": "Karwaan",
    "rating": 8.8,
    "director": "Akash Khurana",
    "id": "267eedb8-0f5d-42d5-8f43-72426b9fb3e6"
  },
  "7daf7208-be4d-4944-a3ae-c1c2f516f3e6": {
    "title": "Mission Impossible 6",
    "rating": 9.5,
    "director": "Christopher McQuarrie",
    "id": "7daf7208-be4d-4944-a3ae-c1c2f516f3e6"
  },
  "276c79ec-a26a-40a6-b3d3-fb242a5947b6": {
    "title": "Avengers Infinity War",
    "rating": 9.8,
    "director": "Anthony Russo",
    "id": "276c79ec-a26a-40a6-b3d3-fb242a5947b6"
  },
  "39ab85e5-5e8e-4dc5-afea-65dc368bd7ab": {
    "title": "The Incredibles 2",
    "rating": 7.1,
    "director": "Brad Bird",
    "id": "39ab85e5-5e8e-4dc5-afea-65dc368bd7ab"
  }
}
```


Building and Executing the solution:

1. To install the necessary files and create a virtual environment run:

   sudo ./setup.sh

2. To start the 4 microservicesrun :

   ./startup.sh

3. To start the command line UI:

   python cmdline.py

Running startup.sh

```
dos@dospc                                                    ./startup.sh
dos@dospc                                                     * Running on http://127.0.0.1:
5003/ (Press CTRL+C to quit)
 * Running on http://127.0.0.1:5001/ (Press CTRL+C to quit)
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Restarting with stat
 * Restarting with stat
 * Running on http://127.0.0.1:5002/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger is active!
 * Debugger is active!
 * Debugger PIN: 229-444-055
 * Debugger PIN: 229-444-055
 * Debugger PIN: 229-444-055
 * Debugger is active!
 * Debugger PIN: 229-444-055
127.0.0.1 - - [26/Dec/2018 16:44:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:41] "GET /movies HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:44] "GET /showtimes HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:44] "GET /movies HTTP/1.1" 200 -
127.0.0.1 - - [26/Dec/2018 16:44:53] "GET /bookings/Shreyas HTTP/1.1" 404 -
```

Running cmdline.py

```
dos@dospc                                                    python cmdline.py
Welcome to cinema app
1.Get Movie list
2.Get Show Times
3.Get Bookings Info
4.Get User list
5.Book a show
6.Clearscreen
7.Exit
Select an option
1
ID:  276c79ec-a26a-40a6-b3d3-fb242a5947b6
Title:  Avengers Infinity War
Director:  Anthony Russo
Rating:  9.8

ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae
Title:  Stree
Director:  Amar Kaushik
Rating:  9.2

ID:  7daf7208-be4d-4944-a3ae-c1c2f516f3e6
Title:  Mission Impossible 6
Director:  Christopher McQuarrie
Rating:  9.5
```

```
1.Get Movie list
2.Get Show Times
3.Get Bookings Info
4.Get User list
5.Book a show
6.Clearscreen
7.Exit
Select an option
2
On date: 20180801
ID:  267eedb8-0f5d-42d5-8f43-72426b9fb3e6  MOVIE:  Karwaan
ID:  7daf7208-be4d-4944-a3ae-c1c2f516f3e6  MOVIE:  Mission Impossible 6
ID:  39ab85e5-5e8e-4dc5-afea-65dc368bd7ab  MOVIE:  The Incredibles 2
ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae  MOVIE:  Stree
On date: 20180803
ID:  720d006c-3a57-4b6a-b18f-9b713b073f3c  MOVIE:  Happy Phirr Bhag Jayegi
ID:  39ab85e5-5e8e-4dc5-afea-65dc368bd7ab  MOVIE:  The Incredibles 2
On date: 20180802
ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae  MOVIE:  Stree
ID:  96798c08-d19b-4986-a05d-7da856efb697  MOVIE:  Gold
ID:  39ab85e5-5e8e-4dc5-afea-65dc368bd7ab  MOVIE:  The Incredibles 2
ID:  276c79ec-a26a-40a6-b3d3-fb242a5947b6  MOVIE:  Avengers Infinity War
On date: 20180805
ID:  96798c08-d19b-4986-a05d-7da856efb697  MOVIE:  Gold
ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae  MOVIE:  Stree
ID:  7daf7208-be4d-4944-a3ae-c1c2f516f3e6  MOVIE:  Mission Impossible 6
```

```
1 set movie list
2.Get Show Times
3.Get Bookings Info
              Get User list Book a show Clearscreen

              Select an option


Anuja Khar&tmol
Souparnika Patil
Yasundhara Eurtakoti

Nachiket Ghorpade
Nayana Patil
Kamraj Ambalkar
```

```
1.Get Movie list
2.Get Show Times
3.Get Bookings Info
4.Get User list
5.Book a show
6.Clearscreen
7.Exit
Select an option
5
>Please enter username for the booking : souparnika_patil
>Please enter the date for the booking : 20180805



ID:  96798c08-d19b-4986-a05d-7da856efb697
Title:  Gold
Director:  Reema Kagdi
Rating:  7.4

ID:  a8034f44-aee4-44cf-b32c-74cf452aaaae
Title:  Stree
Director:  Amar Kaushik
Rating:  9.2

ID:  7daf7208-be4d-4944-a3ae-c1c2f516f3e6
```

```
      Z9ah85*5—     —     —     —
          The Increfibles
Director:  Brzd
Rating:  7.1

      2?6c79ec
          Avengers Infinity
Director:  Anthony Russo
Rat:i n q !   9.B

>Enter Chs id for tha booking   27£c79*o—a26a—40aG—b3d3




Booking the show for the following movie on oats 201SDBJ2
IO:  276c73ec-a26a-40ad-b3d3-fbZ42abS47b6
Title•  Avengers Infinity
E1 x e o- c•z :   An n isn ny Russ c›
Rat• ñ.riq !   9. R

Press        to continue

BOOKING JONEI l Thdnk you for using Cinemz app
```

Conclusion:

With micro services, modules within software can be independently deployable. In amicroservices architecture, each service runs a unique process and usually manages its owndatabase. This not only provides development teams with a more decentralized approach tobuilding software, it also allows each service to be deployed, rebuilt, and redeployed and managed independently. Netflix, eBay, Amazon, The UK Government Digital Service, Twitter, PayPal,The Guardian, and many other large-scale websites and applications have all evolved from monolithic to micro services architecture.