JSPM's
BHIVARABAI SAWANT INSTITUTE OF TECHNOLOGY & RESEARCH, Wagholi
(Approved by AICTE, Delhi, Govt. of Maharashtra & Affiliated to University of Pune)
Gat No. 720(1), Wagholi, Pune-Nagar Road, Pune-412207
Tel. No. (020)27051170 Fax No. (020)27052590

DEPARTMENT OF INFORMATION TECHNOLOGY

LAB MANUAL

Lab Practice - V

CLASS- BE

(2019 PATTERN)

Vision of Institute

" To Satisfy the aspirations of youth force, who wants to lead nation towards prosperity through techno-economic developments "

Mission of Institute

" To provide, nurture and maintain an environment of high academic excellence, research and entrepreneurship for all aspiring students which will prepare them to face global challenges, maintaining high ethical and moral standard "

Vision statement of the Department:

To be a nucleus for nurturing learner to cater current and future digital needs.

Mission statement of the Department:

M1: To groom learners for addressing technical challenges by utilizing knowledge and skill sets.

M2: To inculcate professional values to develop effective and efficient organization through best practices.

Course Objectives:

1. 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.

2. The course covers the building blocks for a study related to the design and the implementation of distributed systems and applications.

Course Outcomes:

Upon successful completion of this course student will be able to:

1. Demonstrate knowledge of the core concepts and techniques in distributed systems.

2. Learn how to apply principles of state-of-the-Art Distributed systems in practical application.

3. Design, build and test application programs on distributed systems

List of Laboratory Assignments

PRACTICAL NO.	NAME OF PRACTICAL
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

ASSIGNMENT NO. 1

Problem Statement:

Implement multi-threaded client/server Process communication using RMI.

Tools / Environment:

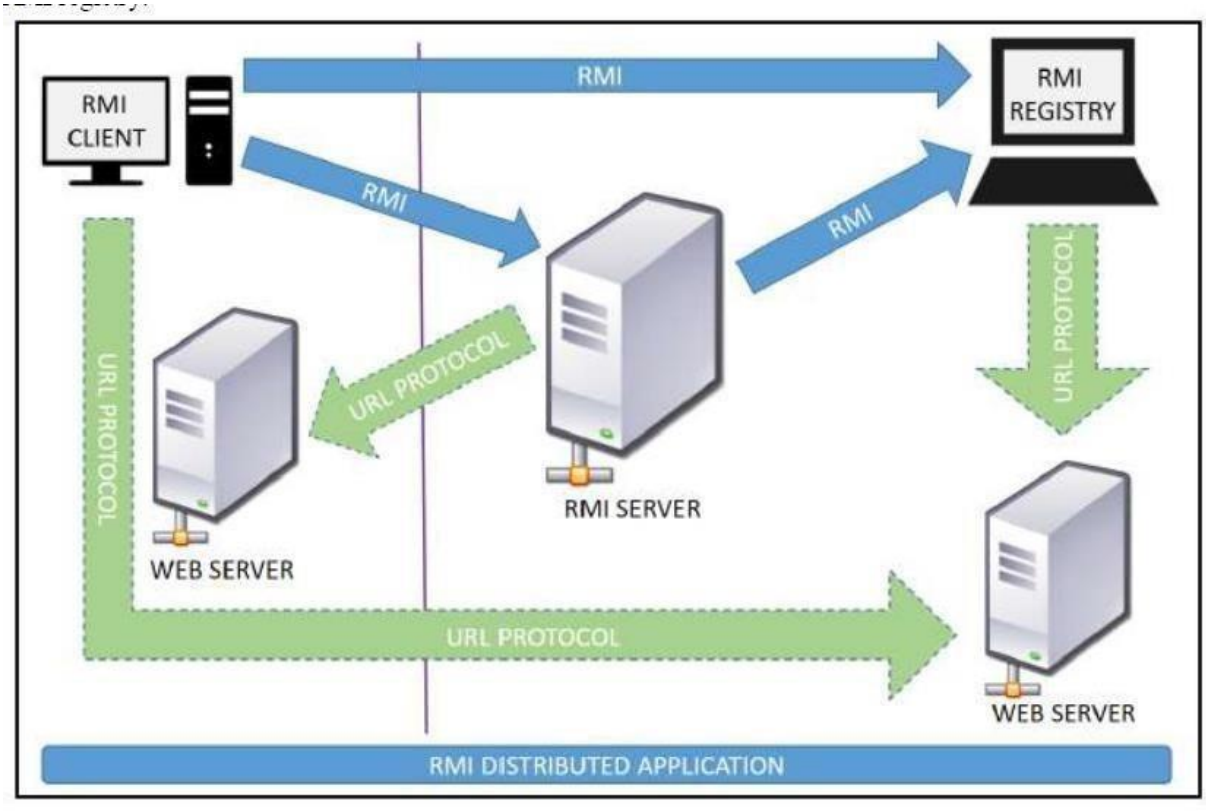
Java Programming Environment, jdk 1.8, rmiregistry

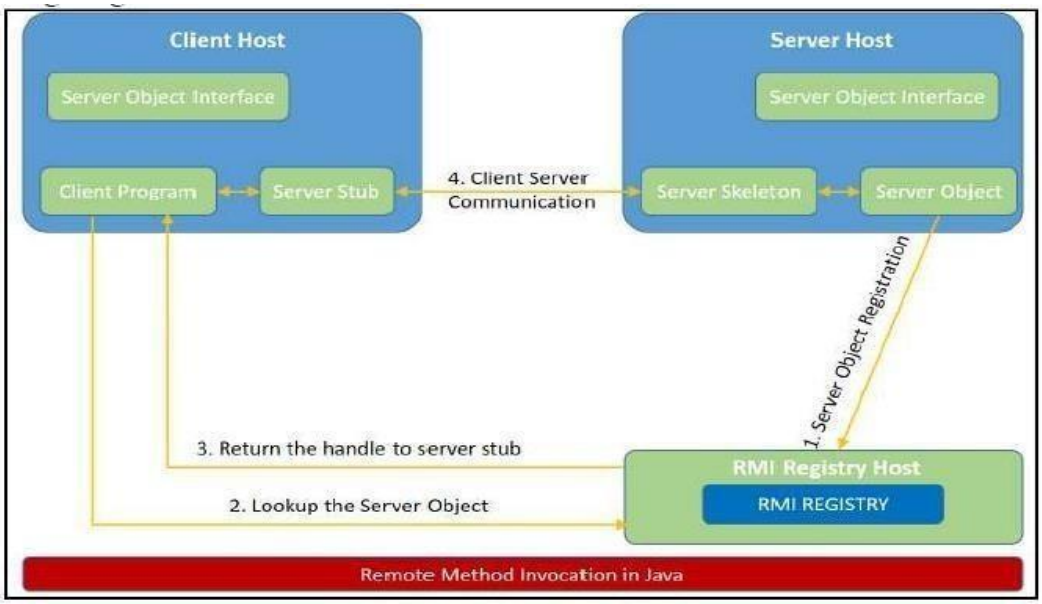
Related Theory:

Introduction:

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 ofan object through its methods.

The following diagram shows how RMI happens between the RMI client and RMI server with the helpof 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 orexception, 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 thesesteps:

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 asfollows:

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

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

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

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

Implementing the solution:

Consider building an application to perform diverse mathematical operations.

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

1. Creating remote interface, implement remote interface, server-side and client-side programand 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 primaryfunction is **to update the RMI registry on that machine**. This is done by using the **rebind()** methodof the **Naming** class (found in **java.rmi**). That method associates a name

with an object reference. Thefirst 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 returnedto the client, the process works in reverse. **The serialization and deserialization facilities are also used if objects are returned to a client.**

To generate a stub the command is RMIcompiler is invoked as follows:

rmic AddServerImpl.

This command generates the file **AddServerImpl_Stub.class**.

3. Install Files on the Client and Server Machines

Copy **AddClient.class**, **AddServerImpl_Stub.class**, **AddServerIntf.class** to a directory on the clientmachine.

Copy AddServerIntf.class, AddServerImpl.class, AddServerImpl_Stub.class, and **AddServer.class** to a directory on the server machine.

4. Start the RMI Registry on the Server Machine

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

start rmiregistry

5. Start the Server

The server code is started from the command line: java AddServer

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

6. Start the Client

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

Compilation and Executing the solution:

1. Create all java files and compile using **javac** command , it will generate **.class** files.
2. Generate stubs invoking **rmic AddServerImpl** it will generate **AddServerImpl_Stub.class**

   file.
3. Copy **AddClient.class**, **AddServerImpl_Stub.class**, and

   AddServerIntf.class to a directoryon the client machine/folder.
4. Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_Stub.class**, and **AddServer.class** to a directory on the server machine/folder.
5. Start the RMI Registry on the Server Machine using **rmiregistry**
6. In new terminal start the Server using **java AddServer**
7. In another new terminal start the Client **java AddClient**

   servername/ip_address 8 9 where servername is first argument and 8 ,

   9 are second & third arguments respectively.

   e.g **java AddClient 127.0.0.1 8 9** for localhost (when client and server on same machine)

   e.g **java AddClient 172.16.86.80 8 9** (when client and server on different

   machine, specify IPaddress of server machine)

Conclusion:

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

ASSIGNMENT NO. 2

Problem Statement:

Develop any distributed application using CORBA to demonstrate object brokering. (Calculator or String operations).

Tools / Environment:

Java Programming Environment, jdk 1.8, rmiregistry

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 arequite 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 throughthe 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

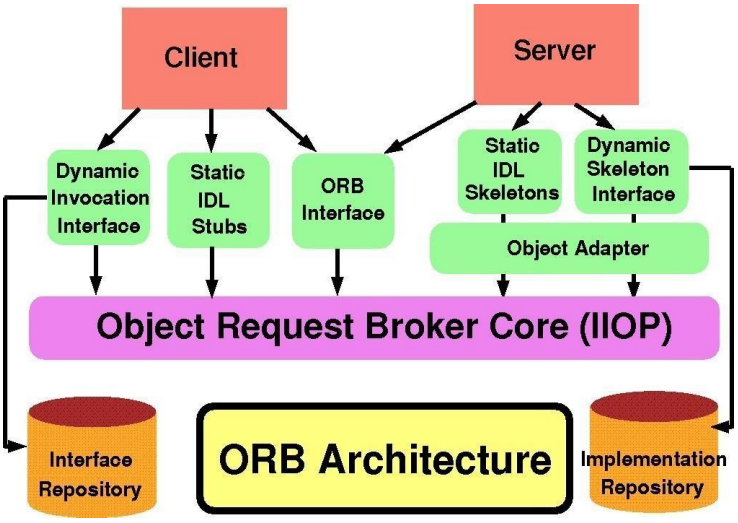
& Benefits" maintain an object model with details of the organization, employees with demographic information, benefits, payroll, and deductions. They are only different in the way they handle the details, based on the country and region they are operating for. For each object type, similar to the HR & 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 requestedby the client operation. Again, during response handling, the interface definition is helpful to marshal (send from the server) and unmarshal (receive and read the response) arguments on the client side once received.

The IDL interface is a design concept that works with multiple programming languages including C, C++, Java, Ruby, Python, and IDL script. This is close to writing a program to aninterface, 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.

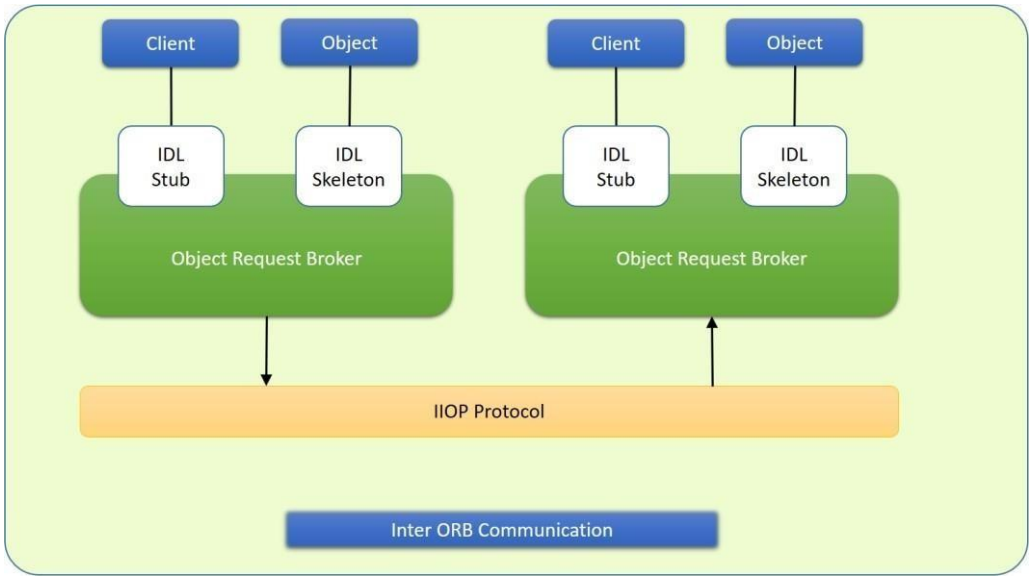
The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons, The objects are written (on the right) and a client for it (on the left), as represented in the diagram. The client and server use stubs and skeletons as proxies, respectively. 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 helpestablish this communication through ORB and pass the arguments to the right method and itsinstance 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 thatpower 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 limitedto 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 theORB.

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 differentORB 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, whichuses a POA server-side model.

1. Creating CORBA Objects using Java IDL:

In order to distribute a Java object over the network using CORBA, one has to define its own CORBA-enabled interface and it implementation. This involves doingthe 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-Javacompiler
- Writing a server-side implementation of the Java interface in JavaInterfaces 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 thismodule (interfaces, constants, other modules) falls within the module and is referenced in otherIDL 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 headerspecifies 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 aminimum.

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

This declares a method called parseString() that accepts a single string argumentand returns a string value.

A complete IDL example

1. Here's a complete IDL example that declares a module within another module, which itselfcontains 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 objectand 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 Javaimplementation of the interface (but you have to write it, since IDL doesn't provide any detailsabout 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 updatethe argument it is passed and return the updated value. The only way to guarantee this, even for primitive Java data types, is to force out and inout arguments to be wrapped in Java holder classes, which are filled with the output value of the argument when the method returns.

The idltoj tool generate 2 other classes:

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

- **A server *skeleton* class,** called *_interface-name*ImplBase, that is a base class for a serverside implementation of the interface. The base class can accept

requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named Server is called _ServerImplBase.

So, in addition to generating a Java mapping of the IDL interface and some helper classes for the Java interface, the *idltoj* compiler also creates subclasses that act as an interface between aCORBA 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 holderclass, 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 thisdirectory.**

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 omplete the application,you simply provide the server **(HelloServer.java)** and client **(HelloClient.java)** implementations.

2. Implementing the Server (HelloServer.java)

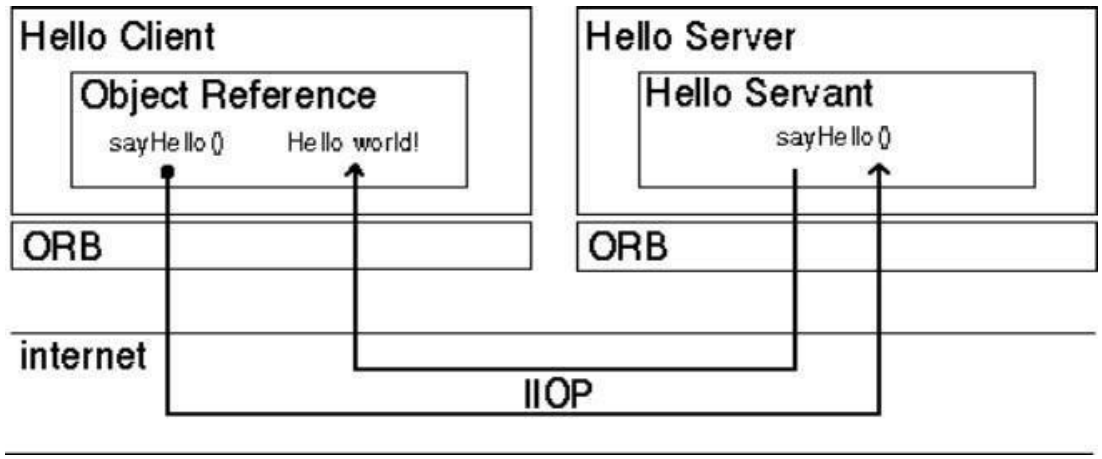
The example server consists of two classes, the servant and the server. The servant, HelloImpl, is the implementation of the Hello IDL interface; each Hello instance is implemented by a HelloImpl instance. The servant is a subclass of HelloPOA, which is generated by the idlj compiler from the example IDL. The servant contains one method foreach IDL operation, in this example, the sayHello() and shutdown() methods. Servantmethods 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) andtells the ORB about it
- Gets a CORBA object reference for a naming context in which to register the newCORBA 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.



4. 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 CORBAobject
- Invokes the object's sayHello() and shutdown() operations and prints the result.

Building and executing the solution:

The Hello World program lets you learn and experiment with all the tasks required to develop almost any CORBA program that uses static invocation, which uses a

client stub for the invocation and a server skeleton for the service being invoked and is used when the interface of the object is known at compile time.

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

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

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

```
idlj -fall Hello.idl
```

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

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

- HelloPOA.java:

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

- HelloStub.java:

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

- Hello.java:

This interface contains the Java version of IDL interface written. The Hello.java interface extends org.omg.CORBA.Object, providing standard CORBA object functionality. It alsoextends the Hell

- 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 typefrom AnyS.** The Holder class delegates to the methods in the Helper class for reading and writing.

- HelloHolder.java

This final class holds a public instance member of type Hello. Whenever the IDL type is an out or an inout parameter, the Holder class is used. It provides operations for org.omg.CORBA.portable.OutputStream and org.omg.CORBA.port able.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 andwriting. It implements org.omg.CORBA.portable.Streamable.

- HelloOperations.java

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

3. Compile the .java files, including the stubs and skeletons (which are in the directory directory HelloApp). This step assumes the java/bindirectory 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 –ORBInitialHostlocalhost&
```

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

6. Run the client application:

java HelloClient -ORBInitialPort 1050 –
ORBInitialHostlocalhost

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).

Compiling and Executing:

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

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

```
idlj -fall
ReverseModule.idl
```

The idlj compiler generates a number of files.

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

```
javac *.java  ReverseModule/*.java
```

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

```
orbd -ORBInitialPort 1050&
```

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

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

6. Run the client application :

```
java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

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. 3

Problem Statement:

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.

Tools / Environment:

Java Programming Environment, jdk 1.8, 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

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)

MPJ Express is an open source Java message passing library that allows application developers to write and execute parallel applications for multicore processors and compute clusters / clouds. The software is distributed under the MIT (a variant of the LGPL) license.
MPJ Express is a message passing library that can be used by application developers to execute their parallel Java applications on compute clusters or network of computers.
MPJ Express is essentially a middleware that supports communication between individual processors of clusters. The programming model followed by MPJ Express is Single Program Multiple Data (SPMD).

The multicore configuration is meant for users who plan to write and execute parallel Java applications using MPJ Express on their desktops or laptops which contains shared memory and multicore processors.

In this configuration, users can write their message passing parallel application using MPJ Express and it will be ported automatically on multicore processors. We except that users can first develop applications on their laptops and desktops using multicore configuration, and then take the same code to distributed memory platform

Conclusion: Thus we 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. Demonstrated by displaying the intermediate sums calculated at different processors.

ASSIGNMENT NO. 4

Problem Statement:

Implement Berkeley algorithm for clock synchronization.

Tools / Environment:

Java Programming Environment, jdk 1.8

Related Theory:

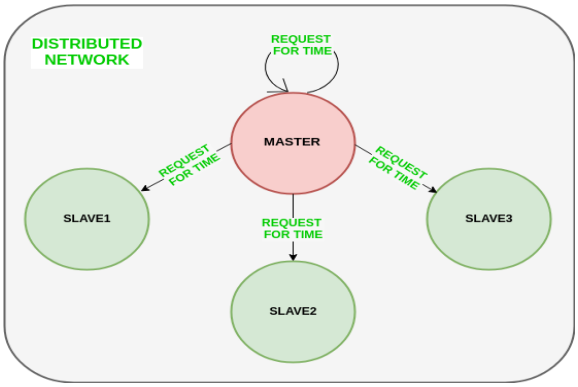
Berkeley's Algorithm is a clock synchronization technique used in distributed systems. The algorithm assumes that each machine node in the network either doesn't have an accurate time source or doesn't possess a UTC server.

Algorithm

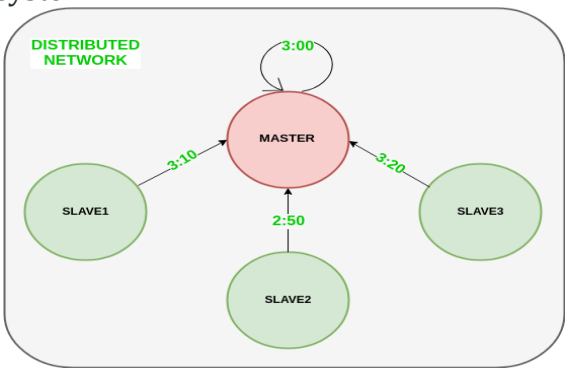
1) An individual node is chosen as the master node from a pool node in the network. This node is the main node in the network which acts as a master and the rest of the nodes act as slaves. The master node is chosen using an election process/leader election algorithm.

2) Master node periodically pings slaves nodes and fetches clock time at them using Cristian's algorithm.

The diagram below illustrates how the master sends requests to slave nodes.



The diagram below illustrates how slave nodes send back time given by their system clock.



3) Master node calculates the average time difference between all the clock times received and the clock time given by the master's system clock itself. This average time difference is added to the current time at the master's system clock and broadcasted over the network.

Features of Berkeley's Algorithm:

Centralized time coordinator: **Berkeley's Algorithm uses a centralized time coordinator, which is responsible for maintaining the global time and distributing it to all the client machines.**

Clock adjustment: The algorithm adjusts the clock of each client machine based on the difference between its local time and the time received from the time coordinator.
Average calculation: The algorithm calculates the average time difference between the client machines and the time coordinator to reduce the effect of any clock drift.
Fault tolerance: Berkeley's Algorithm is fault-tolerant, as it can handle failures in the network or the time coordinator by using backup time coordinators.
Accuracy: The algorithm provides accurate time synchronization across all the client machines, reducing the chances of errors due to time discrepancies.

Implementation:

1. **Program Code**
2. **Output screen shot with student name & roll no.**

Conclusion: Thus we Implemented Berkeley algorithm for clock synchronization.

ASSIGNMENT NO. 5

Problem Statement:

Implement token ring based mutual exclusion algorithm.

Tools / Environment:

Java Programming Environment, jdk 1.8

Related Theory:

Mutual exclusion

Fundamental to distributed systems is the concurrency and collaboration among multiple processes. In many cases, this also means that processes will need to simultaneously access the same resources. To prevent that such concurrent accesses corrupt the resource, or make it inconsistent, solutions are needed to grant mutual exclusive access by processes. In this section, we take a look at some important and representative distributed algorithms that have been proposed.

Overview

Distributed mutual exclusion algorithms can be classified into two different categories. In **token-based solutions** mutual exclusion is achieved by passing a special message between the processes, known as a **token**. There is only one token available and who ever has that token is allowed to access the shared resource. When finished, the token is passed on to a next process. If a process having the token is not interested in accessing the resource, it passes it on.

Token-based solutions have a few important properties. First, depending on how the processes are organized, they can fairly easily ensure that every process will get a chance at accessing the resource. In other words, they avoid **starvation**. Second, **deadlocks** by which several processes are indefinitely waiting for each other to proceed, can easily be avoided, contributing to their simplicity. The main drawback of token-based solutions is a rather serious one: when the token is lost (e.g., because the process holding it crashed), an intricate distributed procedure

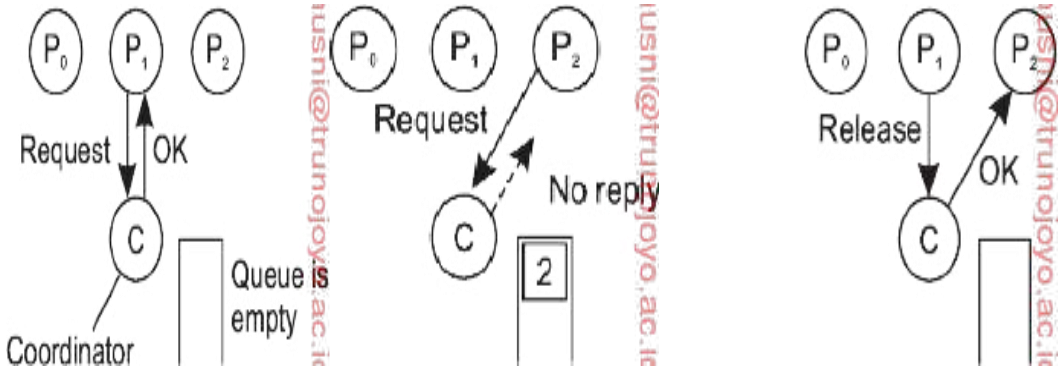
needs to be started to ensure that a new token is created, but above all, that it is also the only token.

As an alternative, many distributed mutual exclusion algorithms follow a **permission-based approach**. In this case, a process wanting to access the resource first requires the permission from other processes. There are many different ways toward granting such a permission and in the sections that follow we will consider a few of them.

A centralized algorithm

A straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator. Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission. If no other process is currently accessing that resource, the coordinator sends back a reply granting permission, as shown in Figure 6.15(a). When the reply arrives, the requester can go ahead.

Now suppose that another process, P2 in Figure 6.15(b) asks for permission to access the resource. The coordinator knows that a different process is already at the resource, so it cannot grant permission. The exact method used to deny permission is system dependent. In Figure 6.15(b) the coordinator just refrains from replying, thus blocking process P2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied." Either way, it queues the request from P2 for the time being and waits for more messages. When process P1 is finished with the resource, it sends a message to the coordinator releasing its exclusive access, as shown in Figure 6.15(c). The



(a) (b) (c)

Figure: (a) Process P1 asks for permission to access a shared resource. Permission is granted. (b) Process P2 asks permission to access the same resource, but receives no reply. (c) When P1 releases the resource, the coordinator replies to P2.

Coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e., this is the first message to it), it unblocks and accesses the resource. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later. Either way, when it sees the grant, it can go ahead as well.

It is easy to see that the algorithm guarantees mutual exclusion: the coordinator lets only one process at a time access the resource. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of resource (request, grant, release). Its simplicity makes it an attractive solution for many practical situations.

The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance

bottleneck. Nevertheless, the benefits coming from its simplicity outweigh in many cases the potential drawbacks.

A token-ring algorithm

A completely different approach to deterministically achieving mutual ex- clusion in a distributed system is illustrated in Figure 6.17. In software, we construct an overlay network in the form of a logical ring in which each process is assigned a position in the ring. All that matters is that each process knows who is next in line after itself.

When the ring is initialized, process P0 is given a **token**. The token circulates around the ring. Assuming there are *N* processes, the token is passed from process Pk to process $P_{(k+1) \bmod N}$ in point-to-point messages.

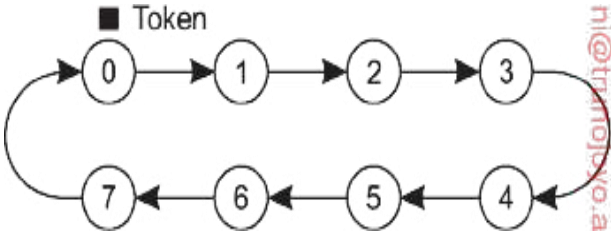


Figure : An overlay network constructed as a logical ring with a token circulating between its members.

When a process acquires the token from its neighbor, it checks to see if it needs to access the shared resource. If so, the process goes ahead, does all the work it needs to, and releases the resources. After it has finished, it passes the token along the ring. It is not permitted to immediately enter the resource again using the same token.

If a process is handed the token by its neighbor and is not interested in the resource, it just passes the token along. As a consequence, when no processes need the resource, the token just circulates around the ring.

The correctness of this algorithm is easy to see. Only one process has the token at any instant, so only one process can actually get to the resource. Since the token

circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to have access to the resource, at worst it will have to wait for every other process to use the resource.

This algorithm has its own problems. If the token is ever lost, for example, because its holder crashes or due to a lost message containing the token, it must be regenerated. In fact, detecting that it is lost may be difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it.

The algorithm also runs into trouble if a process crashes, but recovery is relatively easy. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintains the current ring configuration.

Conclusion: Thus we implemented token ring based mutual exclusion algorithm.

ASSIGNMENT NO. 6

Problem Statement:

Implement Bully and Ring algorithm for leader election.

Tools / Environment:

Java Programming Environment, jdk 1.8

Related Theory:

Election Algorithms

Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role. In general, it does not matter which process takes on this special responsibility, but one of them has to do it. In this section we will look at algorithms for electing a coordinator (using this as a generic name for the special process).

If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process P has a unique identifier *id*(P). In general, elec-tion algorithms attempt to locate the process with the highest identifier and designate it as coordinator. The algorithms differ in the way they locate the coordinator.

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

The bully algorithm

A well-known solution for electing a coordinator is the **bully lgorithm** de-vised by Garcia-Molina [1982]. In the following, we consider N processes

{ P0, . . . , P} N 1 and let *id*(Pk) = *k*. When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, Pk, holds an election as follows:

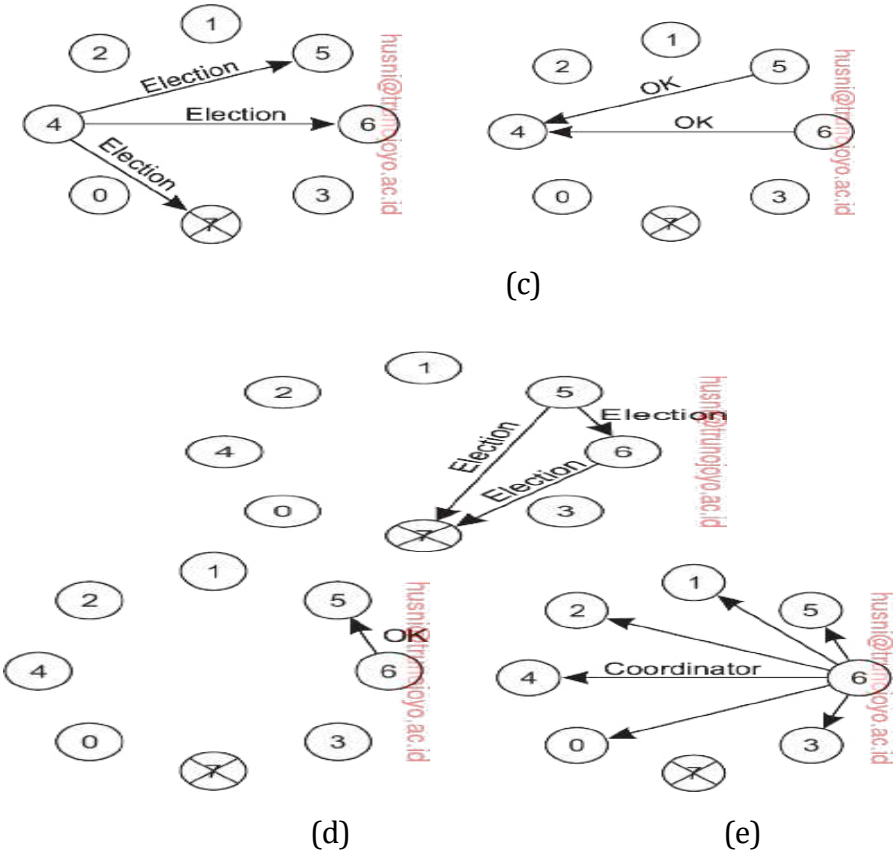
1. Pk sends an ELECTION message to all processes with higher identifiers: Pk+1, Pk+2, . . . , PN−1.
2. If no one responds, Pk wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over and Pk's job is done.

At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

In Figure 6.20 we see an example of how the bully algorithm works. The group consists of eight processes, with identifiers numbered from 0 to 7. Previously process P7 was the coordinator, but it has just crashed. Process P4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely P5, P6, and P7, as shown in Figure 6.20(a). Processes P5 and P6 both respond with OK, as shown in Figure 6.20(b). Upon getting the first of these responses, P4 knows that its job is over, knowing that either one of P5 or P6 will take over and become coordinator. Process P4 just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).

In Figure both P5 and P6 hold elections, each one sending messages only to those processes with identifiers higher than itself. In Figure



(c)

(d) (e)

Figure : The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

P6 tells P5 that it will take over. At this point P6 knows that P7 is dead and that it (P6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, P6 must now do what is needed. When it is ready to take over, it announces the takeover by sending a COORDINATOR message to all running processes. When P4 gets this message, it can now continue with the operation it was trying to do when it discovered that P7 was dead, but using P6 as the coordinator this time. In this way the failure of P7 is handled and the work can continue.

If process P7 is ever restarted, it will send all the others a COORDINATOR message and bully them into submission.

A ring algorithm

Consider the following election algorithm that is based on the use of a (logical) ring. Unlike some ring algorithms, this one does not use a token. We assume that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process identifier and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step along the way, the sender adds its own identifier to the list in the message effectively making itself a candidate to be elected as coordinator.

Eventually, the message gets back to the process that started it all. That pro- cess recognizes this event when it receives an incoming message containing its own identifier. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordi- nator is (the list member with the highest identifier) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.

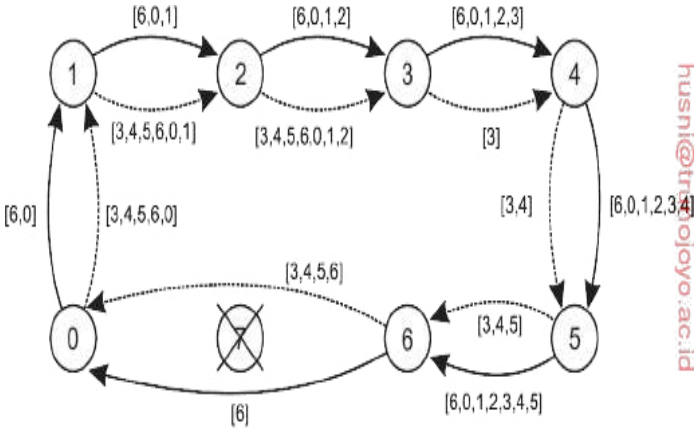


Figure : Election algorithm using a ring. The solid line shows the election messages initiated by P6; the dashed one those by P3.

In Figure we see what happens if two processes, P3 and P6, discover simultaneously that the previous coordinator, process P7, has crashed. Each of these builds an ELECTION message and each of them starts circulating its

message, independent of the other one. Eventually, both messages will go all the way around, and both P3 and P6 will convert them into COORDINATOR messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at worst it consumes a little bandwidth, but this is not considered wasteful.

The Bully Algorithm – This algorithm applies to system where every process can send a message to every other process in the system.

Algorithm – Suppose process P sends a message to the coordinator.

1. If the coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed.
2. Now process P sends an election messages to every process with high priority number.
3. It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.
4. Then it sends a message to all lower priority number processes that it is elected as their new coordinator.
5. However, if an answer is received within time T from any other process Q,
   - (I) Process P again waits for time interval T' to receive another message from Q that it has been elected as coordinator.
   - (II) If Q doesn't responds within time interval T' then it is assumed to have failed and algorithm is restarted.

The Ring Algorithm – This 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. Data structure that this algorithm uses is **active list**, a list that has a priority number of all active processes in the system.

Algorithm –

1. If process P1 detects a coordinator failure, it creates new active list which is empty initially. It sends election message to its neighbour on right and adds number 1 to its active list.
2. If process P2 receives message elect from processes on left, it responds in 3 ways:
   - (I) If message received does not contain 1 in active list then P1 adds 2 to its active list and forwards the message.

- (II) If this is the first election message it has received or sent, P1 creates new active list with numbers 1 and 2. It then sends election message 1 followed by 2.
- (III) If Process P1 receives its own election message 1 then active list for P1 now contains numbers of all the active processes in the system. Now Process P1 detects highest priority number from list and elects it as the new coordinator.

Implementation:

1. **Program Code**
2. **Output screen shot with student name & roll no.**

Conclusion:- Thus we have Implemented Bully and Ring algorithm for leader election.

ASSIGNMENT NO. 7

Problem Statement:

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

Tools / Environment:

Java Programming Environment, jdk 1.8

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

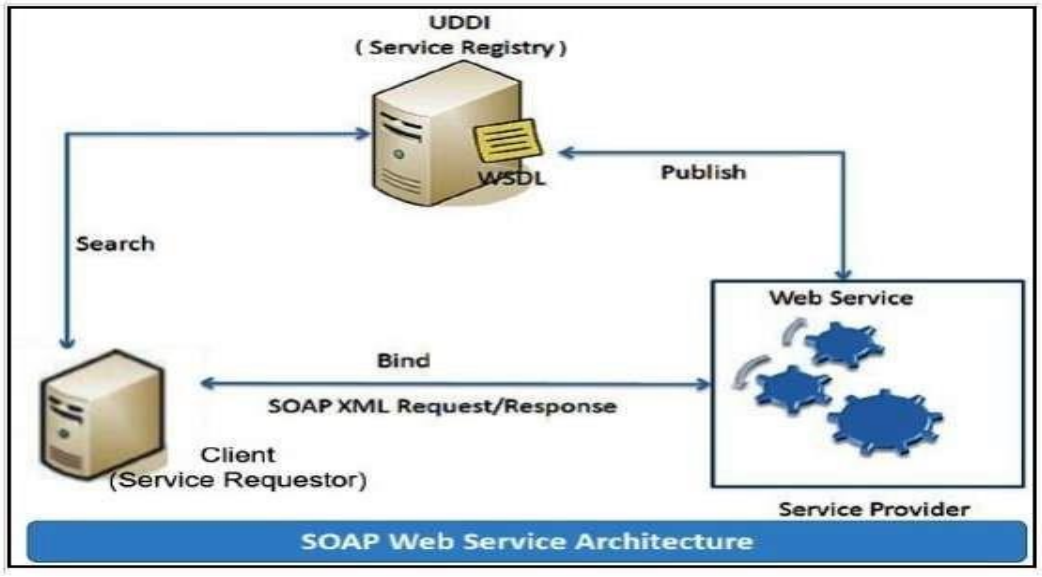
content as a SOAP message.
2. Then, the client program sends this SOAP message to the server web service as an HTTP POST request with the content passed as the body of the request.
3. The web service plays a crucial role in this step by understanding the SOAP request and Converting it into a set of instructions 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:

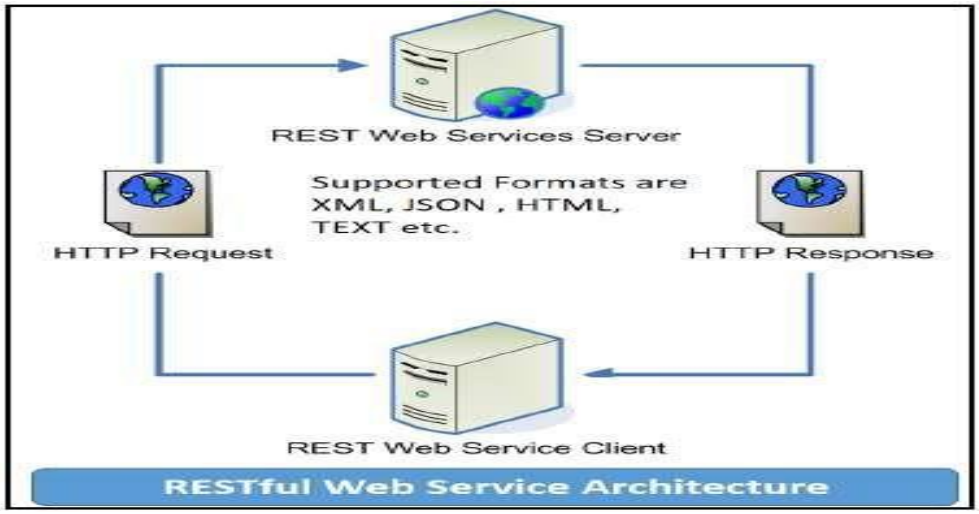
Universal Description, Discovery, and Integration (UDDI)*:* UDDI is an XMLbased framework for describing, discovering, and integrating web services. It acts as a directory of web service interfaces described in the WSDL language.

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:

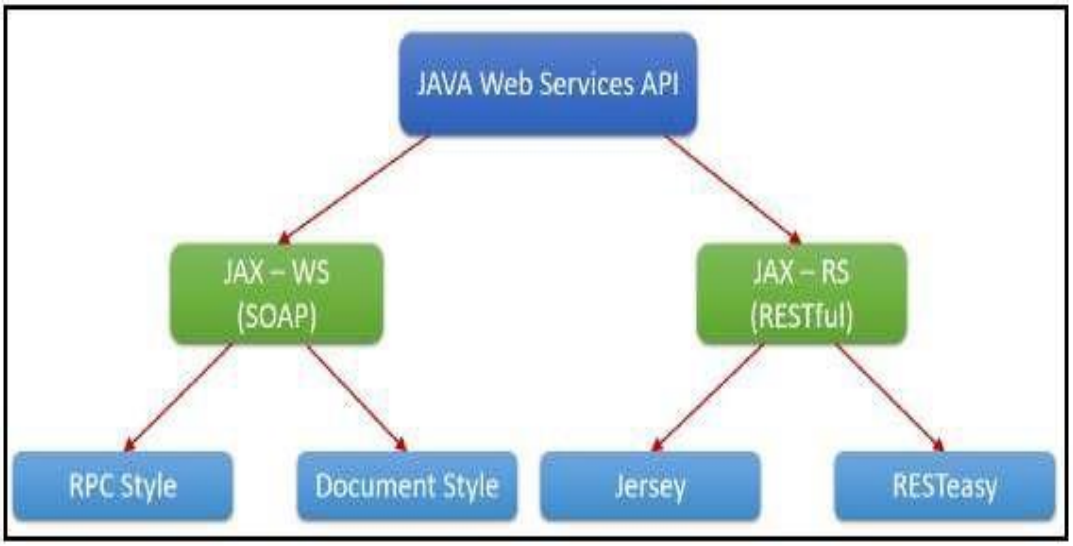


Solution:

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.



Implementation:

1. **Program Code**
2. **Output screen shot**

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.

String Reverse using corba , idl and java implementation :

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

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

```
idlj -fall  ReverseModule.idl
```

The idlj compiler generates a number of files.

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

```
javac *.java  ReverseModule/*.java
```

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

```
orbd -ORBInitialPort 1050&
```

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

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

6. Run the client application :

java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost

```
// Reverse 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();
       }
    }
}
//ReverseImpl

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));
    }
}

// Reverse Server

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

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

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

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

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

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

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

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

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

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

Conclusion: - Thus we have Created a simple web service and write any distributed application to consume the web service.