

D. Y. Patil college of Engineering, Akurdi, Pune-44
Department of Information Technology

LAB PRACTICE-V

SUBJECT CODE: 414454

CLASS: FINAL YEAR (SEMESTER-II)

LAB MANUAL

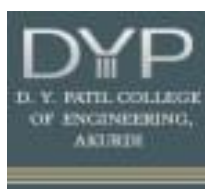
LAB PRACTICE-V

SUBJECT CODE: 414454

CLASS: FINAL YEAR

SEMESTER - II

LAB MANUAL

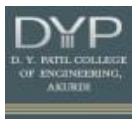


D. Y. PATIL COLLEGE OF ENGINEERING, AKURDI, PUNE-44

DEPARTMENT OF INFORMATION TECHNOLOGY

INDEX

Sr.No	Details
1.	Vision & Mission of Institute
2.	Vision & Mission of Department
3.	Program Educational Objectives (PEOs), Program Specific Outcomes (PSO)
4.	Program Outcomes [POs]
5.	Course Objective, Course Outcomes [COs]
6.	List of Assignments
7.	Rubrics For Laboratory Assessment
8.	Assignment No 1
9.	Assignment No 2
10.	Assignment No 3
11.	Assignment No 4
12.	Assignment No 5
13.	Assignment No 6
15.	Assignment No 7



D. Y. Patil college of Engineering, Akurdi, Pune-44

Department of Information Technology

Vision of the Institute:

“Empowerment through Knowledge”.

Mission of the Institute:

To educate the students to transform them as professionally competent and quality conscious engineers by providing conducive environment for teaching, learning, and overall personality development, culminating the institute into an international seat of excellence.

Vision of the Department

Developing globally competent IT professional for sustainable growth of humanity.

Mission of the Department

- M1** Build a strong foundation and techniques for problem-solving and inculcate communication skills as an integral component of Information Technology
- M2** Develop competency skills in the faculty members and students to serve the societal challenges and needs in lieu of its multidisciplinary applications in the field of Information Technology
- M3** Encourage development of strong technical skills and knowledge and to encourage students to undergo research in the field of Information Technology
- M4** Nurture students to become ethical and committed lifelong IT professionals
- M5** Empower students with strong decision-making skills and technical competency to accomplish start-up ideas in the field of IT Engineering

Program Educational Objectives (PEOs) defined by Department of Information Technology D Y Patil college of Engineering, Akurdi, Pune :

- 1) **Core Competency:** To provide graduates with a solid foundation in Mathematics, Science, Engineering fundamentals required to solve complex Software Engineering Problem.
- 2) **Breadth:** To impart the knowledge and skills in the field of Information Technology; and to comprehend, analyze, design and create novel products and solutions for the real-time and Complex Engineering problems of any domain with innovative approaches.
- 3) **Professionalism:** To inculcate in graduates, professional and ethical values, effective communication skills, teamwork, multidisciplinary approach, and ability to relate engineering issues in broader social context.
- 4) **Learning Environment:** To provide graduates with an academic environment that makes them aware of excellence in leadership, presentation, time management and ethics leading them to become responsible and competent professionals prepared to address challenges in the field of IT at global level.
- 5) **Attainment:** To empower graduates with an attitude and skills of Research, Entrepreneur and Higher education in the field of Information Technology.

Program Specific Outcomes (PSO) defined by Department of Information Technology D Y Patil college of Engineering, Akurdi, Pune :

- 1) Apply design methodologies, application development tools, engineering skills in Software Engineering Domains and IT Application areas like Cloud Computing, Software Testing, Mobile App Development, etc.
- 2) Aspire to pursue Higher Education in the specialized fields of IT Engineering and management programs like Data science, Cyber Security, Artificial Intelligence, etc.
- 3) Formulate decision-making skills, IT Engineering skills, and knowledge to implement start-up ideas as an entrepreneur in the fields such as Cyber Security, Mobile Application Development, etc.
- 4) Devise to design, implement, and evaluate IT based software systems to serve the needs of society or IT industries at large.

PROGRAM OUTCOME (PO)

Students are expected to know and be able to–		
PO1	Engineering knowledge	An ability to apply knowledge of mathematics, computing, science, engineering and technology.
PO2	Problem analysis	An ability to define a problem and provide a systematic solution with the help of conducting experiments, analyzing the problem and interpreting the data.
PO3	Design / Development of Solutions	An ability to design, implement, and evaluate software or a software /hardware system, component, or process to meet desired needs within realistic constraints.
PO4	Conduct Investigation of Complex Problems	An ability to identify, formulate, and provide essay schematic solutions to complex engineering /Technology problems.
PO5	Modern Tool Usage	An ability to use the techniques, skills, and modern engineering technology tools, standard processes necessary for practice as a IT professional.
PO6	The Engineer and Society	An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer- based systems with necessary constraints and assumptions.
PO7	Environment and Sustainability	An ability to analyze and provide solution for the local and global impact of information technology on individuals, organizations and society.
PO8	Ethics	An ability to understand professional, ethical, legal, security and social issues and responsibilities.
PO9	Individual and Team Work	An ability to function effectively as an individual or as a team member to accomplish a desired goal(s).
PO10	Communication Skills	An ability to engage in life-long learning and continuing professional development to cope up with fast changes in the technologies /tools with the help of electives, profession along animations and extra- curricular activities.
PO11	Project Management and Finance	An ability to communicate effectively in engineering community at large by means of effective presentations, report writing, paper publications, demonstrations.
PO12	Life-long Learning	An ability to understand engineering, management, financial aspects, performance, optimizations and time complexity necessary for professional practice.

COURSE OBJECTIVE

CO1	The course aims to provide an understanding of the principles on which the distributed systems are based, their architecture, algorithms and how they meet the demands of Distributed applications
CO2	The course covers the building blocks for a study related to the design and the implementation of distributed systems and applications.

COURSE OUTCOMES

On completion of the course, students will be able to-	
CO1	Demonstrate knowledge of the core concepts and techniques in distributed systems
CO2	Learn how to apply principles of state-of-the-Art Distributed systems in practical application.
CO3	Design, build and test application programs on distributed systems

List of Assignments

1. Implement multi-threaded client/server Process communication using RMI.
2. Develop any distributed application using CORBA to demonstrate object brokering. (Calculator or String operations).
3. Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.
4. Implement Berkeley algorithm for clock synchronization.
5. Implement token ring based mutual exclusion algorithm.
6. Implement Bully and Ring algorithm for leader election.
7. Create a simple web service and write any distributed application to consume the web service.
8. Mini Project (In group): A Distributed Application for Interactive Multiplayer Games

RUBRICS FOR LABORATORY ASSESSMENT

1. Attendance

Assessment Outcome \Rightarrow	Poor (1)	Satisfactory(2)	Good (3)	Very Good (4)	Excellent (5)
Dimensions \Downarrow					
1.Attendance with Involvement of Student (5M)	Passive observer	Very little involvement	Good Involvement in performing experiment	Individual Involvement in performing experiment	Individual and self - Involvement in performing experiment

2. Viva

Assessment Outcome \Rightarrow	Poor (1)	Satisfactory(2)	Good (3)	Very Good (4)	Excellent (5)
Dimensions \Downarrow					
1.Preparation and Basic Knowledge (5M)	No preparation	Little Knowledge	Prepared Well	Very well prepared	Advance Knowledge
2.Program development and execution (5M)	Not Executed	Partially executed	Executed	Executed without additional modification	Executed with additional modification
3.Punctuality and Ethics (5M)	Attendance Below 50% and not following the lab instructions	Attendance 50% to 75% And sometimes copies the program	Regular attendance 75-00% and follows the instruction and try to perform on his own	Regular attendance 80-90% and follows the instruction and try to perform on his own	90-100 % attendance, follows all instructions and execute the program on his own

3. Presentation

Assessment Outcome \Rightarrow	Poor (1)	Satisfactory(2)	Good (3)	Very Good (4)	Excellent (5)
Dimensions \Downarrow					
Journal Presentation (5M)	Not Prepared	Incomplete	Completed documentation	well documented	Very well documented

Outcome: Student will be able to

- i. Apply knowledge to real life examples and develop practical approach
- ii. Design Basic Application.

Note: Students with poor marks should repeat the assignment

ASSIGNMENT NO.1 A**Problem Statement 1a):**

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

Tools / Environment:

Java Programming Environment, rmiregistry, 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.

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 **ServerSocket** class offers to observe connection invocations, and it accepts such invocations from different clients through another socket. High-level wrapper classes, such as **URLConnection** and **URLEncoder**, are more appropriate. If you want to establish a connection to the Web using a URL, then these classes will use the socket internally.

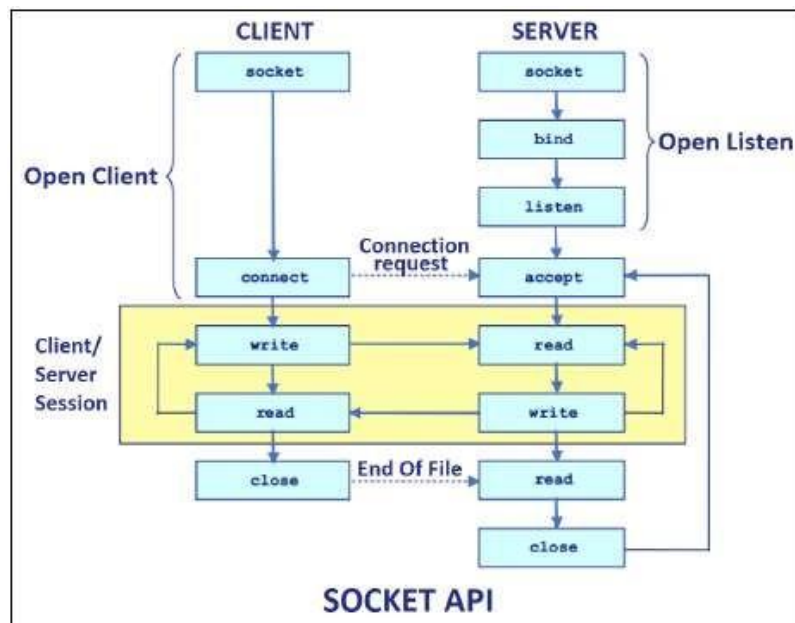
The **java.net** package provides support for the two common network protocols –

- **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 `ServerSocket` object, denoting which port number communication is to occur on.
- The server invokes the `accept()` method of the `ServerSocket` 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 `OutputStream` and an `InputStream`. The client's `OutputStream` is connected to the server's `InputStream`, and the client's `InputStream` is connected to the server's `OutputStream`.
- 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. DatagramPacket objects are used to send data over DatagramSocket. Every DatagramPacket 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 ServerSocket 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.

Implementation of Socket Programming for UDP is required by the students.

Writing the source code:

The socket program using TCP will demonstrate how data can be sent and received between the client and the server.

The following example is a client / server application making use of the Socket class to listen to clients on a port number specified by a command-line argument. The server program doubles the number sent by the client.

```

Client.java  *Server.java
1 import java.io.BufferedReader;
9
10 public class Server {
11
12     private static Socket socket;
13
14     public static void main(String[] args) {
15         try {
16             int port = 25000;
17             ServerSocket serverSocket = new ServerSocket(port);
18             System.out.println("Server Started and listening to the port 25000");
19
20             // Server is running always. This is done using this while(true) loop
21             while (true) {
22                 // Reading the message from the client
23                 socket = serverSocket.accept();
24                 InputStream is = socket.getInputStream();
25                 InputStreamReader isr = new InputStreamReader(is);
26                 BufferedReader br = new BufferedReader(isr);
27                 String number = br.readLine();
28                 System.out.println("Message received from client is " + number);
29
30                 // Multiplying the number by 2 and forming the return message
31                 String returnMessage;
32                 try {
33                     int numberInIntFormat = Integer.parseInt(number);
34                     int returnValue = numberInIntFormat * 2;
35                     returnMessage = String.valueOf(returnValue) + "\n";
36                 } catch (NumberFormatException e) {
37                     // Input was not a number. Sending proper message back to client.
38                     returnMessage = "Please send a proper number\n";
39                 }
40
41                 // Sending the response back to the client.
42                 OutputStream os = socket.getOutputStream();
43                 OutputStreamWriter osw = new OutputStreamWriter(os);
44                 BufferedWriter bw = new BufferedWriter(osw);
45                 bw.write(returnMessage);
46                 System.out.println("Message sent to the client is " + returnMessage);
47                 bw.flush();
48             }
49         } catch (Exception e) {
50             e.printStackTrace();
51         } finally {
52             try {
53                 socket.close();
54             } catch (Exception e) {
55             }
56         }
57     }
58 }

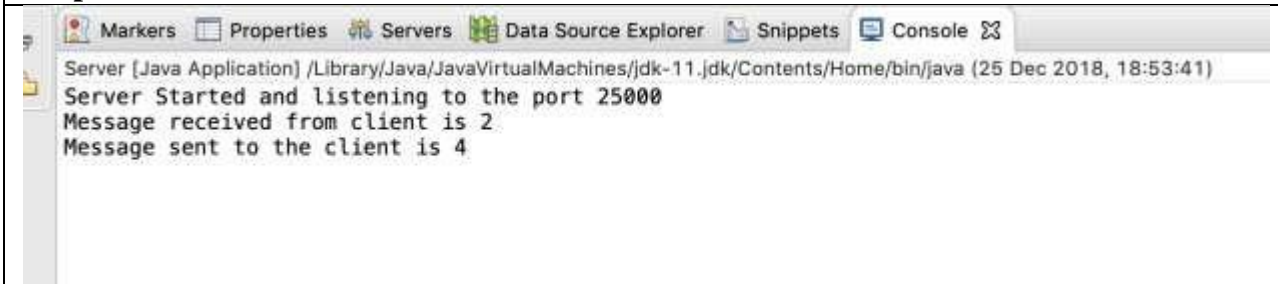
```

Client.java

```
Client.java *Server.java
1 import java.io.BufferedReader;
9
10 public class Client
11 {
12
13     private static Socket socket;
14
15     public static void main(String args[])
16     {
17         try
18         {
19             String host = "localhost";
20             int port = 25000;
21             InetAddress address = InetAddress.getByName(host);
22             socket = new Socket(address, port);
23
24             //Send the message to the server
25             OutputStream os = socket.getOutputStream();
26             OutputStreamWriter osw = new OutputStreamWriter(os);
27             BufferedWriter bw = new BufferedWriter(osw);
28
29             String number = "2";
30
31             String sendMessage = number + "\n";
32             bw.write(sendMessage);
33             bw.flush();
34             System.out.println("Message sent to the server : "+sendMessage);
35
36             //Get the return message from the server
37             InputStream is = socket.getInputStream();
38             InputStreamReader isr = new InputStreamReader(is);
39             BufferedReader br = new BufferedReader(isr);
40             String message = br.readLine();
41             System.out.println("Message received from the server : " +message);
42         }
43         catch (Exception exception)
44         {
45             exception.printStackTrace();
46         }
47         finally
48         {
49             //Closing the socket
50             try
51             {
52                 socket.close();
53             }
54             catch (Exception e)
55             {
56                 e.printStackTrace();
57             }
58         }
59     }
60 }
```

Compilation and Executing the solution:**If you're using Eclipse :**

1. Compile both of them on two different terminals or tabs
2. Run the Server program first
3. Then run the Client program
4. Type messages in the Client Window which will be received and showed by the Server Window simultaneously if you are developing echo server application.
5. Close the socket connection by typing something like "Exit".

Output:A screenshot of the Eclipse IDE's Console window. The window title bar shows tabs for Markers, Properties, Servers, Data Source Explorer, Snippets, and Console. The Console content area displays the following text: "Server [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.jdk/Contents/Home/bin/java (25 Dec 2018, 18:53:41)", "Server Started and listening to the port 25000", "Message received from client is 2", and "Message sent to the client is 4".

```
Server [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.jdk/Contents/Home/bin/java (25 Dec 2018, 18:53:41)
Server Started and listening to the port 25000
Message received from client is 2
Message sent to the client is 4
```

Conclusion:

In this assignment, the students learned about client-server communication through different protocols and sockets. They also learned about Java support through the socket API for TCP and UDP programming.

ASSIGNMENT NO. 1 B**Problem Statement:**

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

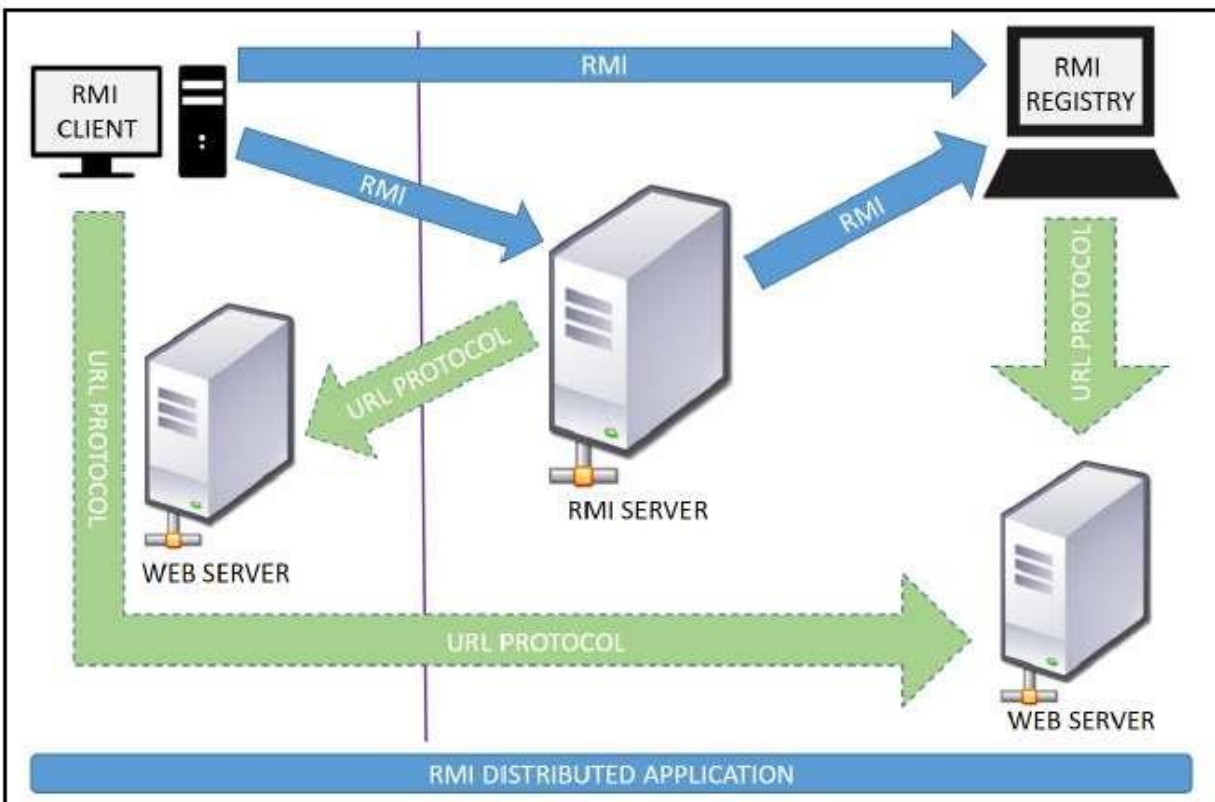
Tools / Environment:

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 makes a 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 any outgoing requests. It exists on the client JVM and represents the handle to the remote object.

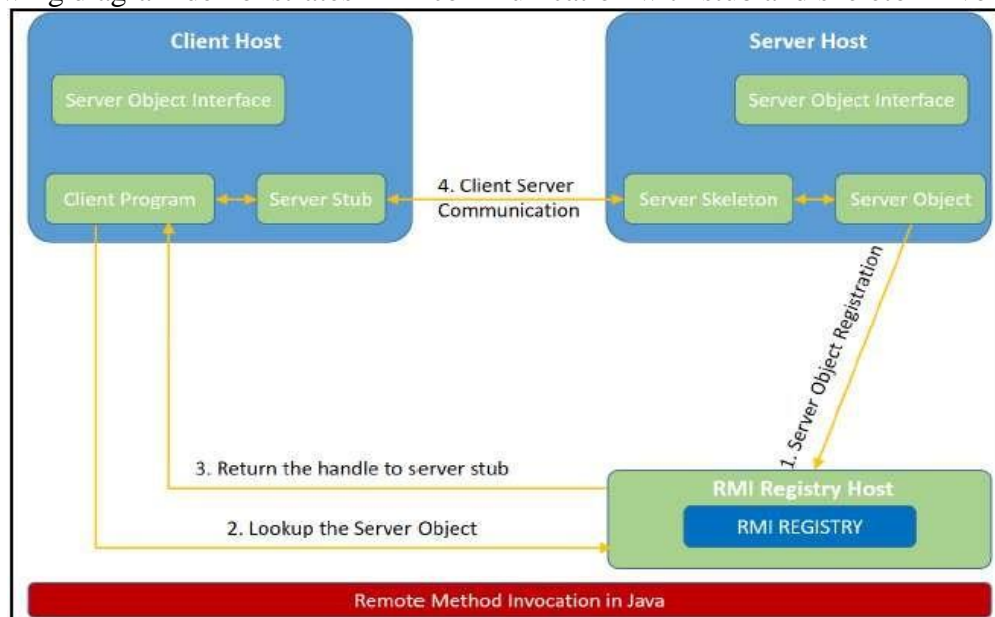
If any object invokes a method on the stub object, the stub establishes RMI by following these steps:

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.

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 by all 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 object class 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.

Let's design a project that can sit on a server. After that different client projects interact with this project to pass the parameters and get the computation on the remote object execute and return the result to the client components.

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 `rmic` 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, as shown here:

```
start rmiregistry
```

5. Start the Server

The server code is started from the command line, as shown here:

```
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. You may invoke it from the command line by using one of the two formats shown here:

```
java AddClient 192.168.13.14 7 8
```

Writing the source code:

```

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

```

```

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;
    }
}

```

```

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

```

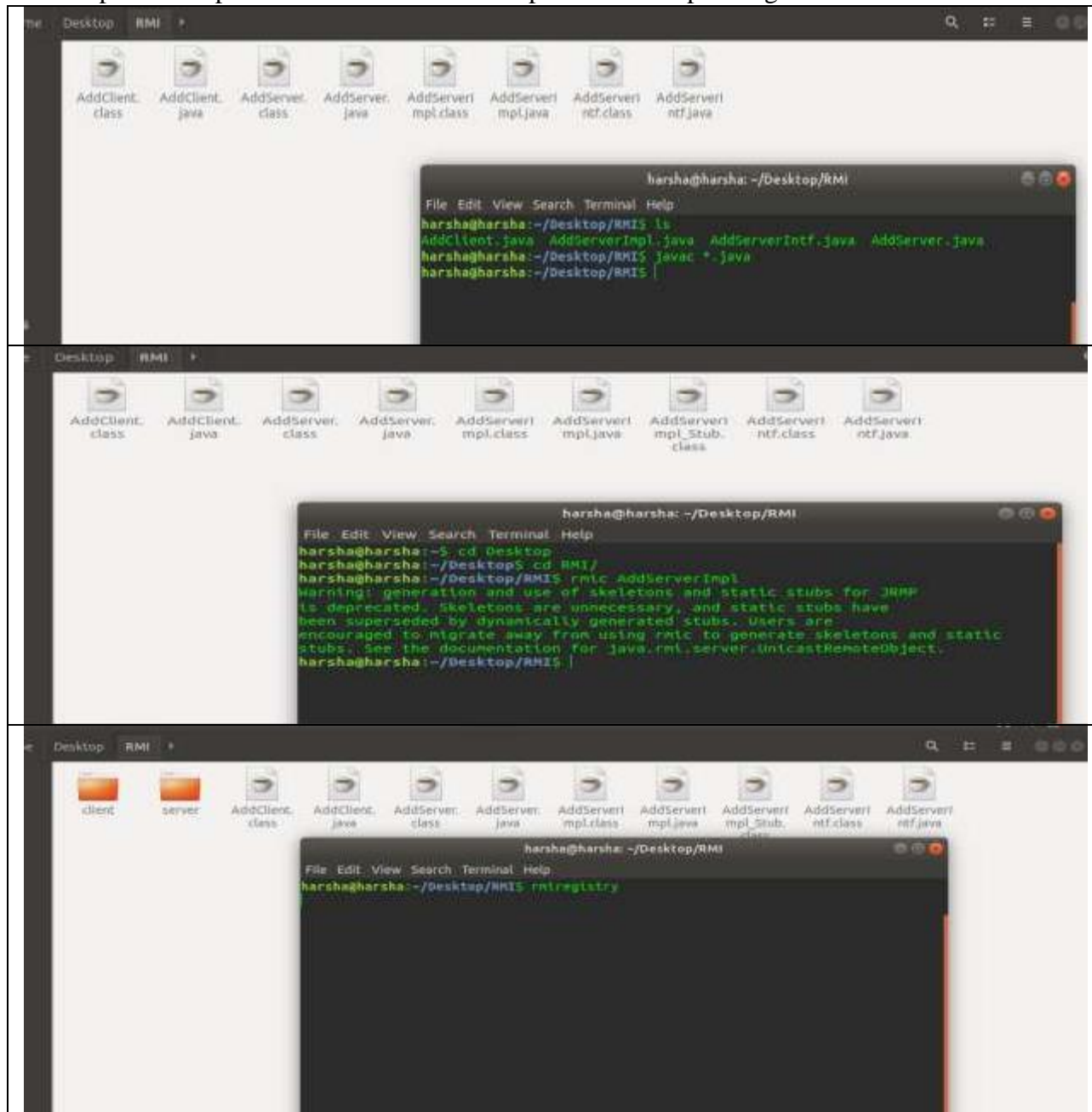
```

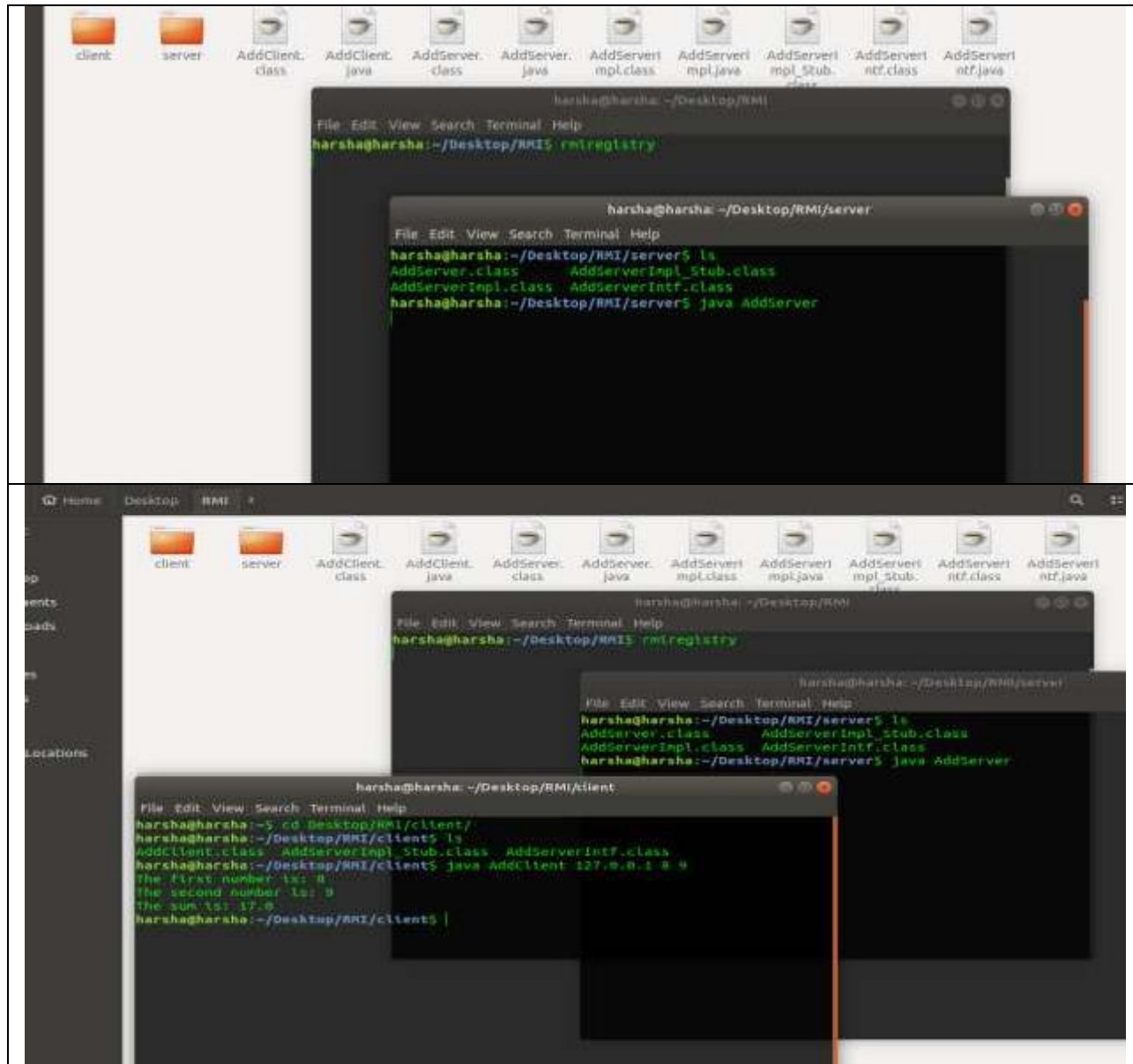
import java.rmi.*;
public class AddClient {
    public static void main(String args[]) {
        try {
            String addServerURL = "rmi://" + args[0] + "/AddServer";
            AddServerIntf addServerIntf =
                (AddServerIntf)Naming.lookup(addServerURL);
            System.out.println("The 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);
        }
    }
}

```

Compilation and Executing the solution:

The steps for compilation and execution are captured in a snapshots given:





Conclusion:

Remote Method Invocation (RMI) allows you 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 you to build distributed applications.

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 (mpj.jar)

Related 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 message passing.

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:

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.

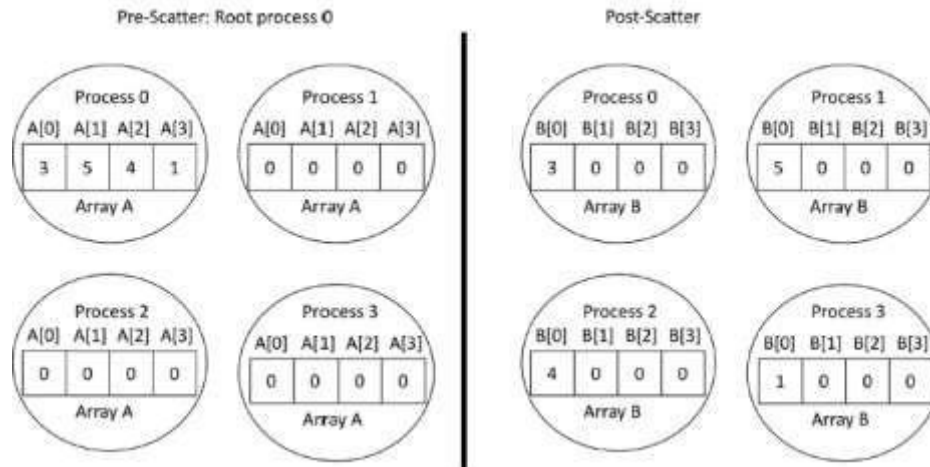
The receive command (MPI_Recv) describes both the data to be transferred and the connection to be established. The MPI_Recv construct is structured as follows:

```
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.



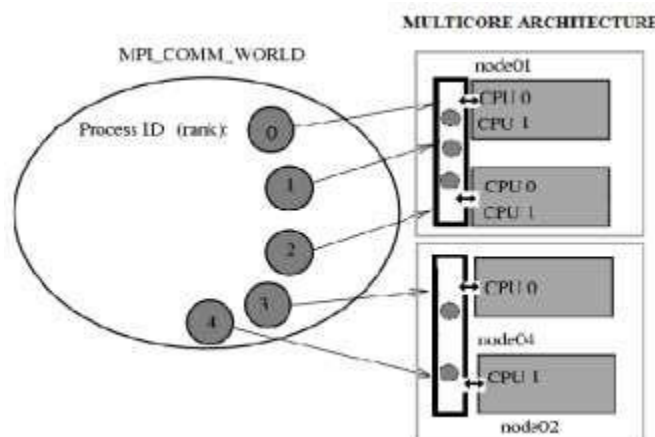
MPJ Express is an open source Java message passing library that allows application developers to write and execute parallel applications **for multicore processors and compute clusters / clouds**. The software is distributed under the MIT (a variant of the LGPL) license. MPJ Express is a message passing 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 expect 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:

1. For implementing the MPI program in multi-core environment, we need to install MPJ express library.
 - a. Download MPJ Express (mpj.jar) and unpack it.
 - b. Set MPJ_HOME and PATH environment variables:
 - c. export MPJ_HOME=/path/to/mpj/
 - d. export PATH=\$MPJ_HOME/bin:\$PATH
2. Write Hello World parallel Java program and save it as HelloWorld.java (Assign2.java).
3. Compile a simple Hello World (Assign) parallel Java program
4. Running MPJ Express in the Multi-core Configuration.

Writing the source code:

Assign2.java

```
import mpi.*;
public class Assign2 {

    public static void main(String args[]) throws Exception {
        MPI.Init(args);
        int me = MPI.COMM_WORLD.Rank();
        int size = MPI.COMM_WORLD.Size();
        System.out.println("Hi from <"+me+">");
        MPI.Finalize();
    }
}
```

Compiling and Executing the solution:

Compile: javac -cp \$MPJ_HOME/lib/mpj.jar Assign2.java
(mpj.jar is inside lib folder in the downloaded MPJ Express)

Execute: \$MPJ_HOME/bin/mpjrun.sh -np 4 Assign2

```
dos@dospc ~/Desktop/Junaid/Assign2 mpjrun.sh -np 2 Assign2
MPJ Express (0.44) is started in the multicore configuration
Hi from <0>
Hi from <1>

dos@dospc ~/Desktop/Junaid/Assign2 mpjrun.sh -np 4 Assign2
MPJ Express (0.44) is started in the multicore configuration
Hi from <3>
Hi from <1>
Hi from <2>
Hi from <0>
```

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 multicore architecture so that user can develop the code on it's own laptop or desktop. This is an effort to develop and run parallel programs according to MPI standard.

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.

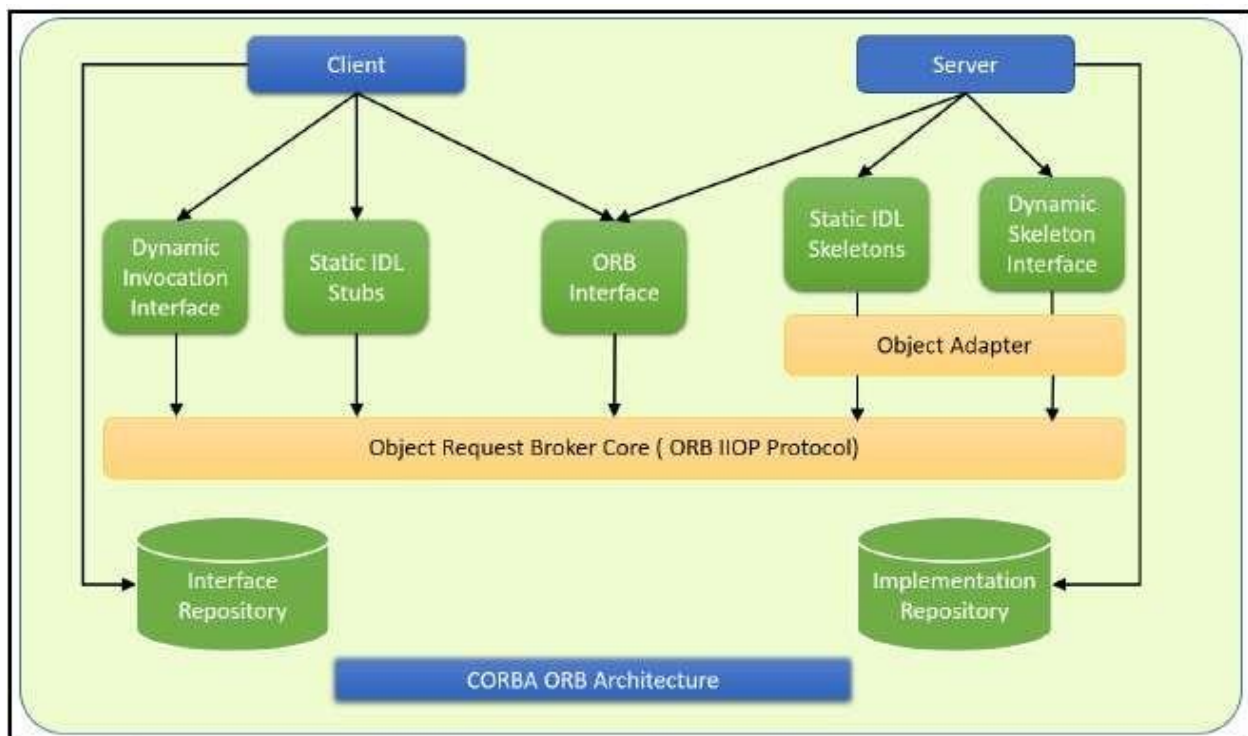
Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR&Benefits" maintain an object model with details of the organization, employees with demographic information, benefits, payroll, and deductions. 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 IDLscript. 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.

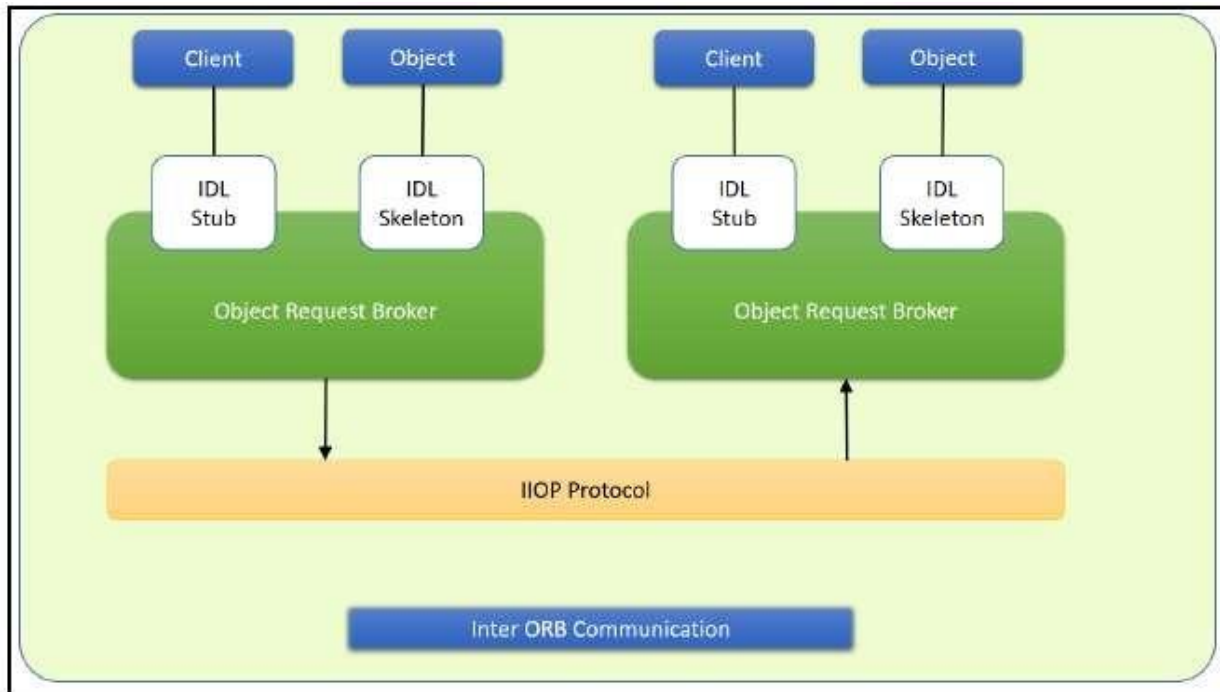
The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons. The objects are written (on the right) and a client for it (on the left), as represented in the diagram. The client and server use stubs and skeletons as proxies, respectively. 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 its 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

Now let's tie all these basic elements together. 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 `idltojava` tool generate 2 other classes:

- A **client stub** class, called `_interface-nameStub`, 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-nameImplBase`, that is a base class for a server-side 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 `idltojava` 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**)

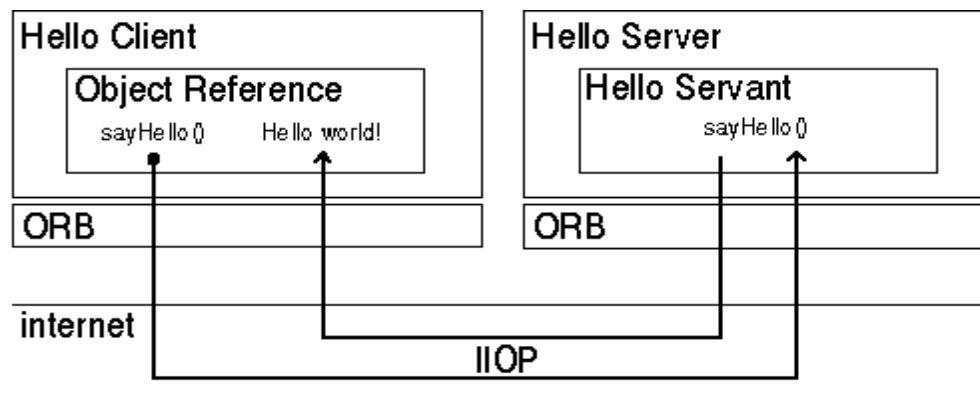
The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL). To complete the application, you simply provide the server (**HelloServer.java**) and client (**HelloClient.java**) implementations.

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.

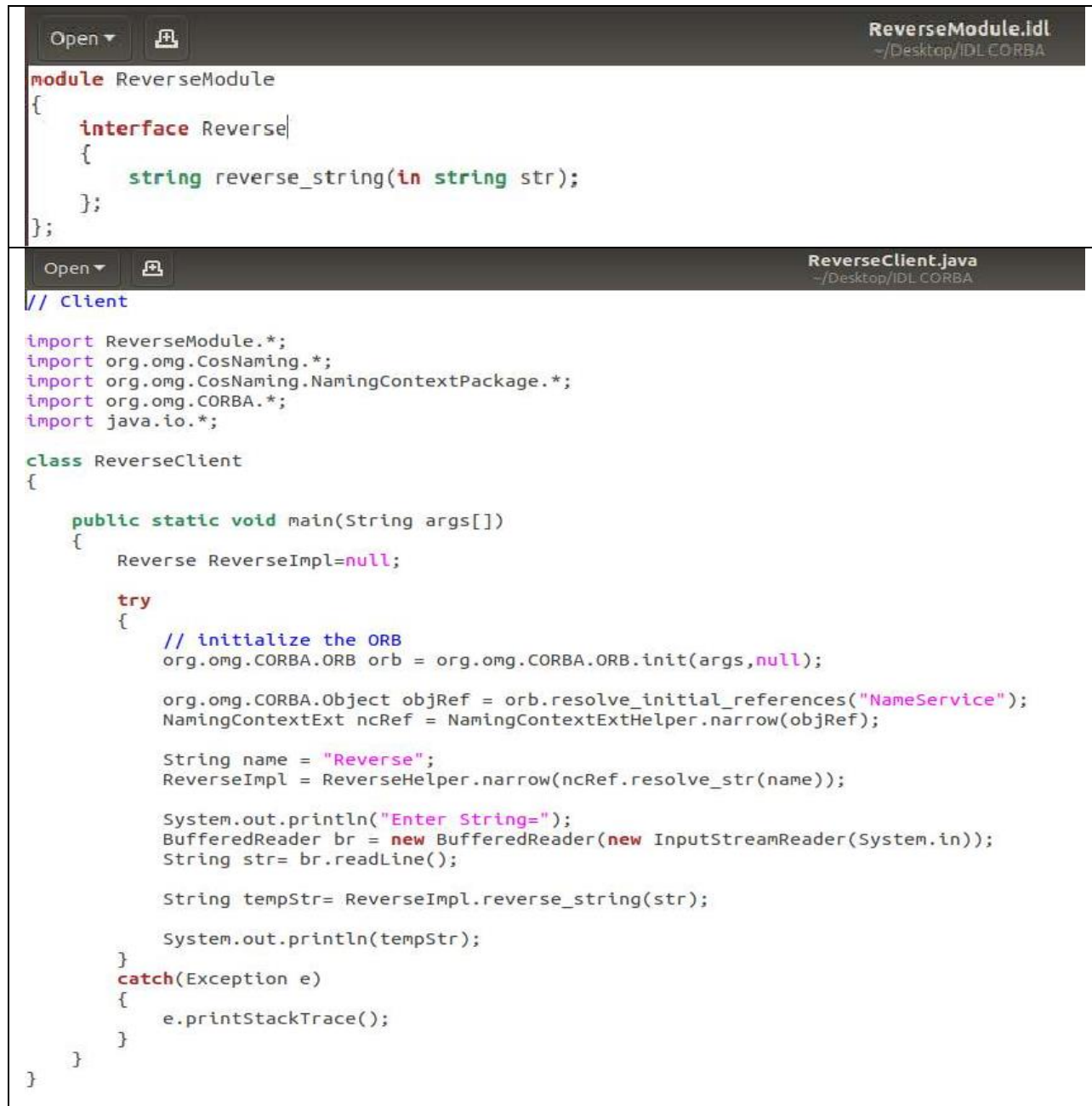


3. Implementing the Client Application (HelloClient.java)

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.

Writing the source code:



```

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

// Client
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();
        }
    }
}
  
```

Open

ReverseImpl.java
~/Desktop/IDL CORBA

```

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);
    }
}

```

Open

ReverseServer.java
~/Desktop/IDL CORBA

```

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();
        }
    }
}

```

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 **orbld**, 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 server-side 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:

- o `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`.
- o `_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.
- o `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`.

- o `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.

- o `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`.

- o `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 `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&
```

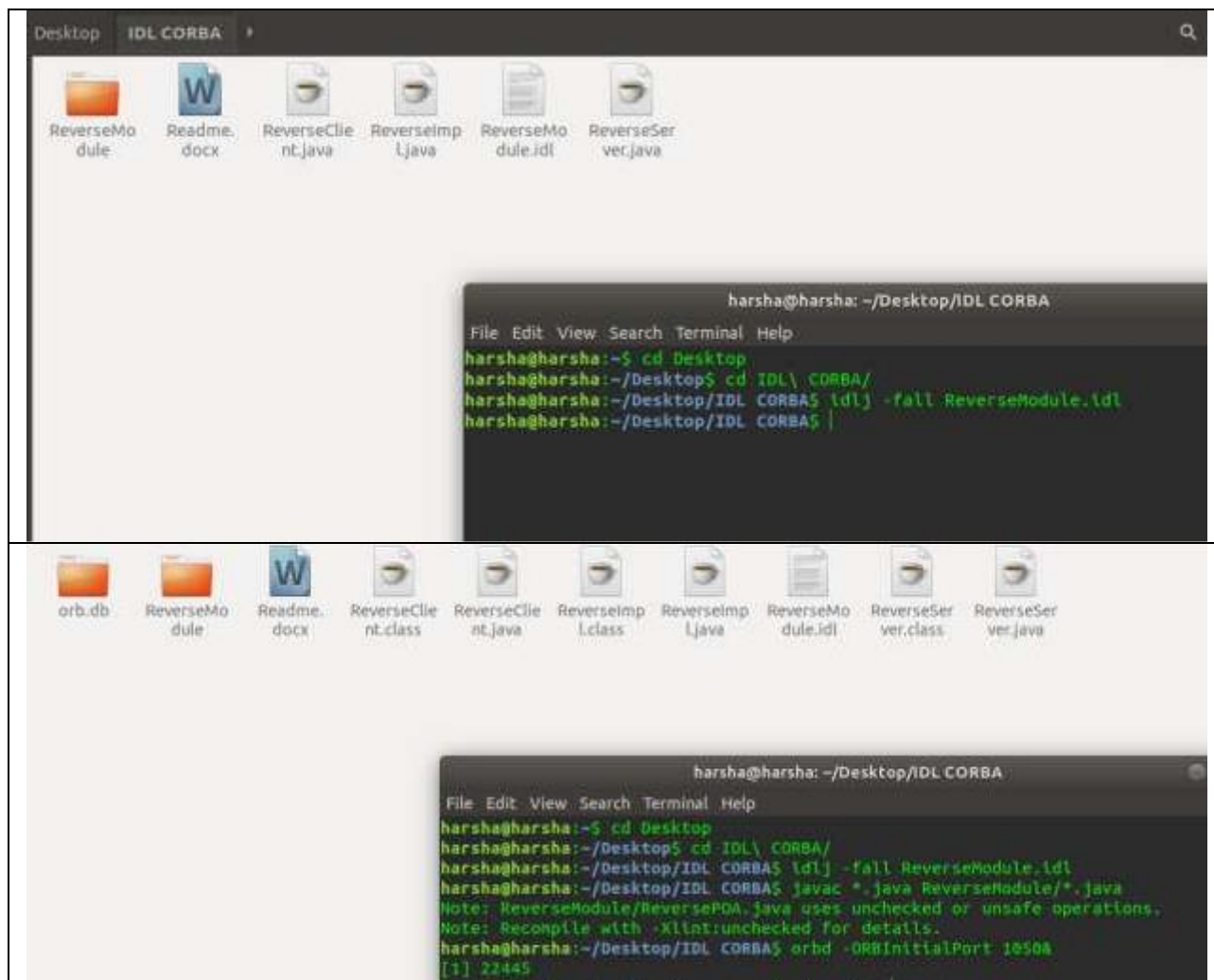
You will see HelloServer ready and waiting... when the server is started.

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).



```

harsha@harsha: ~/Desktop/IDL CORBA
File Edit View Search Terminal Help
harsha@harsha:~/Desktop/IDL CORBA$ java ReverseClient -ORBInitialPort 10500 -ORB
InitialHost localhost
Enter String:
hi hello there
Server Send ereht allah ih
harsha@harsha:~/Desktop/IDL CORBA$

harsha@harsha:~/Desktop/IDL CORBA
File Edit View Search Terminal Help
harsha@harsha:~/Desktop$ cd Desktop
harsha@harsha:~/Desktop$ cd IDL\ CORBA\
harsha@harsha:~/Desktop/IDL CORBA$ idl -Fall ReverseModule.idl
harsha@harsha:~/Desktop/IDL CORBA$ javac *.java ReverseModule/*.java
Note: ReverseModule/ReversePBA.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
harsha@harsha:~/Desktop/IDL CORBA$ orbd -ORBInitialPort 10500
[1] 22445
harsha@harsha:~/Desktop/IDL CORBA$ java ReverseServer -ORBInitialPort 10500 -ORB
InitialHost localhost
[2] 22486
[3] 22487
harsha@harsha:~/Desktop/IDL CORBA$ -ORBInitialHost: command not found
Reverse Object Created
Step1
Step2
Step3
Step4
Reverse Server reading and waiting...

```

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.

ASSIGNMENT NO.4

Problem Statement:

To develop Token Ring distributed algorithm for leader election.

Tools / Environment:

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

Related Theory:

Election Algorithm:

1. Many distributed algorithms require a process to act as a coordinator.
2. The coordinator can be any process that organizes actions of other processes.
3. A coordinator may fail.
4. How is a new coordinator chosen or elected?

Assumptions:

Each process has a unique number to distinguish them. Processes know each other's process number.

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 apart 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 take over the coordinators job.

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

Ring Algorithm:

Initiation:

1. When 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 goes to 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: Message comes back to initiator.

In the queue the **process with maximum ID Number wins**.

Initiator announces the winner by sending another message around the ring.

Designing the solution:

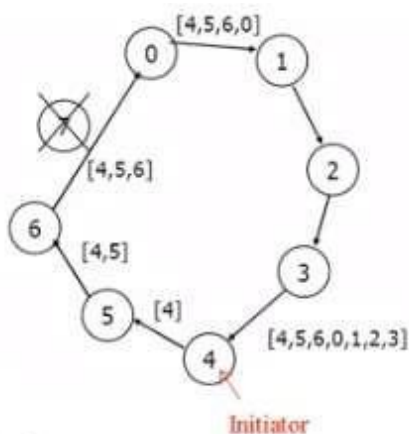
A. For Ring Algorithm

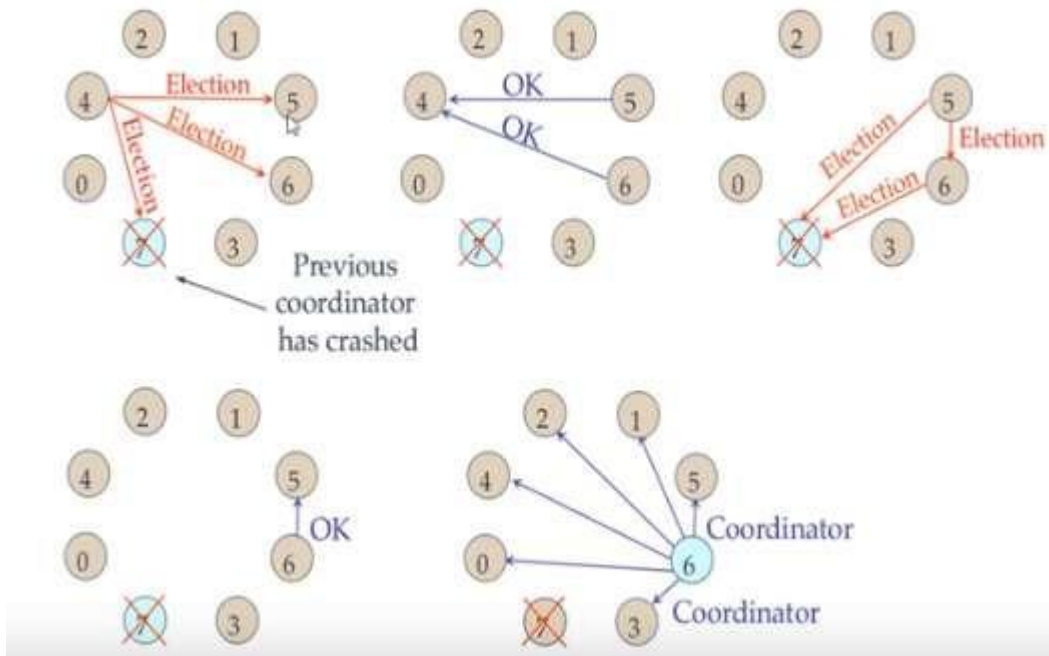
Initiation:

1. Consider the Process 4 understands that Process 7 is not responding.
2. Process 4 initiates the Election by sending "ELECTION" message to its successor (or next alive process) with its ID.

Leader Election:

3. Messages comes back to initiator. Here the initiator is 4.
4. Initiator announces the winner by sending another message around the ring. Here the process with highest process ID is 6. The initiator will announce that Process 6 is Coordinator.



B. For Bully Algorithm:**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.Exit
7. Ask for initializing election process.
8. These inputs will be used by Ring Algorithm.

Writing the source code:

Bully.java

```

1  import java.io.InputStream;
2  import java.io.OutputStream;
3  import java.io.PrintStream;
4  import java.util.Scanner;
5
6  public class Bully {
7      static boolean[] state = new boolean[5];
8      int coordinator;
9
10     public static void up(int up) {
11         if (state[up - 1]) {
12             System.out.println("process " + up + "is already up");
13         } else {
14             int i;
15             Bully.state[up - 1] = true;
16             System.out.println("process " + up + "held election");
17             for (i = up; i <= 5; ++i) {
18                 System.out.println("election message sent from process" + up + "to p" + i);
19             }
20             for (i = up + 1; i <= 5; ++i) {
21                 if (!state[i - 1]) continue;
22                 System.out.println("alive message send from process" + i + "to process" + up);
23                 break;
24             }
25         }
26     }
27
28     public static void down(int down) {
29         if (!state[down - 1]) {
30             System.out.println("process " + down + "is already down.");
31         } else {
32             Bully.state[down - 1] = false;
33         }
34     }
35
36     public static void mess(int mess) {
37         if (state[mess - 1]) {
38             if (state[4]) {
39                 System.out.println("OK");
40             } else if (!state[4]) {
41                 int i;
42                 System.out.println("process" + mess + "election");
43                 for (i = mess; i <= 5; ++i) {
44                     System.out.println("election message sent from process" + mess + "to p" + i);
45                 }
46             }
47         }
48     }
49 }

```

Bully (1) [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (29-Dec-2018, 7:13)
 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

 1 up a process.
 2 down a process
 3 send a message
 4.Exit

 which process will send message
 process2election
 election send from process2to process 3
 election send from process2to process 4
 election send from process2to process 5
 Coordinator message send from process4to all

 1 up a process.
 2 down a process
 3 send a message
 4.Exit

Ring.java

The screenshot shows the Eclipse IDE with the file 'Ring.java' open. The code implements a Ring election algorithm. It starts by importing 'java.util.Scanner'. The 'Ring' class has a 'main' method that initializes an array of processes, takes input for the number of processes and their IDs, and then sorts them by ID. The console output shows the execution of the program, including prompts for the number of processes and their IDs, the sorted list of processes, and the election results where process 8 is selected as the coordinator.

```

1 import java.util.Scanner;
2
3 public class Ring {
4
5     public static void main(String[] args) {
6
7         // TODO Auto-generated method stub
8
9         int temp, i, j;
10        char str[] = new char[10];
11        Rr proc[] = new Rr[10];
12
13        // object initialisation
14        for (i = 0; i < proc.length; i++)
15            proc[i] = new Rr(i);
16
17        // scanner used for getting input from console
18        Scanner in = new Scanner(System.in);
19        System.out.println("Enter the number of process : ");
20        int num = in.nextInt();
21
22        // getting input from users:
23        for (i = 0; i < num; i++) {
24            proc[i].index = i;
25            System.out.println("Enter the id of process : ");
26            proc[i].id = in.nextInt();
27            proc[i].state = "active";
28            proc[i].f = 0;
29        }
30
31
32        // sorting the processes from on the basis of id
33        for (i = 0; i < num - 1; i++) {
34            for (j = 0; j < num - 1; j++) {
35                if (proc[j].id > proc[j + 1].id) {
36                    temp = proc[j].id;
37                    proc[j].id = proc[j + 1].id;
38                    proc[j + 1].id = temp;
39                }
40            }
41        }
42
43    }
44
45 }

```

Console Output:

```

<terminated> Ring (?) [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java [28]
Enter the number of process :
3
Enter the id of process :
5 6 8
Enter the id of process :
0
Enter the id of process :
1
[0] 5: [1] 6 [2] 8
process 8select as co-ordinator

1.election 2.quit
1
Enter the Process number who initialised election :
2
Process 8 send message to 5
Process 5 send message to 6
Process 6 send message to 8
process 8select as co-ordinator
1.election 2.quit
2
Program terminated ...

```

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.
- The output is associated in the above section.

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 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.

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 and exposes 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 Service Provider. It makes an invocation to the web service over the network to the Service Provider and exchanges 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 request content as a SOAP message.
2. Then, the client program sends this SOAP message to the server web service as an HTTP POST 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 and converting 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 output as the response to the SOAP request.
5. Then, the web service takes this response content as a SOAP message and reverts to the SOAP HTTP request invoked by the client program with this response.
6. The client program web service reads the SOAP response message to receive the outcome of the server program for the request content it sent as a request.

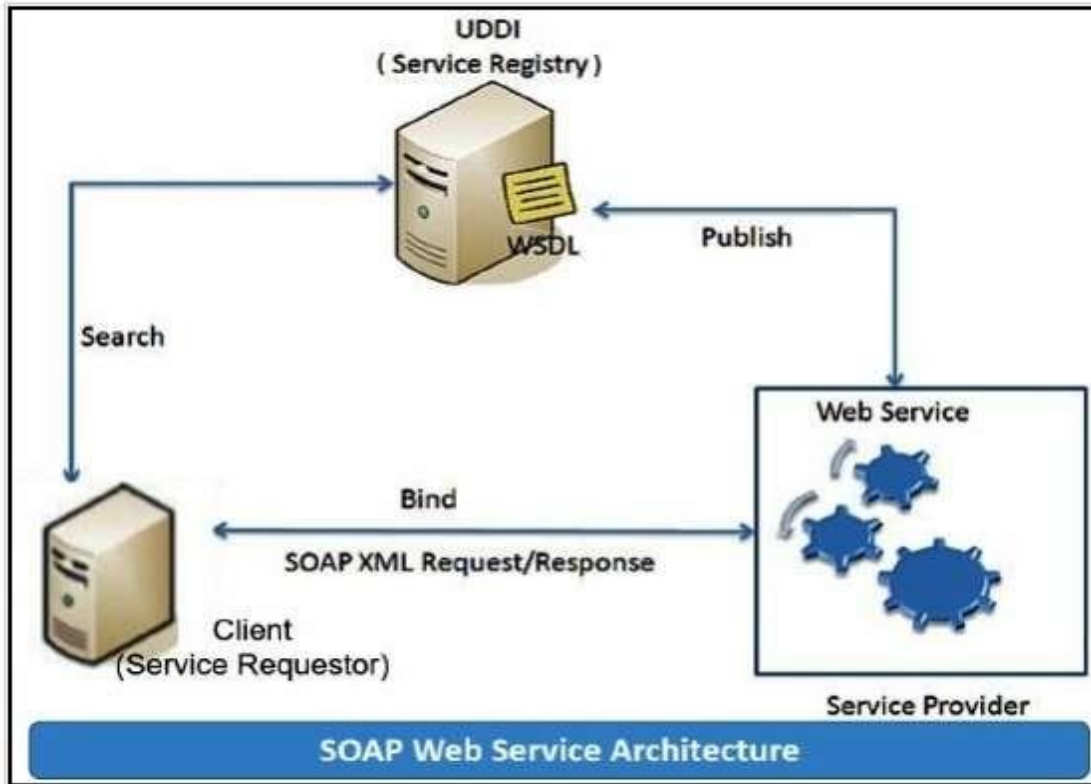
SOAP web services:

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 platform- and language-independent technology in integrated distributed applications.

While XML and HTTP together make the basic platform for web services, the following are the key components of standard SOAP web services:

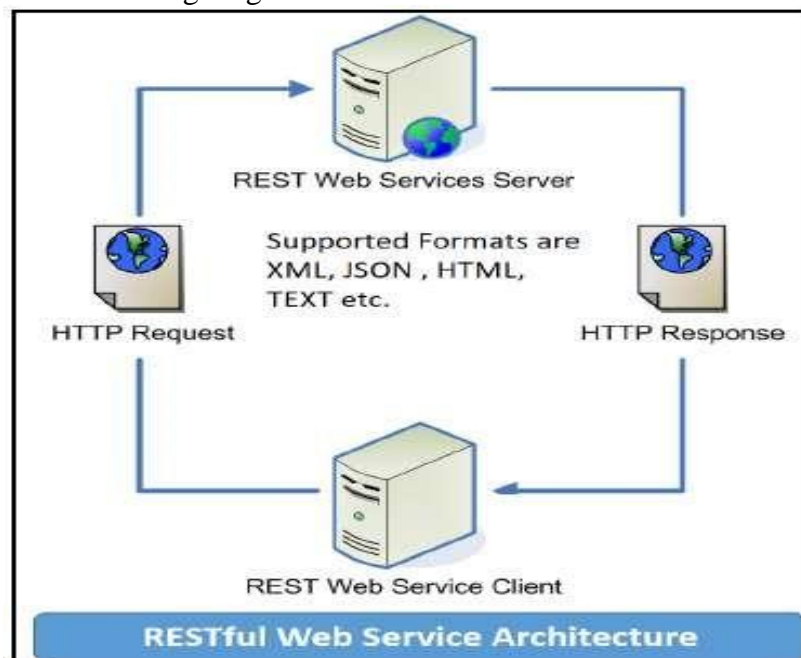
Universal Description, Discovery, and Integration (UDDI): UDDI is an XML-based framework for describing, discovering, and integrating web services. It acts as a directory of web service interfaces described in the WSDL language.

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



RESTful web services

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



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

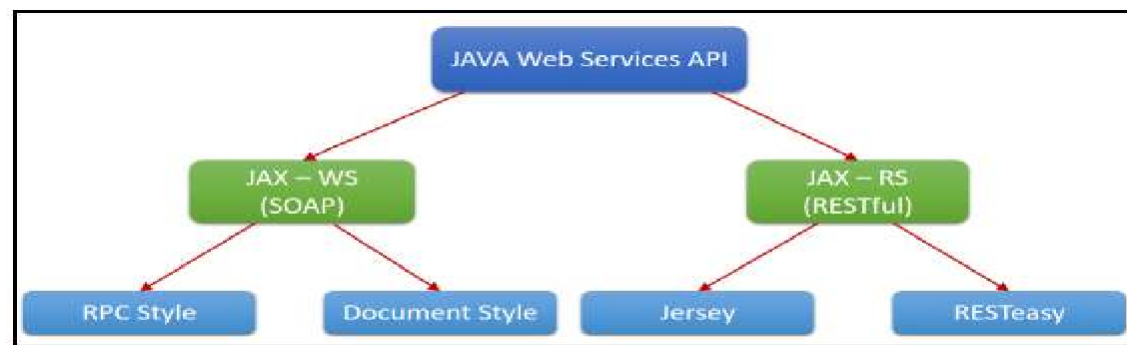
SOAP	REST
SOAP is a protocol.	REST is an architectural style.
SOAP stands for Simple Object Access Protocol.	REST stands for REpresentational State Transfer.
SOAP can't use REST because it is a protocol.	REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.
SOAP uses services interfaces to expose the business logic.	REST uses URI to expose business logic.
JAX-WS is the java API for SOAP web services.	JAX-RS is the java API for RESTful web services.
SOAP defines standards to be strictly followed.	REST does not define too much standards like SOAP.
SOAP requires more bandwidth and resource than REST.	REST requires less bandwidth and resource than SOAP.
SOAP defines its own security.	RESTful web services inherits security measures from the underlying transport.
SOAP permits XML data format only.	REST permits different data format such as Plain text, HTML, XML, JSON etc.
SOAP is less preferred than REST.	REST more preferred than SOAP.

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 based Java 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 creating REST web services. JAX-RS uses annotations to simplify the development and deployment of web services.

Both of these APIs are part of standard JDK installation, so we don't need to add any jars to work with them.



Students are required to implement both i.e. using SOAP and RESTful APIs.

Implementing the solution:**1. Creating a web service CalculatorWSApplication:**

- Create New Project for CalculatorWSApplication.
- Create a package `org.calculator`
- Create class `CalculatorWS`.
- Right-click on the `CalculatorWS` and create New Web Service.
- IDE starts the glassfish server, builds the application and deploys the application on server.

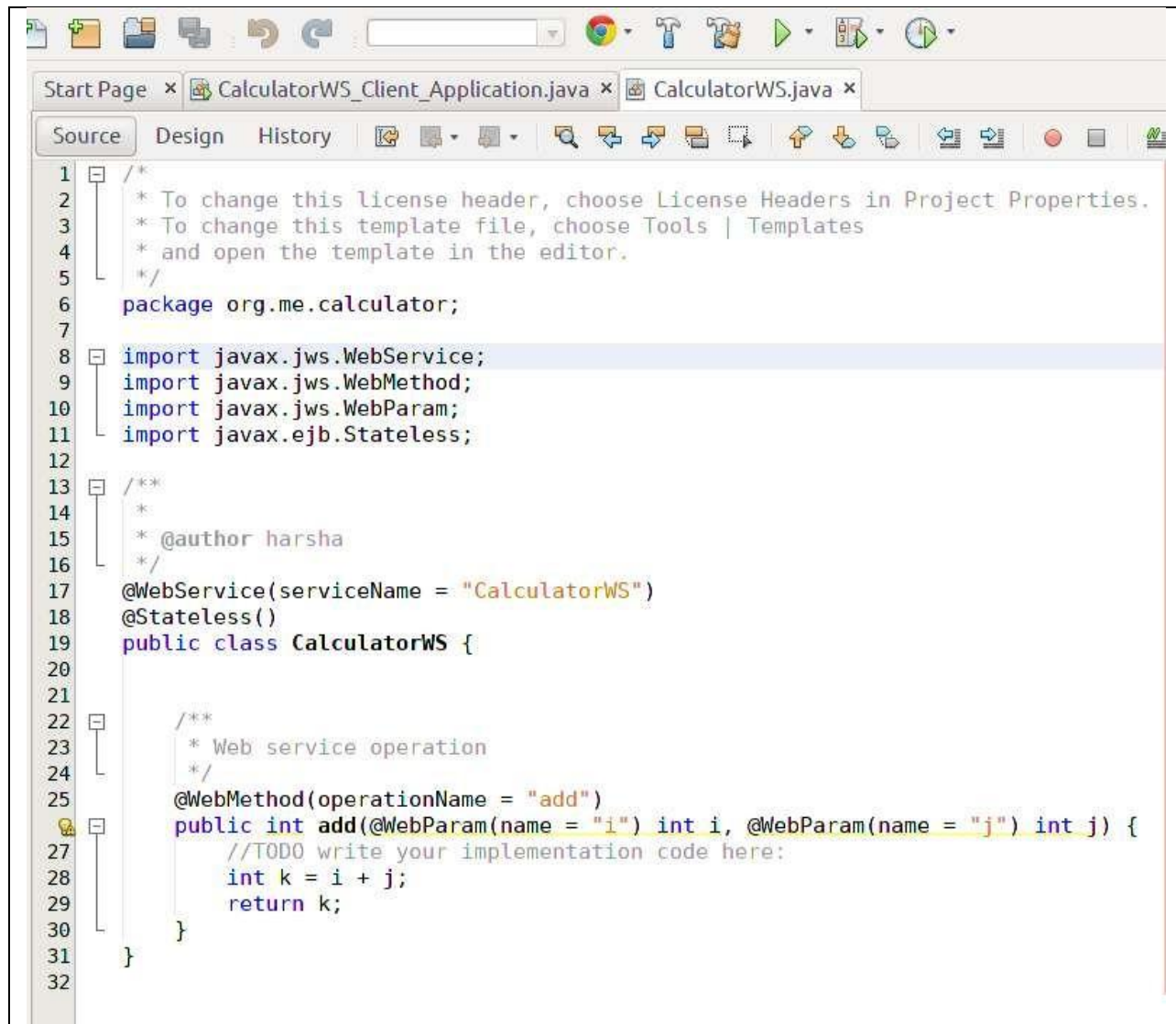
2. Consuming the Webservice:

- Create a project with an `CalculatorClient`
- Create package `org.calculator.client`;
- add java class `CalculatorWS.java`, `addresponse.java`, `add.java`, `CalculatorWSService.java` and `ObjectFactory.java`

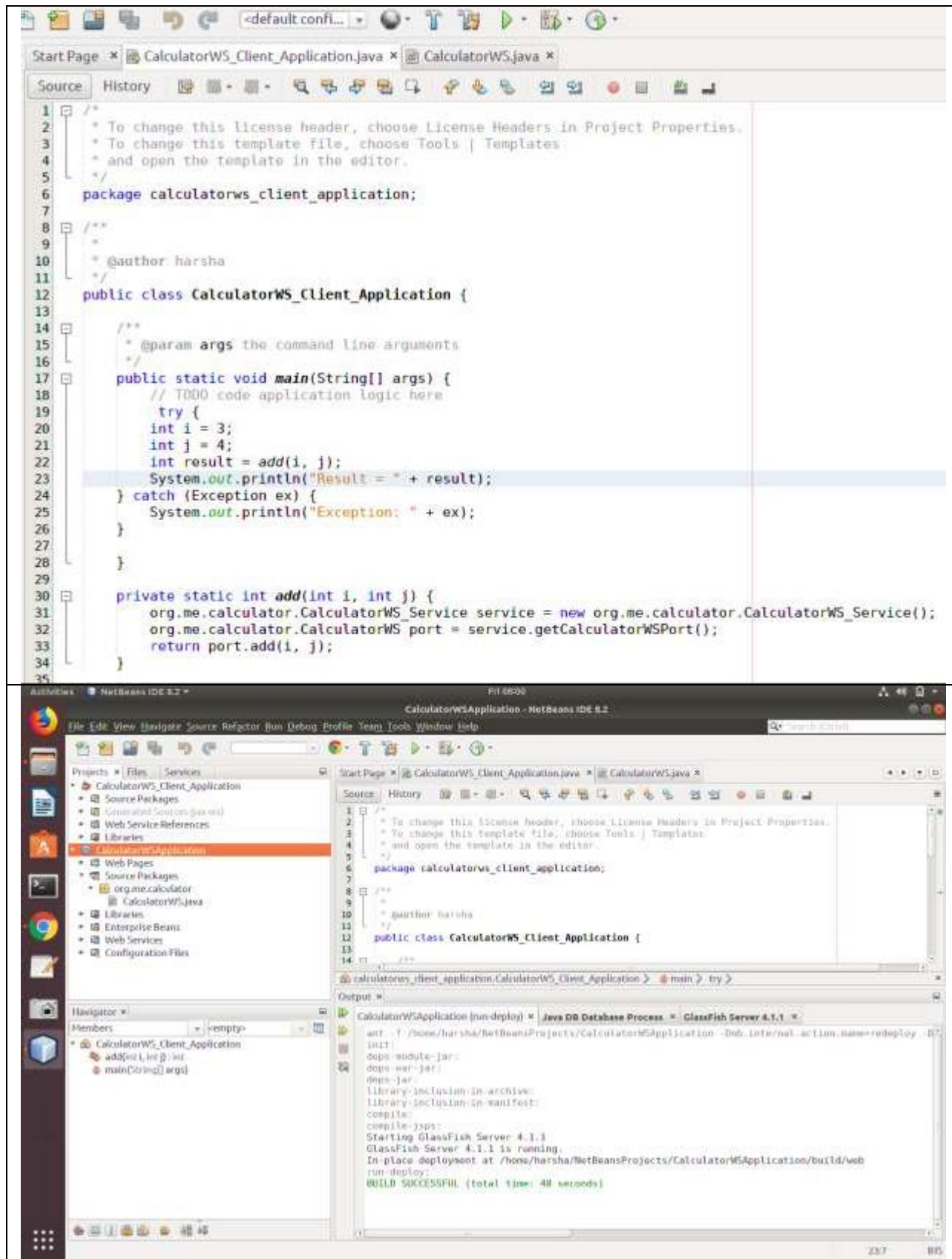
3. Creating servlet in web application

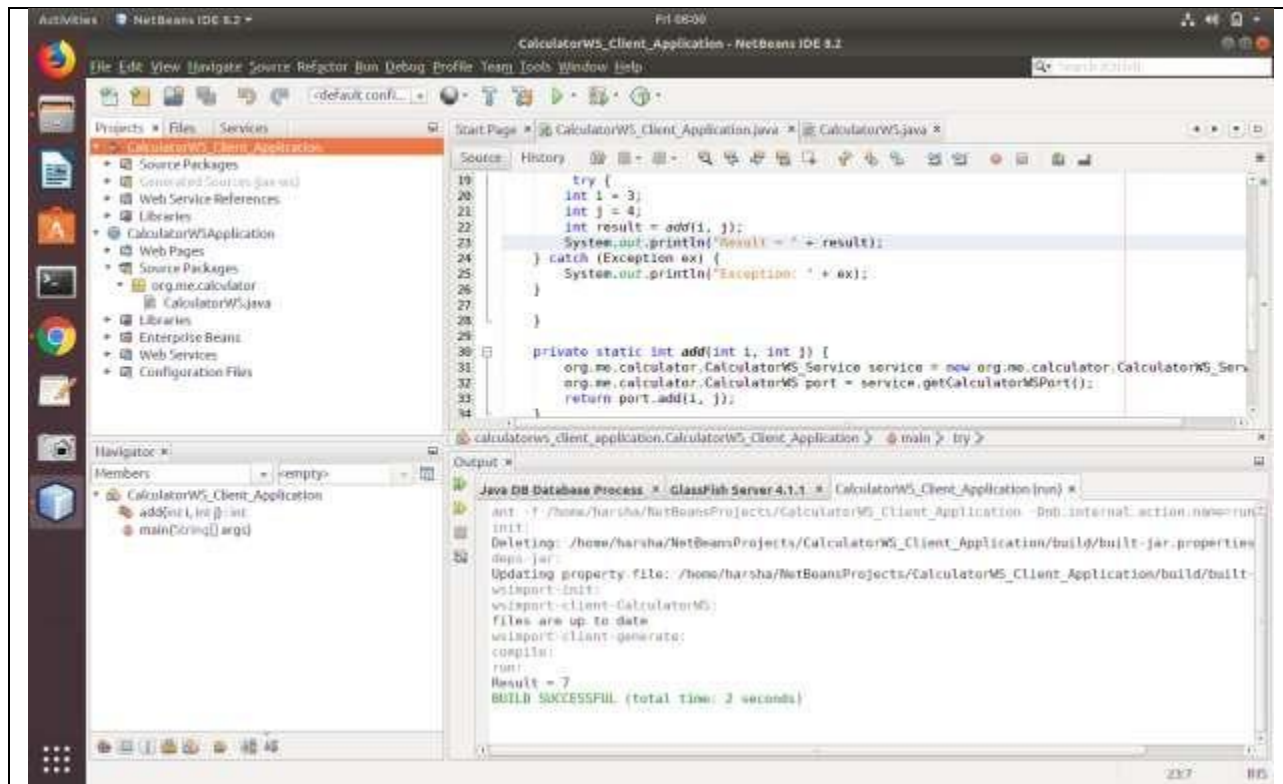
- Create new jsp page for creating user interface.

Writing the source code:



```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6   package org.me.calculator;
7
8   import javax.jws.WebService;
9   import javax.jws.WebMethod;
10  import javax.jws.WebParam;
11  import javax.ejb.Stateless;
12
13  /**
14   *
15   * @author harsha
16   */
17  @WebService(serviceName = "CalculatorWS")
18  @Stateless()
19  public class CalculatorWS {
20
21
22      /**
23       * Web service operation
24       */
25      @WebMethod(operationName = "add")
26      public int add(@WebParam(name = "i") int i, @WebParam(name = "j") int j) {
27          //TODO write your implementation code here:
28          int k = i + j;
29          return k;
30      }
31  }
32
```

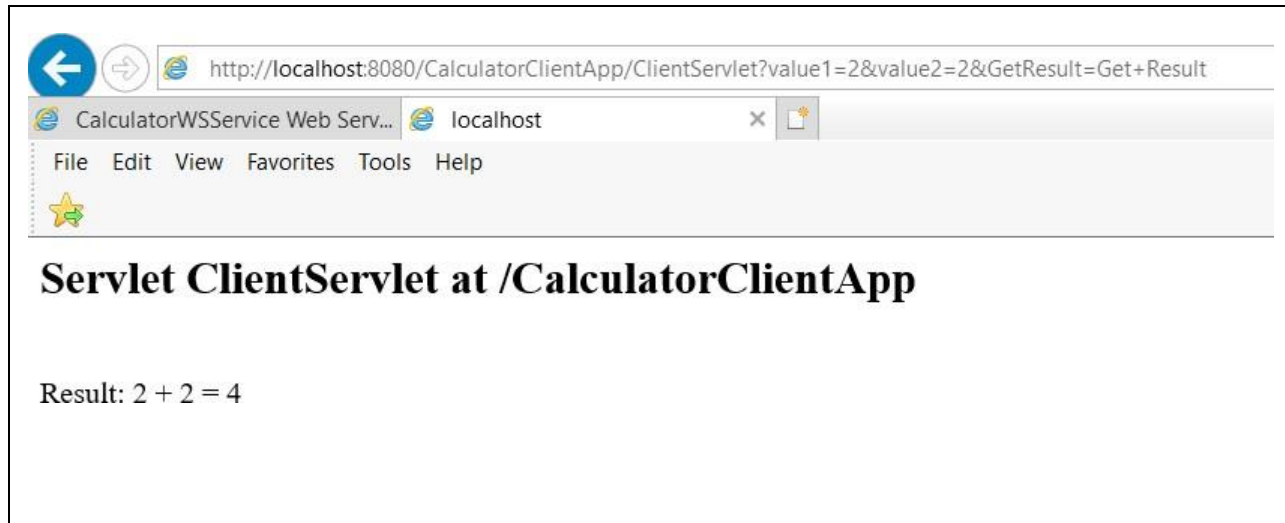




Compiling and Executing the solution:

Right Click on the Project and Choose Run.



**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.

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

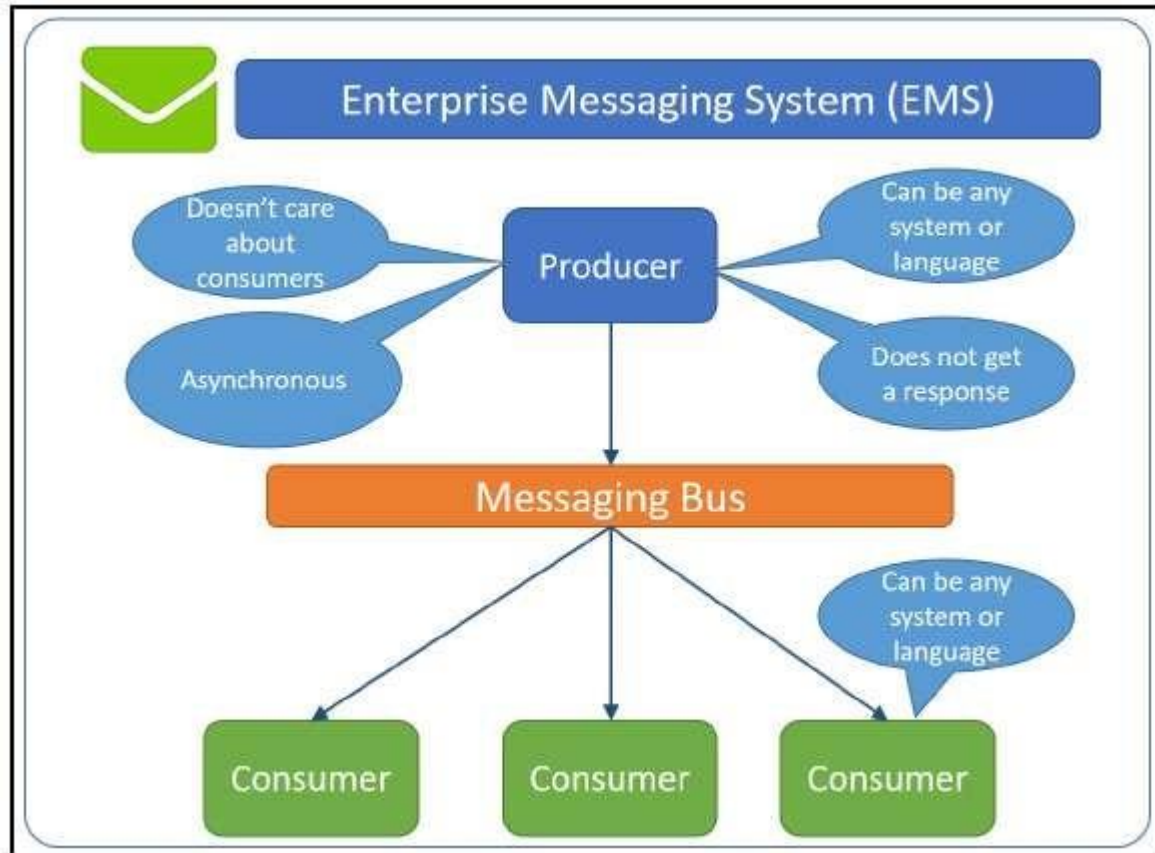
Related Theory:

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

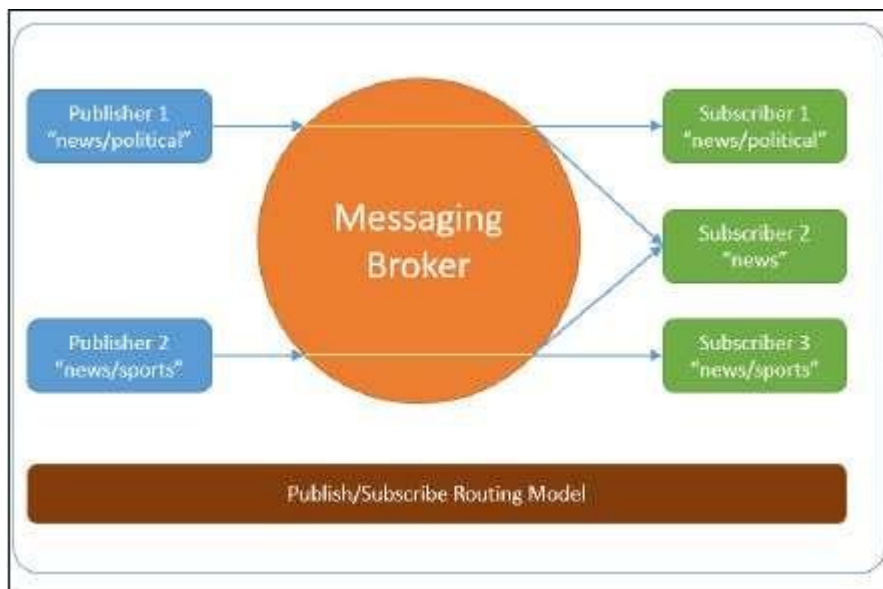
Java's implementation of an EMS in the Application Programming Interface (API) format is known as JMS.

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:



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:

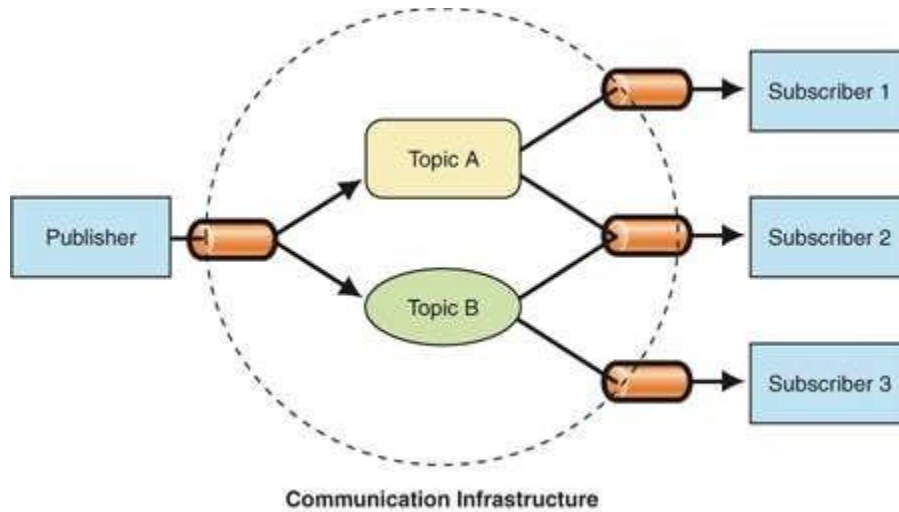
High Level Interface	Publish Subscribe model Interface
ConnectionFactory	TopicConnectionFactory
Connection	TopicConnection
Destination	Topic
Session	TopicSession
MessageProducer	TopicPublisher
MessageConsumer	TopicSubscriber

Designing the solution:

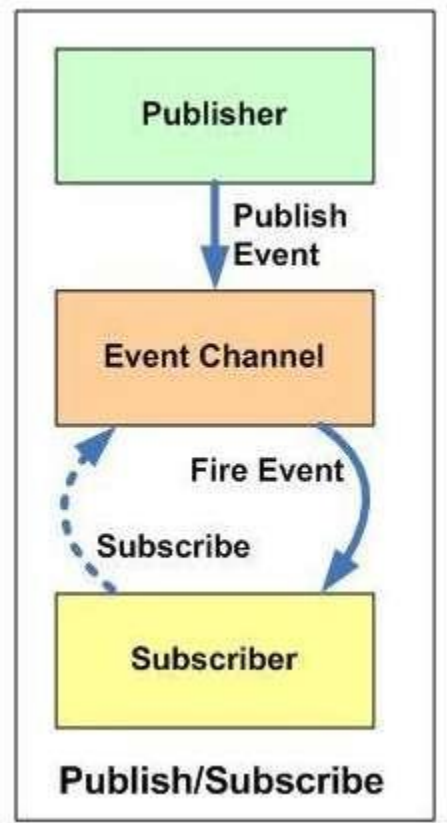
In ‘Publisher-Subscriber’ pattern, senders of messages, called **publishers**, do not program the messages to be sent directly to specific receivers, called **subscribers**.

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).

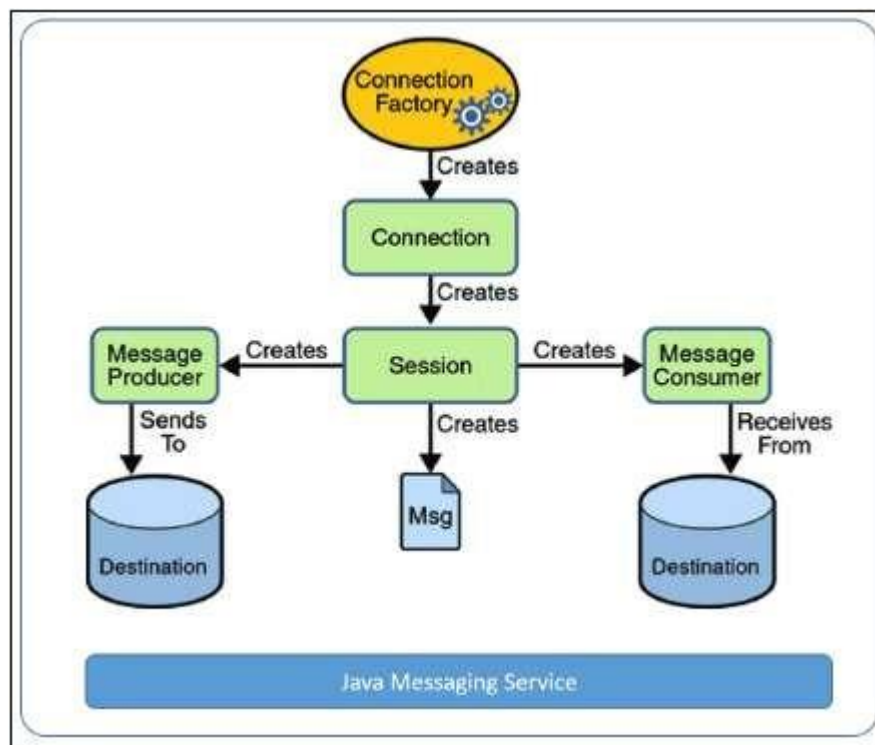


Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

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:



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 applications as 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.

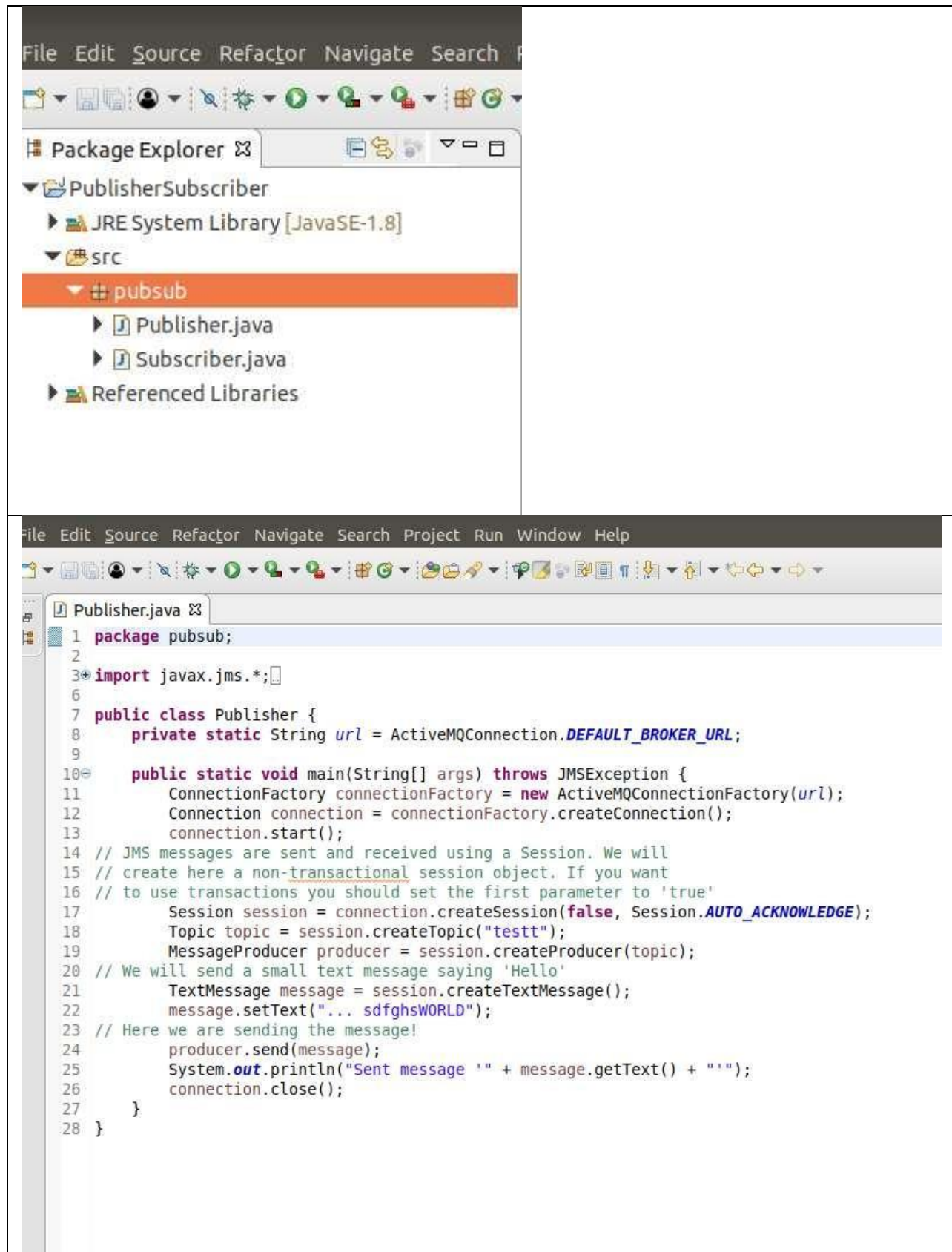
To use JMS, we need to have a JMS provider that can manage the sessions, queues, and topics. Some examples of known JMS providers are Apache ActiveMQ, WebSphere MQ from IBM or SonicMQ from Aurea Software. Starting from Java EE version 1.4, a JMS provider has to be contained in all Java EE application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

Writing the source code:



```

Subscriber.java
1 package pubsub;
2
3 import java.io.IOException;
4
5 public class Subscriber {
6     // URL of the JMS server
7     private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;
8
9     // Name of the topic from which we will receive messages from = " testt"
10    public static void main(String[] args) throws JMSEException {
11        // Getting JMS connection from the server
12        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
13        Connection connection = connectionFactory.createConnection();
14        connection.start();
15        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
16        Topic topic = session.createTopic("testt");
17        MessageConsumer consumer = session.createConsumer(topic);
18        MessageListener listner = new MessageListener() {
19            public void onMessage(Message message) {
20                try {
21                    if (message instanceof TextMessage) {
22                        TextMessage textMessage = (TextMessage) message;
23                        System.out.println("Received message" + textMessage.getText() + "");
24                    }
25                } catch (JMSEException e) {
26                    System.out.println("Caught:" + e);
27                    e.printStackTrace();
28                }
29            }
30        };
31        consumer.setMessageListener(listner);
32        try {
33            System.in.read();
34        } catch (IOException e) {
35            e.printStackTrace();
36        }
37        connection.close();
38    }
39 }
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612

```

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 Installation Link

Links for Download and installation instruction:

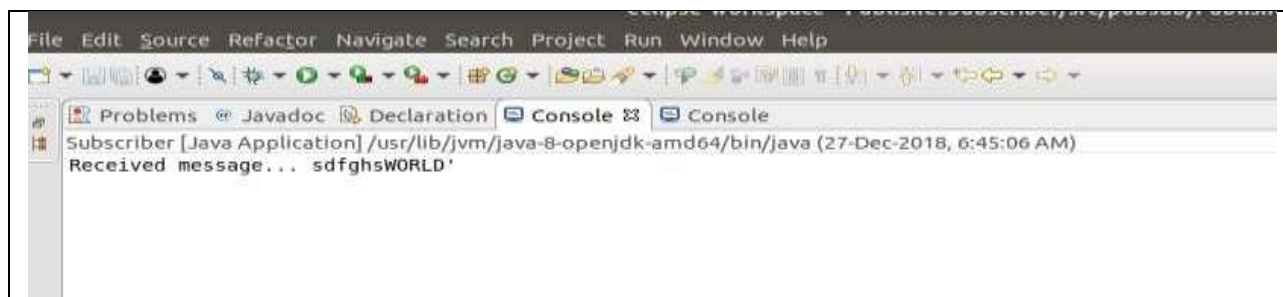
- a. Jms <http://www.java2s.com/Code/Jar/j/Downloadjavaxjmsjar.htm> [Jar file]
- b. Apache <http://www.java2s.com/Code/Jar/a/Downloadapacheactivemq411jar.htm> [Jar file]
- c. Download - <http://activemq.apache.org/activemq-5158-release.html>.....[ApacheMQ Download link]
- d. Install - <https://docs.wso2.com/display/BAM200/Installing+Apache+ActiveMQ+on+Linux> [Apache MQ Installation Instructions]
- e. Concept - <https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c>

Steps to execute:

1. Create a Publisher.java file and copy paste the Publisher code
2. Create a Subscriber.java file and copy paste the Subscriber code
5. Add external jars
 - 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
6. Run activemq with the following command:

```
sudo sh active start
```

7. Run the publisher code and pin console for publisher
8. Run Subscriber



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.

ASSIGNMENT NO. 7**Problem Statement:**

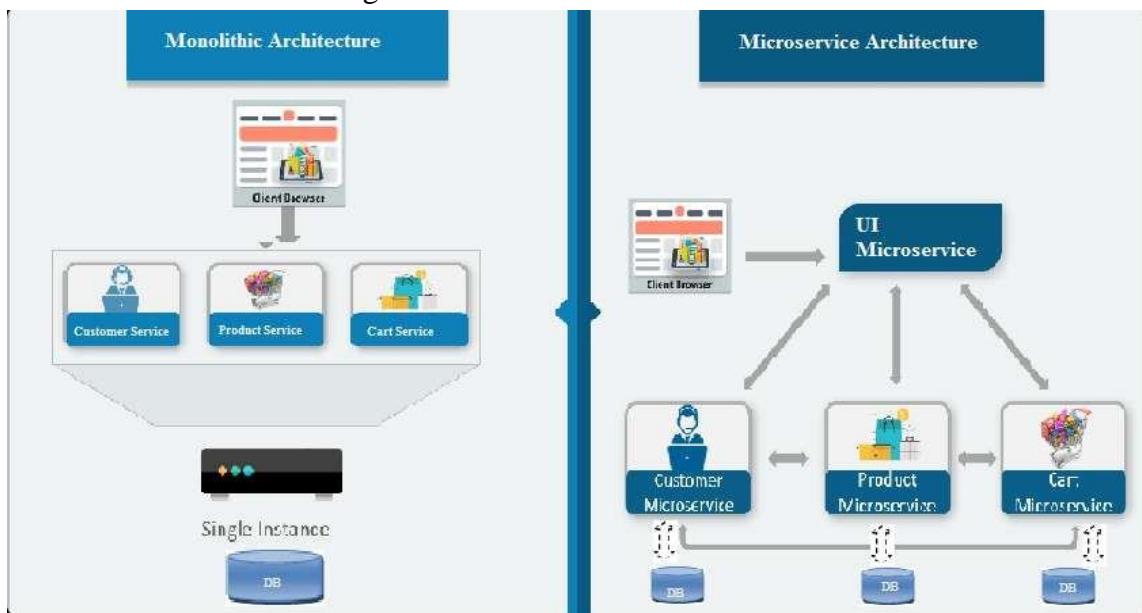
To develop microservices framework based distributed application.

Tools / Environment:

Python 3.6.0 using Flask framework.

Related Theory:**1. Microservices:**

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 timeframes. In contrast with the monolith type application, here’s what an app developed with a microservices focus might look like:



A team designing a microservices 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.

Microservices - also known as the microservice 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 microservice 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 Python web 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.

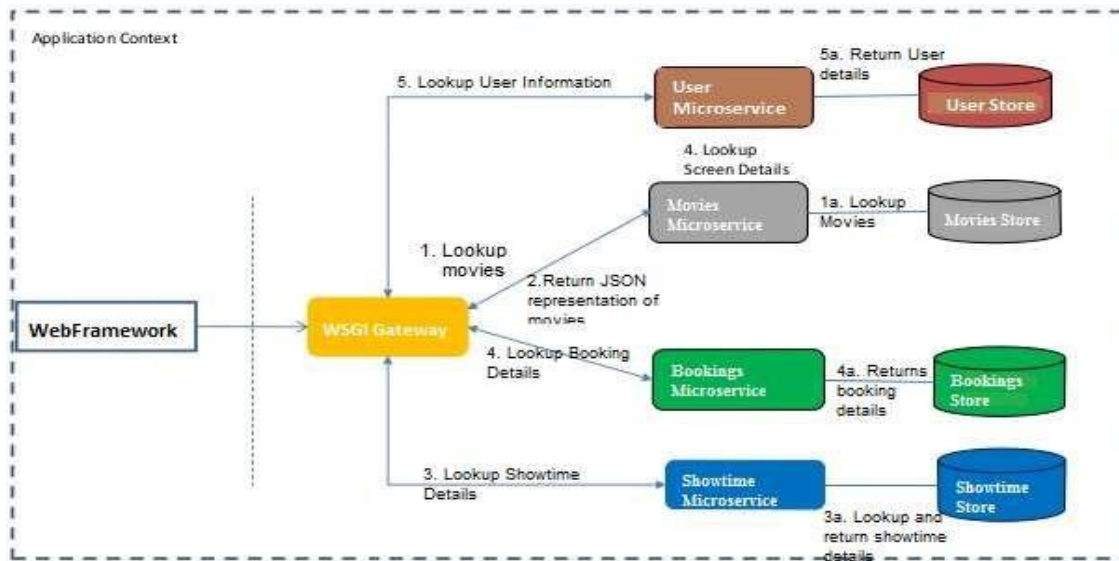
6. Virtual Environment:

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: *ProjectA* and *ProjectB*, both of which have a dependency on the same library, *ProjectC*. The problem becomes apparent when we start requiring different versions of *ProjectC*. Maybe *ProjectA* needs v1.0.0, while *ProjectB* 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, *ProjectA* and *ProjectB*, 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 `virtualenv` or `pyenv` command line tools.

Designing the solution:



Here, we are attempting to develop an microservice 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 `virtualenv` 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 (`__name__`) 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:

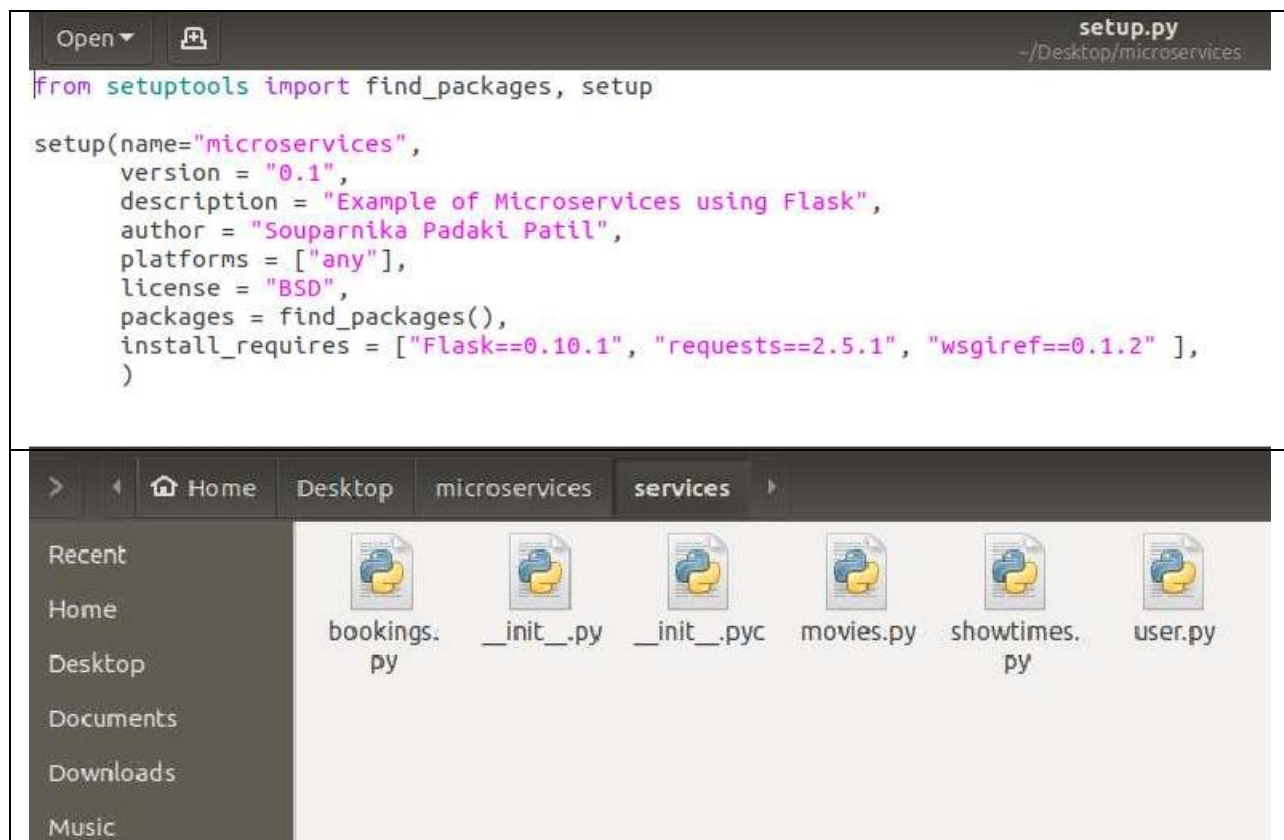
The route() decorator in Flask is used to bind URL to a function.

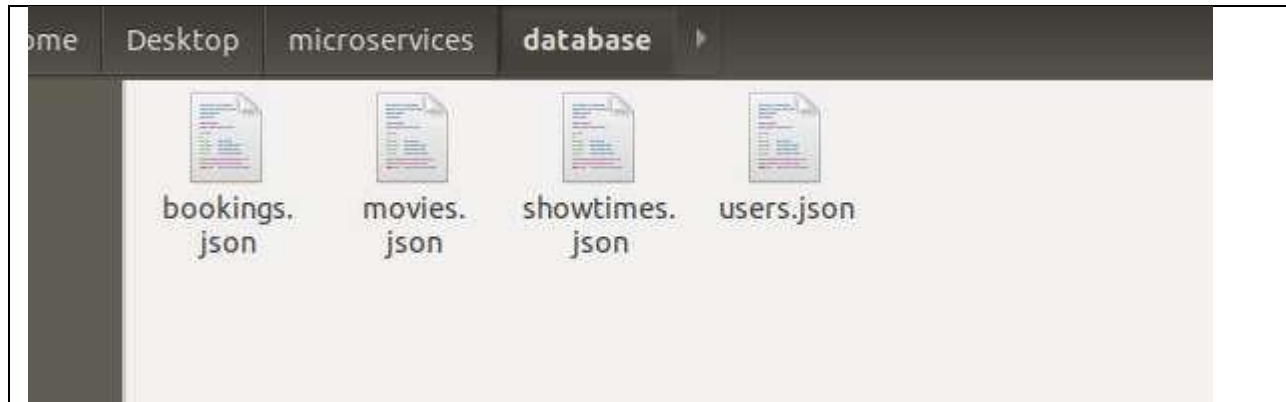
For example –


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

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

3. **Writing the subroutine for the four microservices:** There are four microservices viz., user, Showtimes, Bookings and Movies for which microservices are to be implemented.

Writing the source code:




```
Open ▾  movies.json  
~/Desktop/microservices/database  
{  
  "720d006c-3a57-4b6a-b18f-9b713b073f3c": {  
    "title": "Happy Phirr Bhag Jayegi",  
    "rating": 7.4,  
    "director": "Mudassar Aziz",  
    "id": "720d006c-3a57-4b6a-b18f-9b713b073f3c"  
  },  
  "a8034f44-ae4-44cf-b32c-74cf452aaaae": {  
    "title": "Stree",  
    "rating": 9.2,  
    "director": "Amar Kaushik",  
    "id": "a8034f44-ae4-44cf-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 microservices run :

```
./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-ae4-44cf-b32c-74cf452aaaae
Title: Stree
Director: Amar Kaushik
Rating: 9.2

ID: 7daf7208-be4d-4944-a3ae-clc2f516f3e6
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-clc2f516f3e6 MOVIE: Mission Impossible 6
ID: 39ab85e3-5e8e-4dc5-afea-65dc368bd7ab MOVIE: The Incredibles 2
ID: a8034f44-ae4-44cf-b32c-74cf452aaaae MOVIE: Stree
On date: 20180803
ID: 720d006c-3a57-4b6a-b18f-9b713b073f3c MOVIE: Happy Phirr Bhag Jayegi
ID: 39ab85e3-5e8e-4dc5-afea-65dc368bd7ab MOVIE: The Incredibles 2
On date: 20180802
ID: a8034f44-ae4-44cf-b32c-74cf452aaaae MOVIE: Stree
ID: 96798c08-d19b-4986-a05d-7da856efb697 MOVIE: Gold
ID: 39ab85e3-5e8e-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-ae4-44cf-b32c-74cf452aaaae MOVIE: Stree
ID: 7daf7208-be4d-4944-a3ae-clc2f516f3e6 MOVIE: Mission Impossible 6

```

```

1.Get Movie list (no 04) Address already in use
2.Get Show Times (no 04) Address already in use
3.Get Bookings Info (no 04) Address already in use
4.Get User list (no 04) Address already in use
5.Book a show (no 04) Address already in use
6.Clearscreen (no 04) Address already in use
7.Exit (no 04) Address already in use
Select an option (no 04) debug=True
4
Anuja Kharatmol (no 04) port: 5050, (optional)
Souparnika Patil (no 04) python3.7.1/dist-packages/Vartraag-0.14.1-py3.7-egg/Var
Vasundhara Kurtakoti
Yojane Mane (no 04) python3.7.1/dist-packages/Vartraag-0.14.1-py3.7-egg/Var
Nachiket Ghorpade (no 04) python3.7.1/dist-packages/Vartraag-0.14.1-py3.7-egg/Var
Nayana Patil (no 04) python3.7.1/dist-packages/Vartraag-0.14.1-py3.7-egg/Var
Kamraj Ambalkar (no 04) Address already in use

```

```
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-ae44-44cf-b32c-74cf452aaaae
Title: Stree
Director: Amar Kaushik
Rating: 9.2
```

```
ID: 7daf7208-be4d-4944-a3ae-clc2f516f3e6
```

```
ID: 39ab85e5-5e8e-4dc5-afea-65dc368bd7ab
Title: The Incredibles 2
Director: Brad Bird
Rating: 7.1
```

```
ID: 276c79ec-a26a-40a6-b3d3-fb242a5947b6
Title: Avengers Infinity War
Director: Anthony Russo
Rating: 9.8
```

```
>Enter the id for the booking : 276c79ec-a26a-40a6-b3d3-fb242a5947b6
```

```
Booking the show for the following movie on date 20180802
```

```
ID: 276c79ec-a26a-40a6-b3d3-fb242a5947b6
Title: Avengers Infinity War
Director: Anthony Russo
Rating: 9.8
```

```
Press enter to continue
```

```
BOOKING DONE!! Thank you for using Cinema app
```

Conclusion:

With microservices, modules within software can be independently deployable. In a microservices architecture, each service runs a unique process and usually manages its own database. This not only provides development teams with a more decentralized approach to building software, it also allows each service to be deployed, rebuilt, 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 microservices architecture.

References:**Assignment 1 :**

jdk - <https://www.oracle.com/technetwork/java/javase/downloads/index.html>

Eclipse IDE - <https://www.eclipse.org/downloads>

Java socket programming :

<https://www.careerbless.com/samplecodes/java/beginners/socket/SocketBasic1.php>

Assignment 2 :

MPJ Reference link : <http://mpj-express.org/docs/guides/linuxguide.pdf>

Assignment 3 :

Java IDL: The "Hello World" Example :

<https://docs.oracle.com/javase/7/docs/technotes/guides/idl/jidlExample.html>

Assignment 4 :

jdk - <https://www.oracle.com/technetwork/java/javase/downloads/index.html>

Eclipse IDE - <https://www.eclipse.org/downloads>

Ring : <http://mymindcmind.blogspot.com/p/ring-and-bully-algorithm.html>

Bully : https://github.com/prasadgujar/Source_Code?files=1

Assignment 5 :

Netbeans IDE with glassfish server : <https://netbeans.org/downloads/>

Getting Started with JAX-WS Web Services : <https://netbeans.org/kb/docs/websvc/jax-ws.html>

Assignment 6 :

JMS <http://www.java2s.com/Code/Jar/j/Downloadjavaxjmsjar.htm> [Jar file]

Apache <http://www.java2s.com/Code/Jar/a/Downloadapacheactivemq411jar.htm> [Jar file]

Download - <http://activemq.apache.org/activemq-5158-release.html>

[ApacheMQ Download link]Install -

<https://docs.wso2.com/display/BAM200/Installing+Apache+ActiveMQ+on+Linux>

[Apache MQ Installation Instructions]Concept - <https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c>

Assignment 7 :

Building Microservices with Python:-

<https://medium.com/@ssola/building-microservices-with-python-part-i-5240a8dcc2fb>.

Python Virtual Environments: A Primer:-

<https://realpython.com/python-virtual-environments-a-primer/>

.....End.....