SCTR's

PUNE INSTITUTE OF COMPUTER TECHNOLOGY

FINAL YEAR
INFORMATION TECHNOLOGY
(2019 COURSE)

LABORATORY MANUAL
FOR

LABORATORY PRACTICE - V

SEMESTER - VIII

[Subject code: 414454]

[Prepared By]

Mrs. A. S, Kadam
Mrs. P. S. Shinde
Mrs. R. D. Kapadi
Mrs. A. S. Awati
Mrs. R. S. Malpani
Mr. G. S. Pise

DEPARTMENT OF INFORMATION TECHNOLOGY

INSTITUTE VISION AND MISSION

VISION

Pune Institute of Computer Technology aspires to be the leader inhigher technical education and research of international repute.

MISSION

To be leading and most sought-after Institute of education and research in emerging engineering and technology disciplines that attracts, retains and sustains gifted individuals of significant potential.

DEPARTMENT VISION AND MISSION

VISION

The department endeavors to be recognized globally as a center of academic excellence & research in Information Technology.

MISSION

To inculcate research culture among students by imparting information technology related fundamental knowledge, recent technological trends and ethics to get recognized as globally acceptable and socially responsible professionals.

Savitribai Phule Pune University, Pune Final Year Information Technology (2019 Course) 414454: Lab Practice - V		
Teaching Scheme:	**Credit Scheme:**	**Examination Scheme:**
Practical (PR): 4 hrs/week	**02 Credits**	**PR: 25 Marks** **TW: 50 Marks**

Prerequisites:

1. Operating Systems
2. Computer Network Technology
3. Web Application Development

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

Guidelines for Instructor's Manual

The faculty member should prepare the laboratory manual for all the experiments, and it should be made available to students and laboratory instructor/Assistant.
The instructor's manual should include prologue, university syllabus, conduction & Assessment guidelines, topics under consideration-concept, objectives, outcomes, algorithm written in pseudo language, sample test cases and references.

Guidelines for Student's Lab Journal

1. The laboratory assignments are to be submitted by students in the form of journals. The Journal consists of prologue, Certificate, table of contents, and handwritten/printed write-up of each assignment (Title, Objectives, Problem Statement, Outcomes, Software & Hardware requirements, Date of Completion, Assessment grade/marks and assessor's sign, Theory-Concept, algorithms, printouts of the code written using coding standards, sample test cases etc.)
2. Practical Examination will be based on the term work.
3. Candidate is expected to know the theory involved in the experiment.
4. The practical examination should be conducted if the journal of the candidate is completed in all respects and certified by concerned faculty and head of the department.

Guidelines for Lab /TW Assessment

Examiners will assess the term work based on performance of students considering the parameters such as timely conduction of practical assignment, methodology adopted for implementation of practical assignment, timely submission of assignment in the form of handwritten/printed write-up along with results of implemented assignment, attendance etc.
Examiners will judge the understanding of the practical performed in the examination by asking some

questions related to theory & implementation of experiments he/she has carried out.

Guidelines for Laboratory Conduction

Staff in-charge will suitably frame the assignments and flexibility may be incorporated. All the assignments should be conducted on the latest version of Open-Source Operating Systems, tools and Multi-core CPU supporting Virtualization and Multi-Threading.

Guidelines for Practical Examination

Both internal and external examiners should jointly set problem statements for practical examination. During practical assessment, the expert evaluator should give the maximum weightage to the satisfactory implementation of the problem statement. The supplementary and relevant questions may be asked at the time of evaluation to judge the student 's understanding of the fundamentals, effective and efficient implementation. The evaluation should be done by both external and internal examiners.

List of Laboratory 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

Reference Books:

1. Distributed Systems –Concept and Design, George Coulouris, Jean Dollimore, Tim Kindberg& Gordon Blair,Pearson,5th Edition,ISBN:978-13-214301-1.
2. Distributed Algorithms,Nancy Ann Lynch, Morgan Kaufmann Publishers, illustrated, reprint, ISBN: 9781558603486.
3. Java Network Programming & Distributed Computing by David Reilly, Michael Reilly
4. Distributed Systems - An Algorithmic approach by Sukumar Ghosh (good book for distributed algorithms)
5. Distributed Algorithms: Principles, Algorithms, and Systems by A. D. Kshemkalyani and M. Singhal (Good for algorithms, but very detailed, has lots of algorithms; good reference)
6. Design and Analysis of Distributed Algorithms by Nicola Santoro (good, distributed algorithms book)

ASSIGNMENT NO. 1

Problem Statement:

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

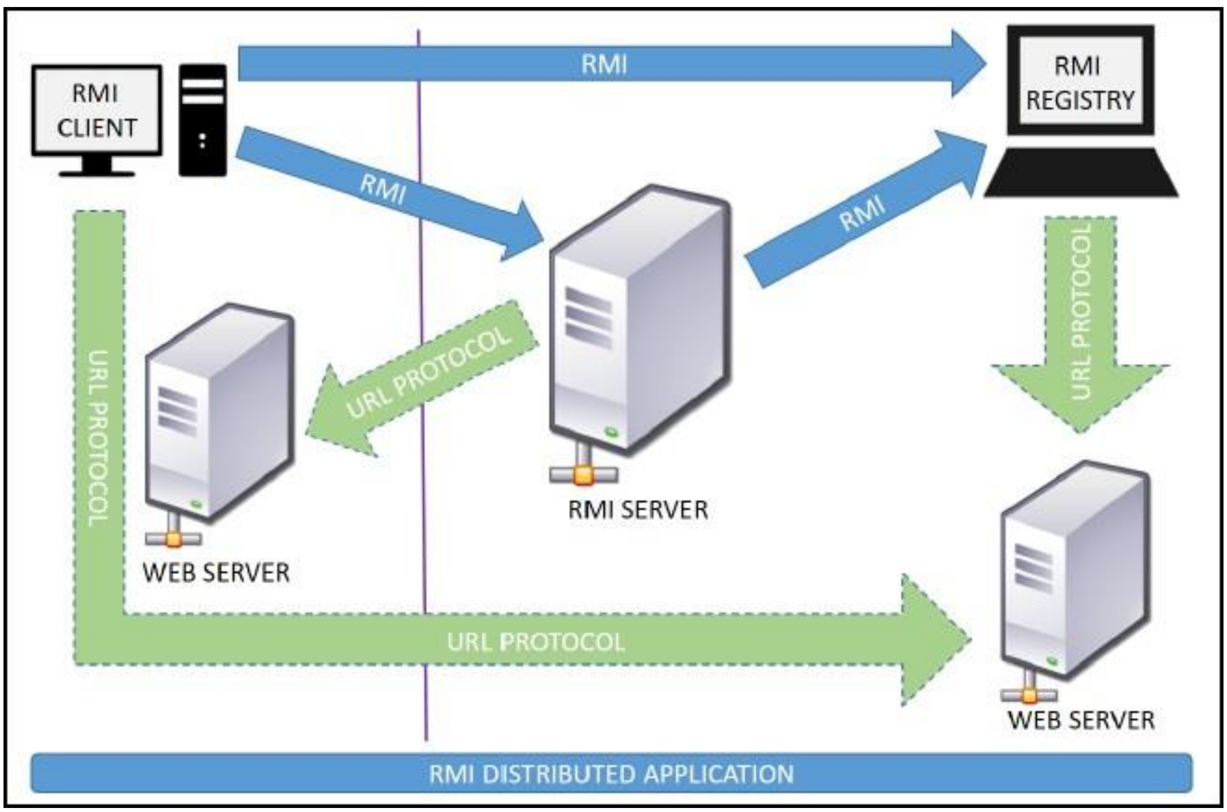
Tools / Environment:

Java Programming Environment, jdk 1.8, rmiregistry

Theory:

RMI provides communication between java applications that are deployed on different servers and connected remotely using objects called stub and skeleton. This communication architecturemakes 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 thehelp 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 beinvoked 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.

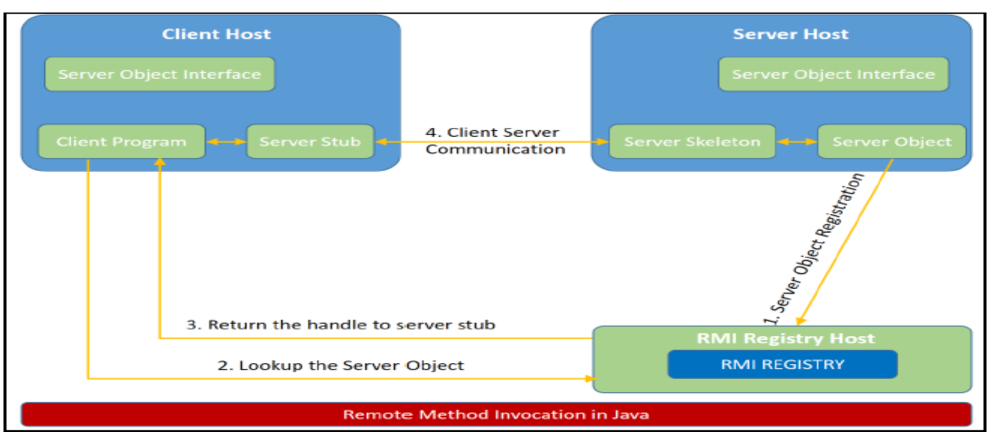
If any object invokes a method on the stub object, the stub establishes RMI by following these steps:

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, itestablishes 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.

1 Remote object implementation: Java allows a class to implement more than one interface ata 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.
2 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 doublearguments and returns their sum. All remote interfaces must extend the Remoteinterface, whichis 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 Namingclass (found in java.rmi). That method associates a namewith 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 thatare to be summed.

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

2. Generate a Stub

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

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

rmic AddServerImpl.
This command generates the file AddServerImpl_Stub.class.

3. Install Files on the Client and Server Machines

Copy AddClient.class, AddServerImpl_Stub.class, AddServerIntf.class
to a directory on the client machine.

Copy AddServerIntf.class, AddServerImpl.class, AddServerImpl_Stub.class, and AddServer.classto 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 mapsnames 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:

AddServerIntf.java

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

AddServerImpl.java

```java
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;
}
}
```

AddServer.java

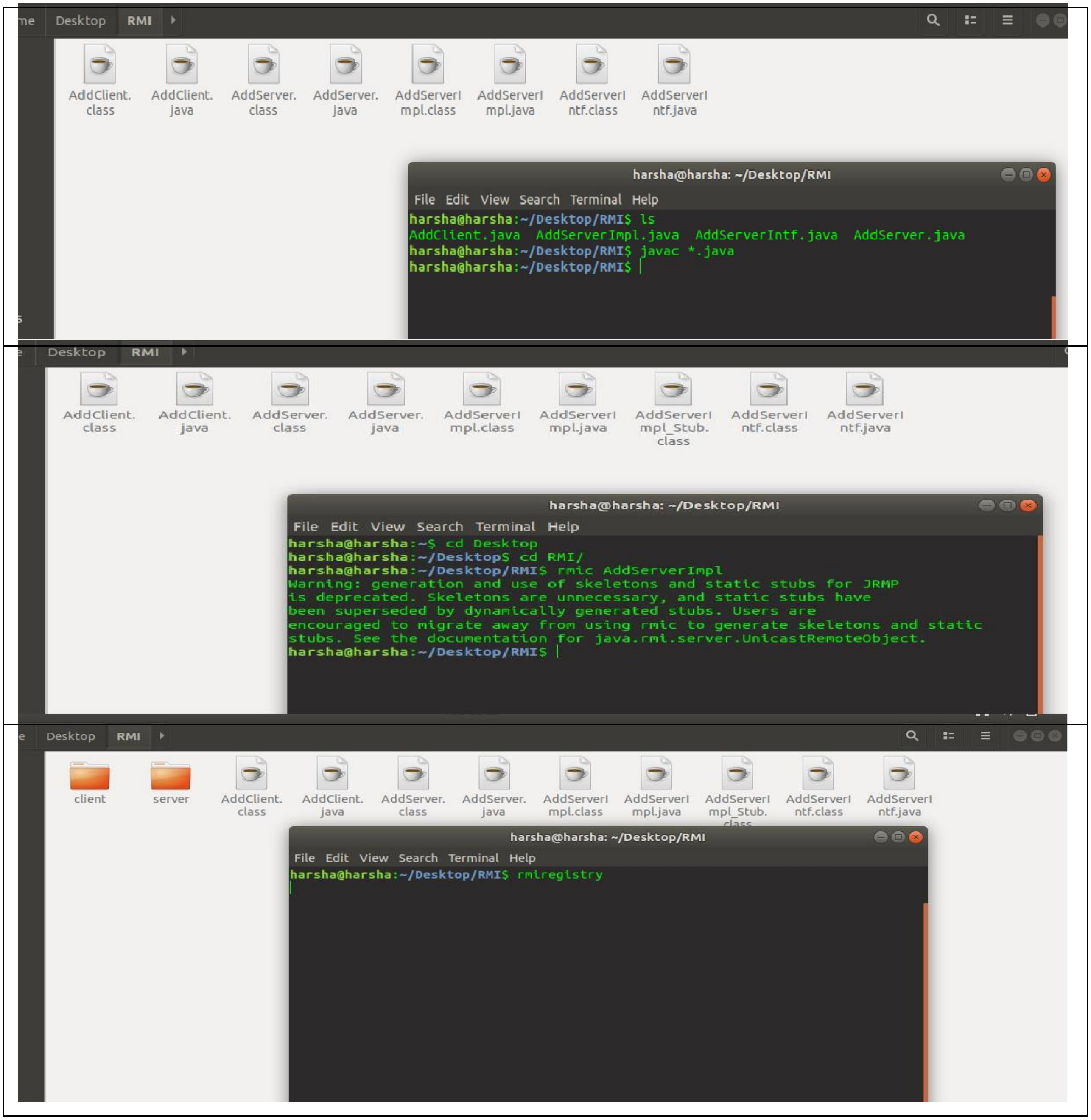
```java
import java.net.*;
import java.rmi.*;
public class AddServer {
public static void main(String args[]) {
try {
AddServerImpl addServerImpl = new AddServerImpl();
Naming.rebind("AddServer", addServerImpl);
}
catch(Exception e) {
System.out.println("Exception: " + e);
}
}
}
```

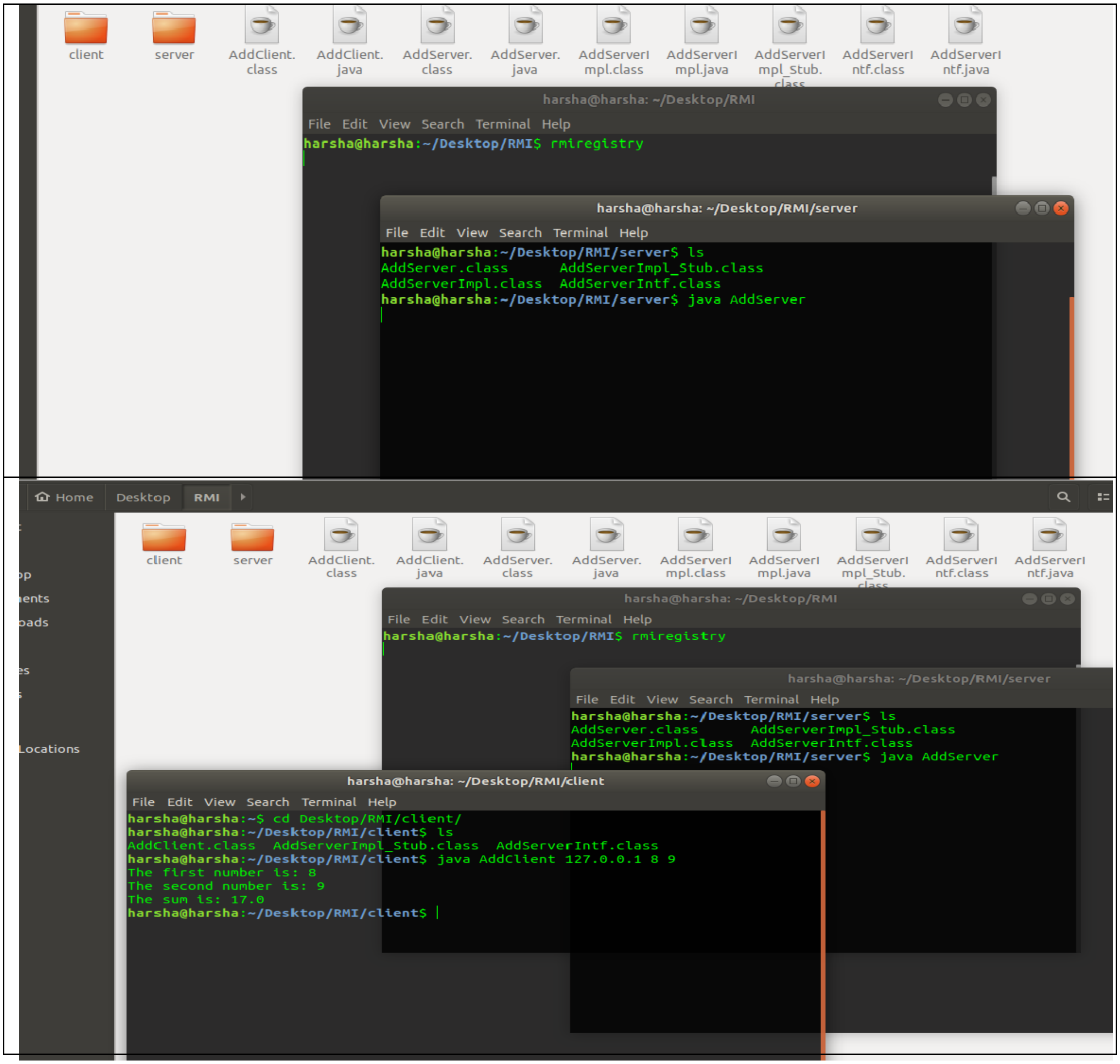
AddClient.java

```java
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 executeson 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 CORBA to demonstrate object brokering. (Calculator or String operations).

Tools / Environment:

Java Programming Environment, JDK 1.8

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.

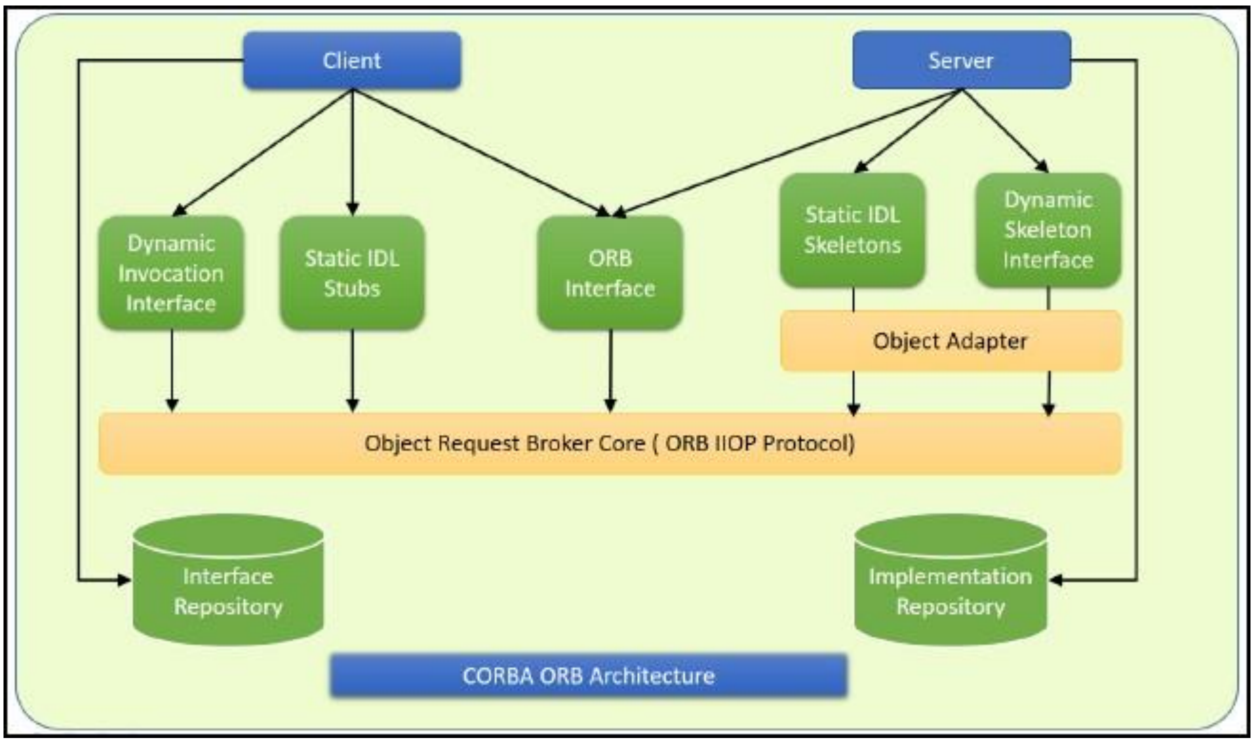
An application developed based on CORBA standards with standard **Internet Inter-ORB Protocol** (**IIOP**), irrespective of the vendor that develops it, should be able to smoothly integrateand operate with another application developed based on CORBA standards through the same or different vendor.

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 bythe client operation. Again, during response handling, the interface definition is helpful tomarshal (send from the server) and unmarshal (receive and read the response) arguments on the client side once received.

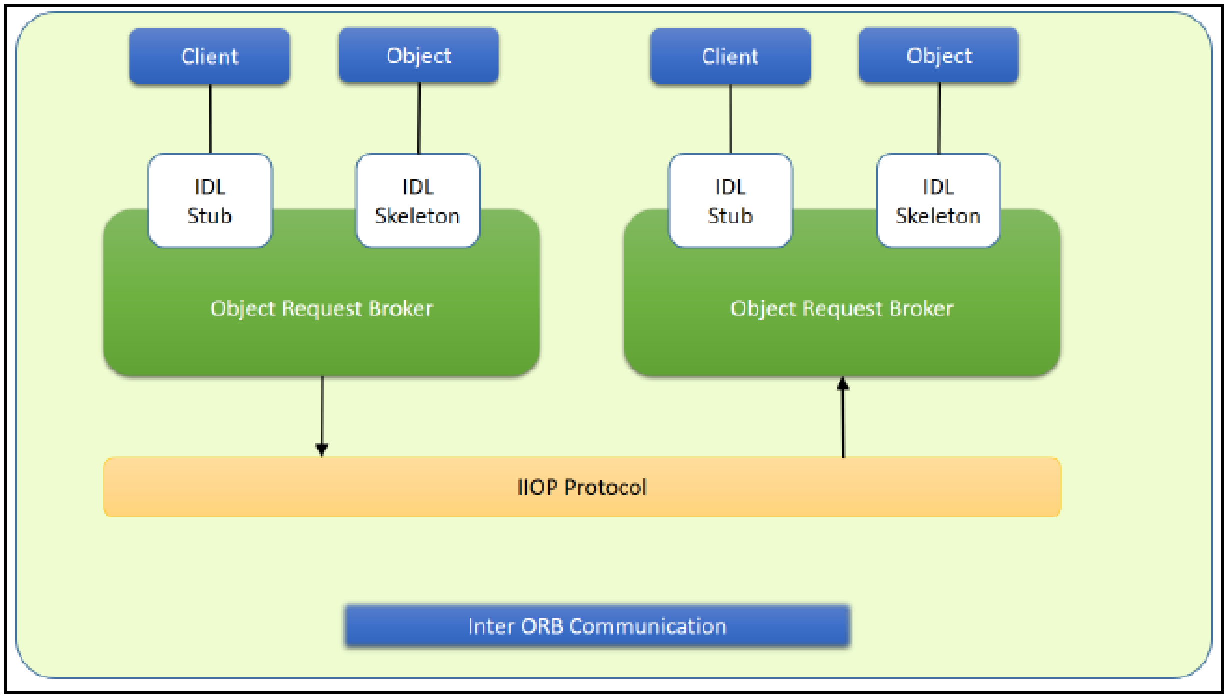
The IDL interface is a design concept that works with multiple programming languages including C, C++, Java, Ruby, Python, and IDLscript. This is close to writing a program to an interface, a concept we have been discussing that most recent programming languages and frameworks, such as Spring. The interface 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 theinterface 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 clientfor 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 skeletonrepresent the client and server, respectively, to their counterparts. They help establish thiscommunication 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, servicesto 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 keyenterprise 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 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 cannotbe 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 Packagesection 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 theinterface, 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 it's own CORBA-enabled interface and it implementation. This involves doing the following:

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

Interfaces in IDL are declared much like interfaces in Java.

1.1. 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 ...
};
};
```

1.2. 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 Serverinterface.

1.4 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.

1.5 A complete IDL example

Now let's tie all these basic elements together. Here's a complete IDL example thatdeclares 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 PrintServerinterface 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 inkeyword.

2. Turning IDL Into Java

Once the remote interfaces in IDL are described, you need to generate Java classes that actas 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 usedas 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 ableto 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 outand inoutarguments to be wrappedin Java holder classes, which are filled with the output value of the argument when the method returns.

The idltojtool generate 2 other classes:

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

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

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

3. **Writing the Implementation**

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

Implementing the solution:

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

1. **Defining the Interface (`Hello.idl`)**

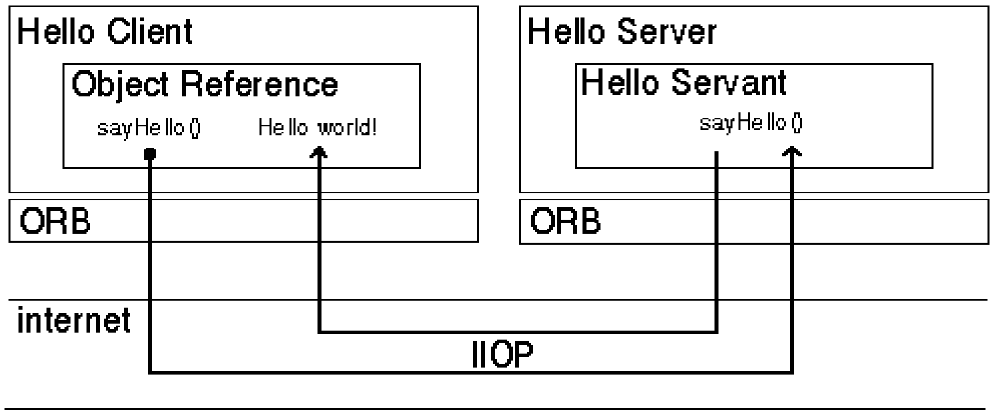
The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL).To complete the application, you simply provide the server **(`HelloServer.java`)** and client **(`HelloClient.java`)** implementations.

2. **Implementing the Server (HelloServer.java)**

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

The HelloServerclass 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 Helloobject) and tells theORB about it
- Gets a CORBA object reference for a naming context in which to register the new CORBAobject
- 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:

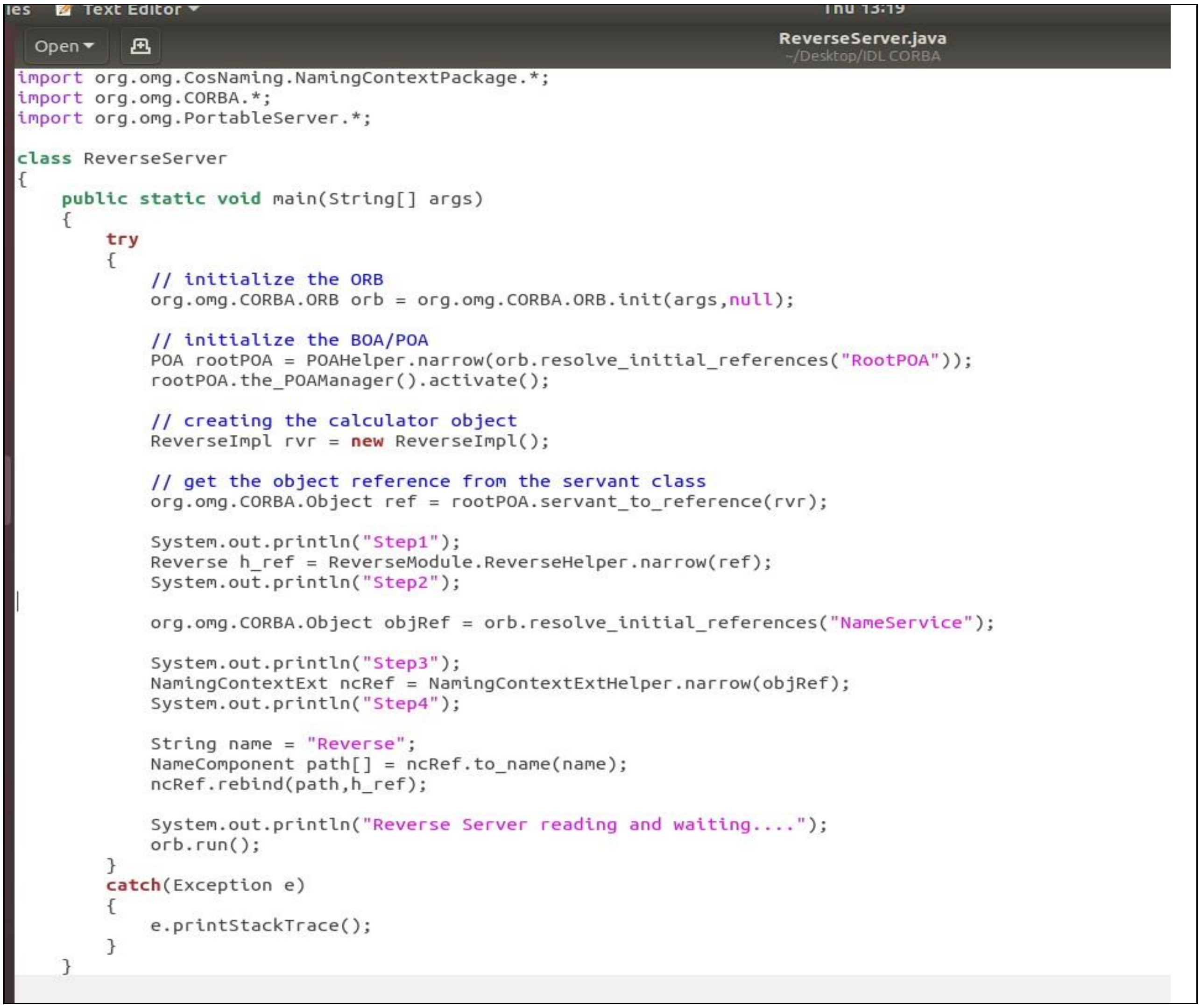
```
module ReverseModule
{
    interface Reverse
    {
        string reverse_string(in string str);
    };
};
```

ReverseClient.java
~/Desktop/IDL CORBA

```java
// 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.java
~/Desktop/IDL CORBA

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

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



```
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 **orbd**, a daemon process containing a Bootstrap Service, a Transient Naming Service,

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

1. Change to the directory that contains the file `Hello.idl`.

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

idlj -fall Hello.idl

You must use the -falloption with the idljcompiler 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 idljcompiler for Hello.idl, with the -fallcommand lineoption, 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, andimplements 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.ObjectImpland implements the Hello.javainterface.

- Hello.java:

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

- HelloHelper.java

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

- HelloHolder.java

This final class holds a public instance member of type Hello. Whenever the IDL typeis an outor an inoutparameter, the Holder class is used. It provides operationsfor org.omg.CORBA.portable.OutputStream and org.omg.CORBA.portable.InputStreamarguments, which CORBA allows, but which do not map easilyto Java's semantics. The Holder class delegates to the methods in the Helper class forreading and writing. It implements org.omg.CORBA.portable.Streamable.

- HelloOperations.java

This interface contains the methods sayHello() and shutdown(). The IDL-to-Javamapping 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 yourpath.

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

4. Start orbd.

To start orbdfrom 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 HelloServerfrom a UNIX command shell, enter:

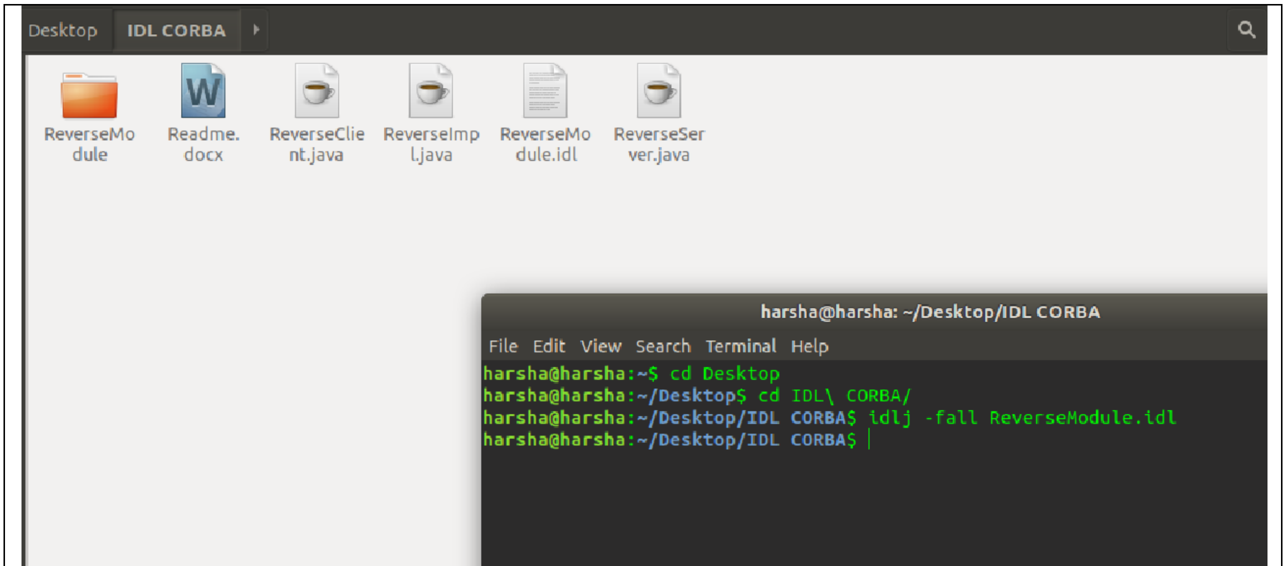
```
java HelloServer -ORBInitialPort 1050 -ORBInitialHostlocalhost&
```

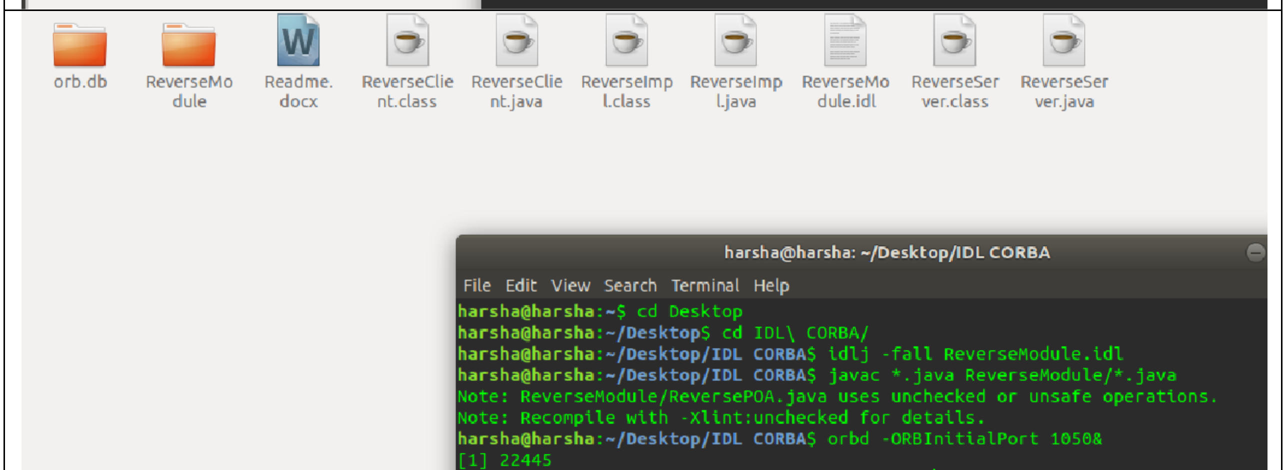
You will see HelloServerready and waiting... when the server is started.

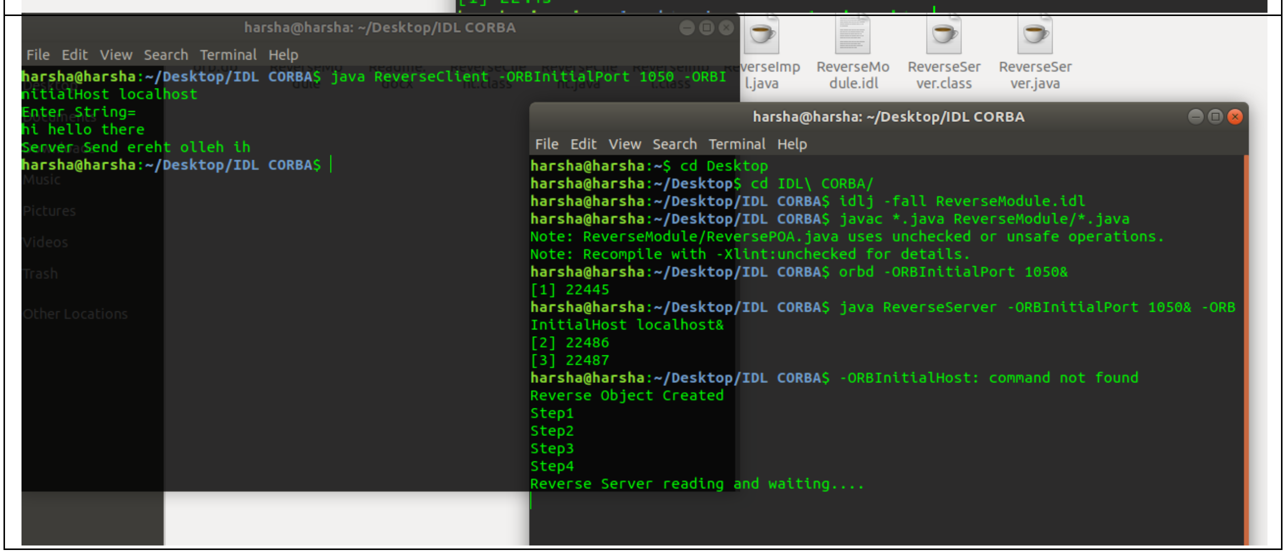
6. Run the client application:

```
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).


Desktop
IDL CORBA
ReverseModule
Readme.docx
ReverseClient.java
ReverseImpl.java
ReverseModule.idl
ReverseServer.java
harsha@harsha: ~/Desktop/IDL CORBA
File Edit View Search Terminal Help
harsha@harsha:~$ cd Desktop
harsha@harsha:~/Desktop$ cd IDL\ CORBA/
harsha@harsha:~/Desktop/IDL CORBA$ idlj -fall ReverseModule.idl
harsha@harsha:~/Desktop/IDL CORBA$


orb.db
ReverseModule
Readme.docx
ReverseClient.class
ReverseClient.java
ReverseImpl.class
ReverseImpl.java
ReverseModule.idl
ReverseServer.class
ReverseServer.java
harsha@harsha: ~/Desktop/IDL CORBA
File Edit View Search Terminal Help
harsha@harsha:~$ cd Desktop
harsha@harsha:~/Desktop$ cd IDL\ CORBA/
harsha@harsha:~/Desktop/IDL CORBA$ idlj -fall ReverseModule.idl
harsha@harsha:~/Desktop/IDL CORBA$ javac *.java ReverseModule/*.java
Note: ReverseModule/ReversePOA.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
harsha@harsha:~/Desktop/IDL CORBA$ orbd -ORBInitialPort 1050&
[1] 22445


harsha@harsha: ~/Desktop/IDL CORBA
File Edit View Search Terminal Help
harsha@harsha:~/Desktop/IDL CORBA$ java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost
Enter String=
hi hello there
Server Send ereht olleh ih
harsha@harsha:~/Desktop/IDL CORBA$
ReverseImpl.java
ReverseModule.idl
ReverseServer.class
ReverseServer.java
harsha@harsha: ~/Desktop/IDL CORBA
File Edit View Search Terminal Help
harsha@harsha:~$ cd Desktop
harsha@harsha:~/Desktop$ cd IDL\ CORBA/
harsha@harsha:~/Desktop/IDL CORBA$ idlj -fall ReverseModule.idl
harsha@harsha:~/Desktop/IDL CORBA$ javac *.java ReverseModule/*.java
Note: ReverseModule/ReversePOA.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
harsha@harsha:~/Desktop/IDL CORBA$ orbd -ORBInitialPort 1050&
[1] 22445
harsha@harsha:~/Desktop/IDL CORBA$ java ReverseServer -ORBInitialPort 1050& -ORBInitialHost 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, servicesto 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.

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, JDK1.8 or higher, MPI Library (mpi.jar), MPJ Express (mpj.jar)

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 beexecuted 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 MPIprogram.

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 sendfunction is used by the source process to define the data and establish the connectionof 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)

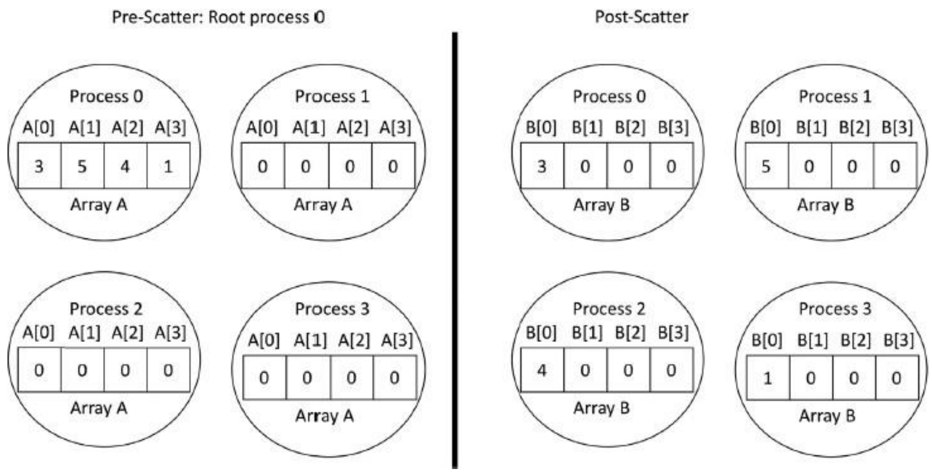
The first three operands establish the data to be transferred between the source and destination processes. The first argument points to the message content itself, which may be a simple scalar or a group of data. The message data content is described by the next two arguments. The second operand specifies 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 theconnection 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).**

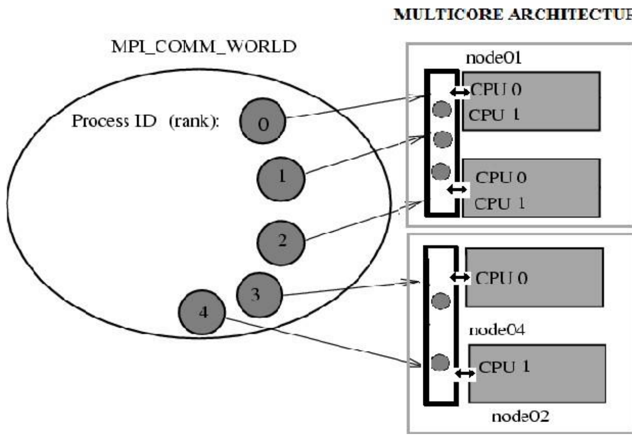
The multicore configuration is meant for users who plan to write and execute parallel Java applications using MPJ Express on their desktops or laptops which contains shared memory and multicore processors. In this configuration, users can write their message passing parallel application using MPJ Express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

Designing the solution:

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

Implementing the solution:

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 ScatterGather parallel Java program and save it as ScatterGather.java.
3. Compile a ScatterGather.javaparallel Java program
4. Running MPJ Express in the Multi-core Configuration.

Writing the source code:

Asign2.java

```
import mpi.*;
public class Asign2 {

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 $MPJ_HOME/lib/mpj.jar Asign2.java` (mpj.jar is inside lib folder in the downloaded MPJ Express)
Execute: `$MPJ_HOME/bin/mpjrun.sh -np 4 Asign2`

```
dos@dospc  ~/Desktop/Junaid/Asign2  mpjrun.sh -np 2 Asign2
MPJ Express (0.44) is started in the multicore configuration
Hi from <0>
Hi from <1>
dos@dospc  ~/Desktop/Junaid/Asign2  mpjrun.sh -np 4 Asign2
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.4

Problem Statement:

Implement Berkeley algorithm for clock synchronization.

Tools/Environment:

Python3, Eclipse IDE

Theory :

Distributed System is a collection of computers connected via the high-speed communication network. In the distributed system, the hardware and software components communicate and coordinate their actions by message passing. Each node in distributed systems can share their resources with other nodes. So, there is need of proper allocation of resources to preserve the state of resources and help coordinate between the several processes. To resolve such conflicts, synchronization is used. Synchronization in distributed systems is achieved via clocks.

The physical clocks are used to adjust the time of nodes. Each node in the system can share its local time with other nodes in the system. The time is set based on UTC (Universal Time Coordination). UTC is used as a reference time clock for the nodes in the system.

The clock synchronization can be achieved by 2 ways: External and Internal Clock Synchronization.

- **External clock synchronization** is the one in which an external reference clock is present. It is used as a reference and the nodes in the system can set and adjust their time accordingly.
- **Internal clock synchronization** is the one in which each node shares its time with other nodes and all the nodes set and adjust their times accordingly.

There are 2 types of clock synchronization algorithms: Centralized and Distributed.

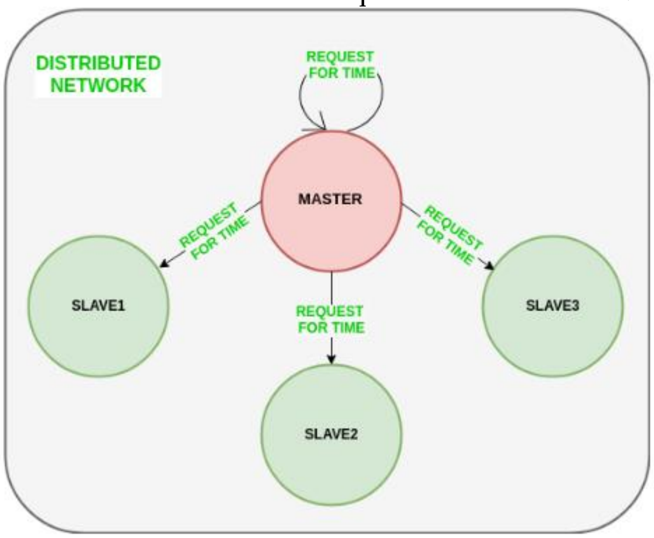
- **Centralized** is the one in which a time server is used as a reference. The single time server propagates its time to the nodes and all the nodes adjust the time accordingly. It is dependent on single time server so if that node fails, the whole system will lose synchronization. Examples of centralized are- Berkeley Algorithm, Passive Time Server, Active Time Server etc.
- **Distributed** is the one in which there is no centralized time server present. Instead the nodes adjust their time by using their local time and then, taking the average of the differences of time with other nodes. Distributed algorithms overcome the issue of centralized algorithms like the scalability and single point failure. Examples of Distributed algorithms are – Global Averaging Algorithm, Localized Averaging Algorithm, NTP (Network time protocol) etc.

Berkeley Algorithm

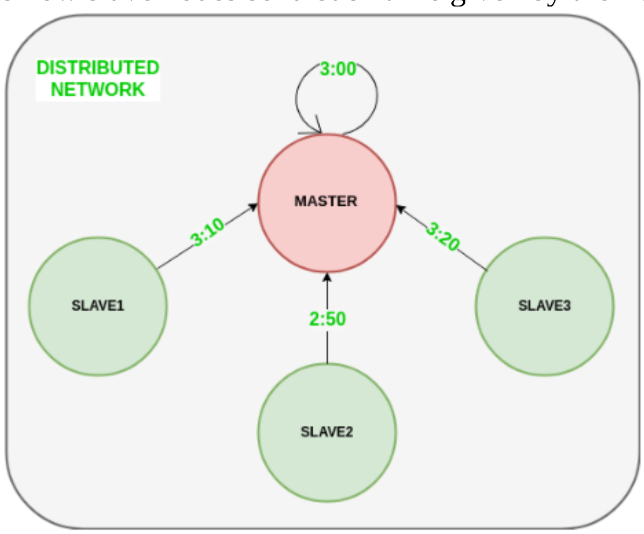
Berkeley's Algorithm is a clock synchronization technique used in distributed systems. The algorithm assumes that each machine node in the network either doesn't have an accurate time source or doesn't possess a UTC server.

Algorithm :
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.

1. 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.

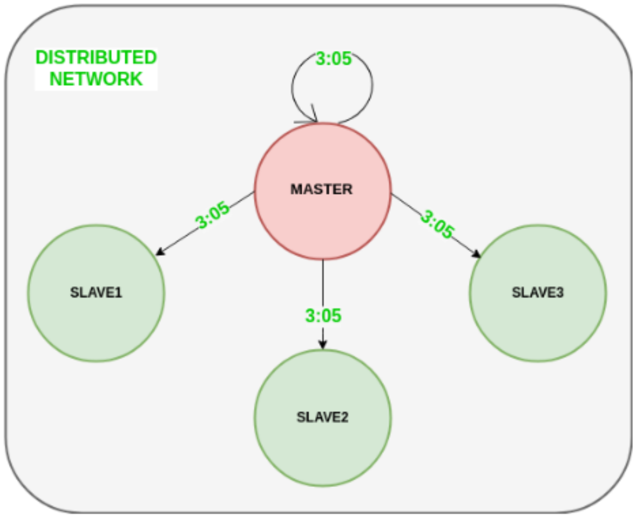


   The diagram below illustrates how slave nodes send back time given by their system clock.



   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.

The diagram below illustrates the last step of Berkeley's algorithm.



Scope of Improvement

- Ignoring significant outliers in the calculation of average time difference
- In case the master node fails/corrupts, a secondary leader must be ready/pre-chosen to take the place of the master node to reduce downtime caused due to the master's unavailability.
- Instead of sending the synchronized time, master broadcasts relative inverse time difference, which leads to a decrease in latency induced by traversal time in the network while the time of calculation at slave node.

The code below is a python script that can be used to trigger a master clock server.

<u># program imitating a clock server</u>

```python
# Python3 program imitating a clock server

from functools import reduce
from dateutil import parser
import threading
import datetime
import socket
import time


# datastructure used to store client address and clock data
client_data = {}


''' nested thread function used to receive
    clock time from a connected client '''
def startReceivingClockTime(connector, address):

    while True:
        # receive clock time
        clock_time_string = connector.recv(1024).decode()
        clock_time = parser.parse(clock_time_string)
        clock_time_diff = datetime.datetime.now() - \
                                                 clock_time

        client_data[address] = {
                    "clock_time"      : clock_time,
                    "time_difference" : clock_time_diff,
                    "connector"   : connector
                    }

        print("Client Data updated with: "+ str(address),
                                             end = "\n\n")
```

```python
        time.sleep(5)


''' master thread function used to open portal for
    accepting clients over given port '''
def startConnecting(master_server):

    # fetch clock time at slaves / clients
    while True:
        # accepting a client / slave clock client
        master_slave_connector, addr = master_server.accept()
        slave_address = str(addr[0]) + ":" + str(addr[1])

        print(slave_address + " got connected successfully")

        current_thread = threading.Thread(
                        target = startReceivingClockTime,
                        args = (master_slave_connector,
                                        slave_address, ))
        current_thread.start()


# subroutine function used to fetch average clock difference
def getAverageClockDiff():

    current_client_data = client_data.copy()

    time_difference_list = list(client['time_difference']
                                for client_addr, client
                                    in client_data.items())


    sum_of_clock_difference = sum(time_difference_list, \
```

Welcome berkey.py

C: > Users > Lenovo > Desktop > berkey.py

```python
                                    datetime.timedelta(0, 0))

    average_clock_difference = sum_of_clock_difference \
                                        / len(client_data)

    return average_clock_difference


''' master sync thread function used to generate
    cycles of clock synchronization in the network '''
def synchronizeAllClocks():

    while True:

        print("New synchronization cycle started.")
        print("Number of clients to be synchronized: " + \
                                    str(len(client_data)))

        if len(client_data) > 0:

            average_clock_difference = getAverageClockDiff()

            for client_addr, client in client_data.items():
                try:
                    synchronized_time = \
                        datetime.datetime.now() + \
                                    average_clock_difference

                    client['connector'].send(str(
                            synchronized_time).encode())

                except Exception as e:
                    print("Something went wrong while " + \
```

Welcome berkey.py

C: > Users > Lenovo > Desktop > berkey.py

```python
                    print("Something went wrong while " + \
                        "sending synchronized time " + \
                        "through " + str(client_addr))

        else :
            print("No client data." + \
                        " Synchronization not applicable.")

        print("\n\n")

        time.sleep(5)


# function used to initiate the Clock Server / Master Node
def initiateClockServer(port = 8080):

    master_server = socket.socket()
    master_server.setsockopt(socket.SOL_SOCKET,
                                socket.SO_REUSEADDR, 1)

    print("Socket at master node created successfully\n")

    master_server.bind(('', port))

    # Start listening to requests
    master_server.listen(10)
    print("Clock server started...\n")

    # start making connections
    print("Starting to make connections...\n")
    master_thread = threading.Thread(
                        target = startConnecting,
                        args = (master_server, ))
```

```python
                    args = (master_server, ))
    master_thread.start()

    # start synchronization
    print("Starting synchronization parallelly...\n")
    sync_thread = threading.Thread(
                    target = synchronizeAllClocks,
                    args = ())
    sync_thread.start()



# Driver function
if __name__ == '__main__':

    # Trigger the Clock Server
    initiateClockServer(port = 8080)
```


Python3 program imitating a client process

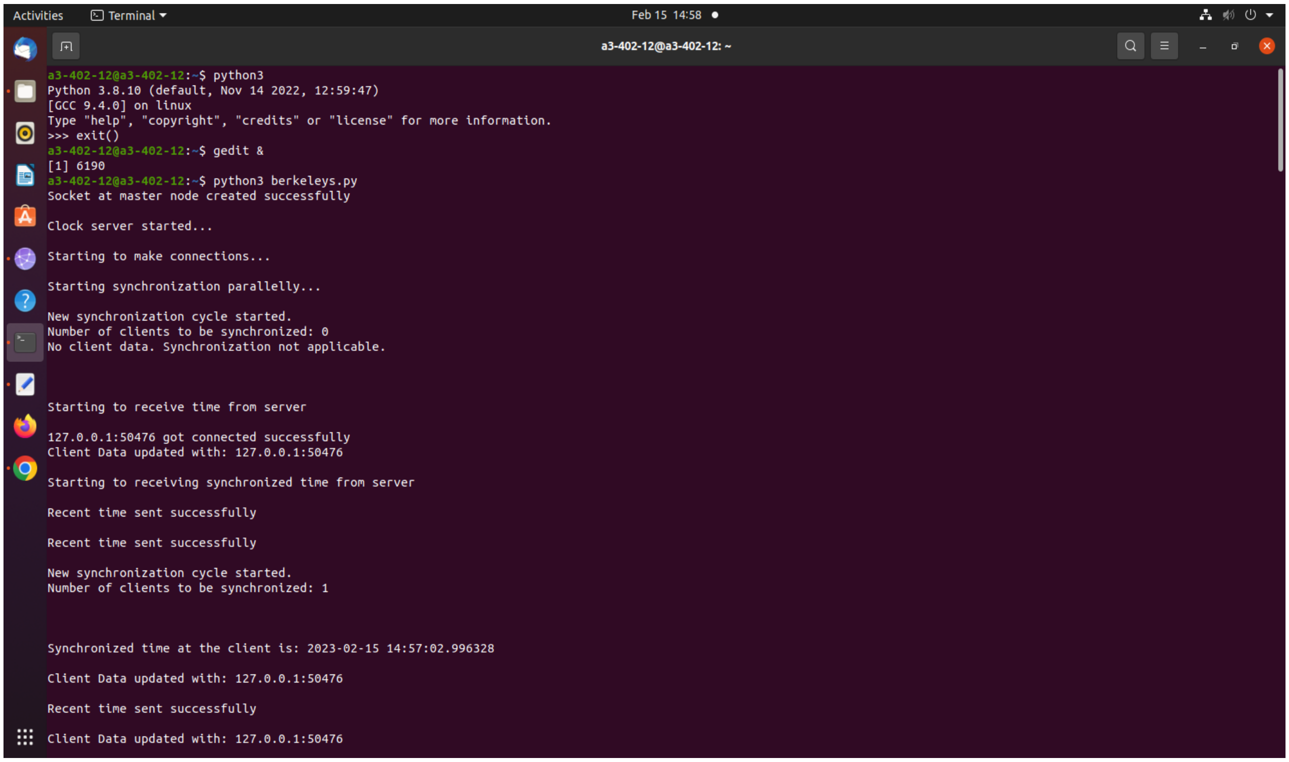
```python
    # Python3 program imitating a client process

from timeit import default_timer as timer
from dateutil import parser
import threading
import datetime
import socket
import time


# client thread function used to send time at client side
def startSendingTime(slave_client):

    while True:
        # provide server with clock time at the client
        slave_client.send(str(
                    datetime.datetime.now()).encode())

        print("Recent time sent successfully",
                                end = "\n\n")
        time.sleep(5)


# client thread function used to receive synchronized time
def startReceivingTime(slave_client):

    while True:
        # receive data from the server
        Synchronized_time = parser.parse(
                    slave_client.recv(1024).decode())

        print("Synchronized time at the client is: " + \
                                    str(Synchronized_time),
```

Activate Windows
Go to Settings to activate Wi

```python
            print("Synchronized time at the client is: " + \
                                    str(Synchronized_time),
                                    end = "\n\n")


# function used to Synchronize client process time
def initiateSlaveClient(port = 8080):

    slave_client = socket.socket()

    # connect to the clock server on local computer
    slave_client.connect(('127.0.0.1', port))

    # start sending time to server
    print("Starting to receive time from server\n")
    send_time_thread = threading.Thread(
                    target = startSendingTime,
                    args = (slave_client, ))
    send_time_thread.start()


    # start receiving synchronized from server
    print("Starting to receiving " + \
                        "synchronized time from server\n")
    receive_time_thread = threading.Thread(
                    target = startReceivingTime,
                    args = (slave_client, ))
    receive_time_thread.start()


# Driver function
if __name__ == '__main__':
```

```python

# Driver function
if __name__ == '__main__':

    # initialize the Slave / Client
    initiateSlaveClient(port = 8080)
```

How to run code in python:



Conclusion :

Thus, we have implemented the 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

Theory :

Process synchronization is the set of techniques that are used to coordinate execution among processes. For example, a process may need to run to a certain point, at which point it will stop and wait for another process to finish certain actions. A shared resource, such as a file or fields in a database, may require exclusive access and processes have to coordinate among themselves to ensure that access to the resource is fair and exclusive.

The modification of a resource may be implemented as an atomic operation. That is, there might be a single remote procedure call or web service that performs an operation on the resource. In this case, the server may be able to handle access to the resource on its own. In many cases, however, there might be concurrent requests on different servers that are competing for the same resource. There are also frequently situations where resource access is not inherently atomic and a client will need to request a *lock* on the resource and release it at a later time when all modifications are complete. Distributed transactions are an example of a framework that relies on this.

In centralized systems, it was common to enforce exclusive access to shared code. Mutual exclusion was accomplished through mechanisms such as test and set locks in hardware and semaphores, messages, and condition variables in software.

Mutual exclusion in distributed systems:
We assume that there is group agreement on how a resource or critical section is identified (e.g., a name or number) and that this identifier is passed as a parameter with any requests.
We also assume that processes can be uniquely identified throughout the system (e.g., using a combination of machine ID and process ID). The goal is to get a lock on a resource: permission to access the resource exclusively. When a process is finished using the resource, it releases the lock, allowing another process to get the lock and access the resource.
Any viable mutual exclusion algorithm must satisfy three properties:

- Safety
  This means the algorithm does indeed provide mutual exclusion. At any instant, only one process may hold the resource.
- Liveness
  The algorithm should make progress. Processes should not wait forever for messages that will never arrive.
- Fairness
  Each process should a fair chance to hold the resource. This means that a process that asks for a critical section should not wait forever to get it. No process should experience *starvation*. For example, two processes cannot transfer their lock on a resource back and forth to each other while a third process that wants the resource is denied from grabbing a lock.

There are three categories of mutual exclusion algorithms:

1. Centralized algorithms use a central coordinator. A process can access a resource because a central coordinator allowed it to do so.
2. Token-based algorithms move a token around. A process can access a resource if it is holding a token permitting it to do so.
3. Contention-based algorithms use a distributed algorithm that sorts out points of conflict (contention) where two or more processes may want access to the same resource.

Central server algorithm

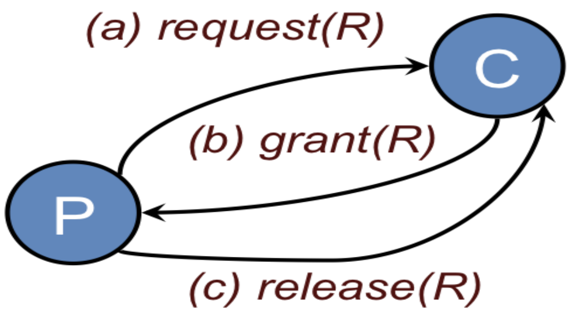


Figure 1. Centralized mutual exclusion

The central server algorithm simulates a single processor system. One process in the distributed system is elected as the coordinator (Figure 1). When a process wants to enter a resource, it sends a request message (Figure 1a) identifying the resource, if there are more than one, to the coordinator.

If nobody is currently in the section, the coordinator sends back a grant message (Figure 1b) and marks that process as using the resource. If, however, another process has previously claimed the resource, the server simply does not reply, so the requesting process is blocked. The coordinator keeps state on which process is currently granted the resource and a which processes are requesting the resource. The list of requestors is a first-come, first-served queue per resource. If some process has been granted access to the resource but has not released it, any incoming *grant* requests are queued on the coordinator. When a coordinator receives a *release(R)* message, it sends a *grant* message to the next process in the queue for resource *R*.

When a process is done with its resource, it sends a release message (Figure 1c) to the coordinator. The coordinator then can send a grant message to the next process in its queue of processes requesting a resource (if any).

This algorithm is easy to implement and verify. It is fair in that all requests are processed in order. Unfortunately, it suffers from having a single point of failure. Another issue is that a process cannot distinguish between being blocked (not receiving a grant because someone else is in the resource) and not getting a response because the coordinator is down. From the coordinator's side, the coordinator does not know if a process using a resource has died, is in an infinite loop, or is simply taking a longer time to release a resource. Moreover, a centralized server can be a potential bottleneck in systems with a huge number of processes.

Token Ring algorithm

For this algorithm, we assume that there is a group of processes with no inherent ordering of processes, but that some ordering can be imposed on the group. For example, we can identify each process by its machine address and process ID to obtain an ordering. Using this imposed ordering, a logical ring is constructed in software. Each process is assigned a position in the ring and each process must know who is next to it in the ring (Figure 2). Here is how the algorithm works:

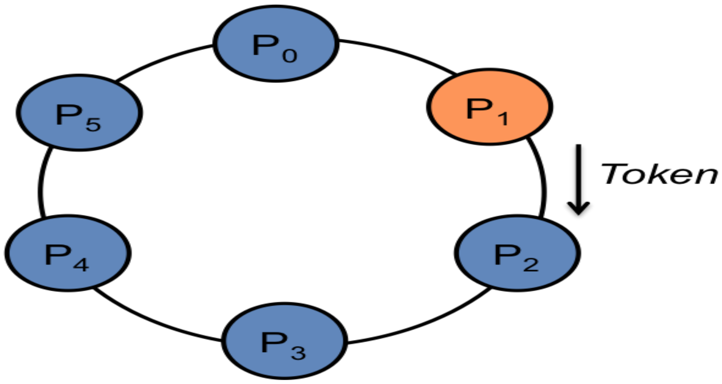


Figure 2. Token Ring algorithm

1. The ring is initialized by giving a token to process 0. The token circulates around the ring: process *n* passes it to process *(n+1) mod ringsize*.

2. When a process acquires the token, it checks to see if it is is waiting to use the resource. If so, it uses it and does its work. On exit, it passes the token to its neighboring process. In Figure 2, Process 1 had the token, completed its access to the resource, and is sending the token to its neighbor, process 2.

3. If a process is not interested in grabbing the lock on the resource, it simply passes the token along to its neighbor.

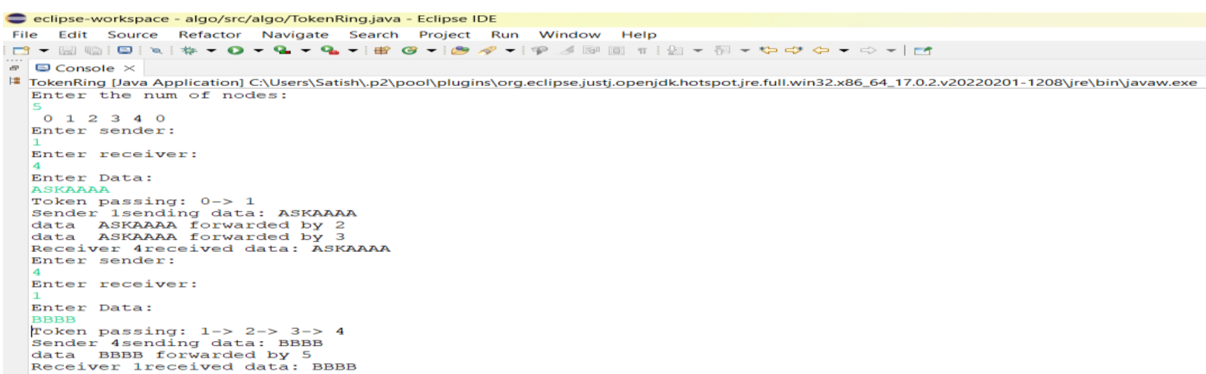
Only one process has the token, and hence the lock on the resource, at a time. Therefore, mutual exclusion is guaranteed. Order is also well-defined, so starvation cannot occur. The biggest drawback of this algorithm is that if a token is lost, it will have to be generated. Determining that a token is lost can be difficult.

```java
// Implementation of Token Ring Algorithm
import java.io.*;
import java.lang.*;

class Token Ring {
 public static void main(String args[]) throws Throwable {
  DataInputStream dis = new DataInputStream(System.in);
  System.out.println("Enter the num of nodes:");
  int n = Integer.parseInt(dis.readLine());
  // Decides the number of nodes forming the ring
  int token = 0;
  int ch = 1;
  for (int i = 0; i < n; i++)
   System.out.print(" " + i);
  System.out.println(" " + 0);
  try {
   while (ch == 1) {
    System.out.println("Enter sender:");
    int s = Integer.parseInt(dis.readLine());
    System.out.println("Enter receiver:");
    int r = Integer.parseInt(dis.readLine());
    System.out.println("Enter Data:");
    String d = dis.readLine();
    System.out.print("Token passing:");
    for (int i = token; i != s; i++)
```

.

```java
     System.out.print(" " + i + "->");
    System.out.println(" " + s);
    System.out.println("Sender " + s + "sending data: " + d);
    for (int i = s + 1; i != r; i = (i + 1) % n)
     System.out.println("data  " + d + " forwarded by " + i);
    System.out.println("Receiver " + r + "received data: " + d);
    token = s;
   }
  } catch (Exception e) {
  }
 }
}
```

Compiling and Executing Program:



Conclusion:

In this way Token Ring algorithm has been implemented. This algorithm is easy to implement and verify but the biggest drawback of this algorithm is that if a token is lost, it will have to be generated. Determining that a token is lost can be difficult.

ASSIGNMENT NO.6

Problem Statement:

Implement Bully and Ring algorithm for leader election.

Tools / Environment:

Java Programming Environment, JDK 1.8, Eclipse IDE.

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 processnumber.

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, itinitiates 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.
4.

B. When a process gets an ELECTION message from one of its lower-numberedcolleagues:

1. Receiver sends an OK message back to the sender to indicate that he is alive and will takeover.
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 takeover 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 (containingits own process number)

Leader Election:

3. Initiator sends the message to it's successor (if successor is down, sender skips over it and goesto the next member along the ring, or the one after that, until a running process is located).

4. At each step, sender adds its own process number to the list in the message.
5. When the message gets back to the process that started it all: 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

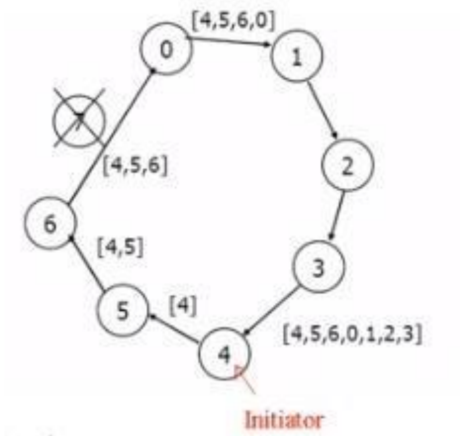
Algorithm

Initiation:

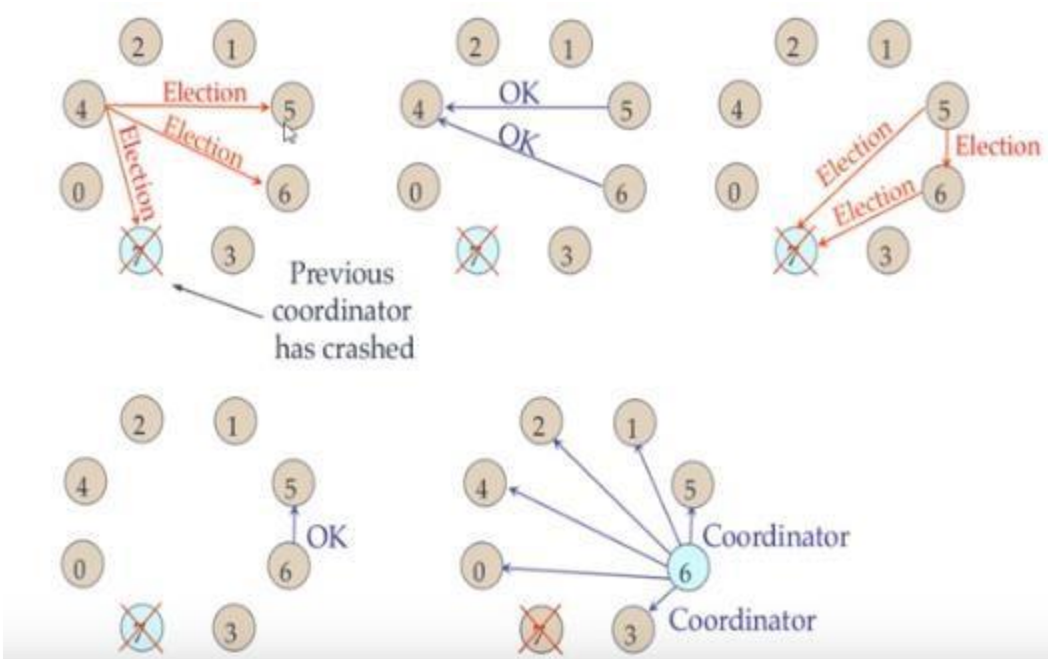
1. Consider the Process 4 understands that Process 7 is not responding.
2.Process 4 initiates the Election by sending "ELECTION" message to it's successor (or nextalive process) with it's ID.

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 processwith highest process ID is 6. The initiator will announce that Process 6 is Coordinator.



A. 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".

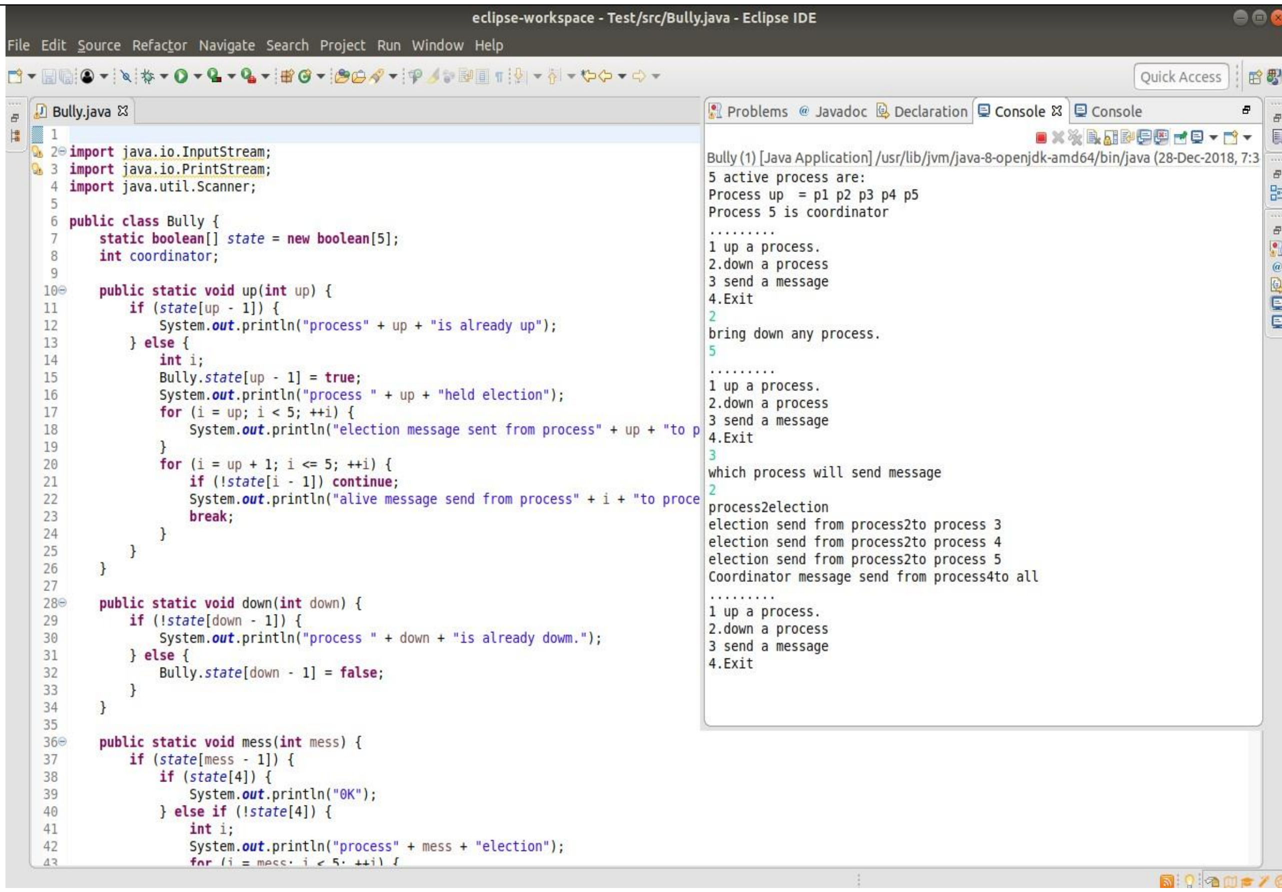
6. Ask for menu 1.Election 2.Exit

7. Ask for initializing election process.

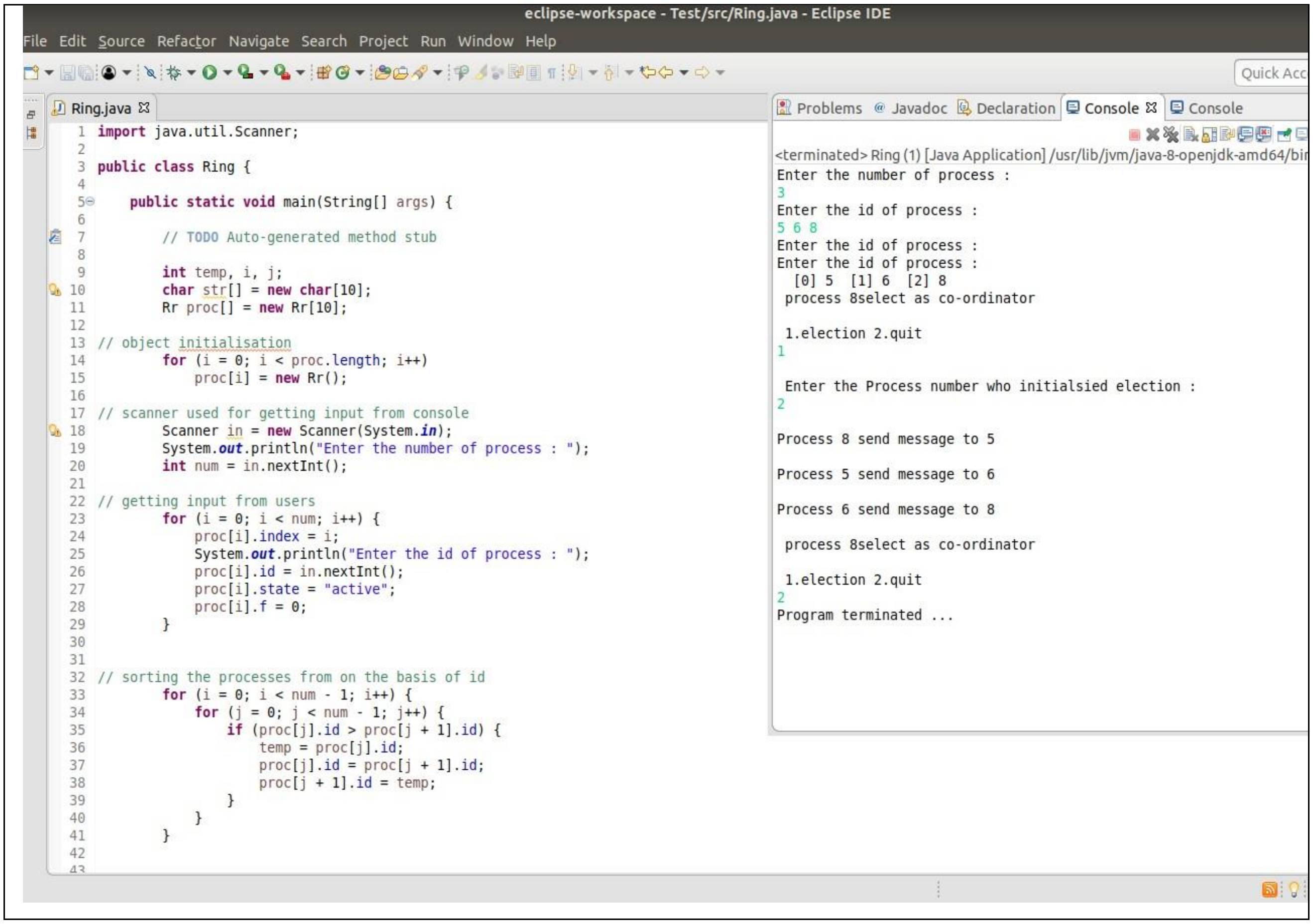
8. These inputs will be used by Ring Algorithm.

Writing the source code:

Bully.java



Ring.java



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 .
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. 7

Problem Statement:

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:

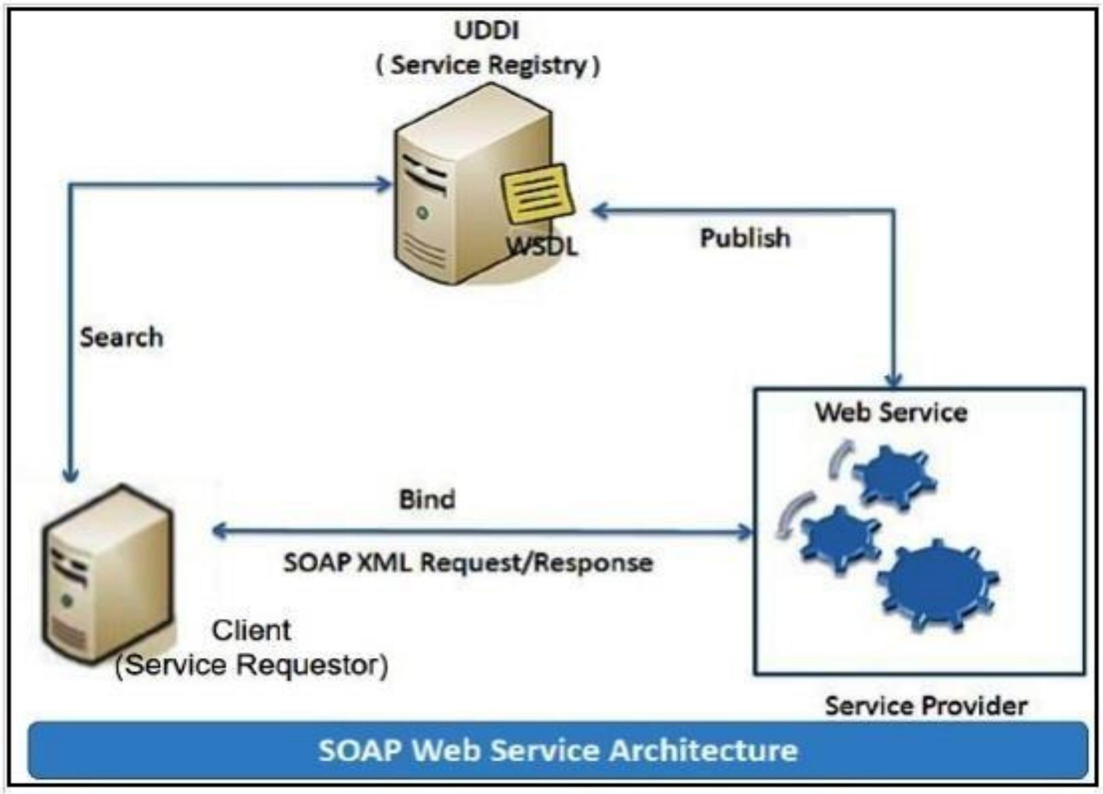
1. The client program that wants to interact with another application prepares its request content as a SOAP message.
2. Then, the client program sends this SOAP message to the server web service as an HTTP POST request with the content passed as the body of the request.
3. The web service plays a crucial 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:

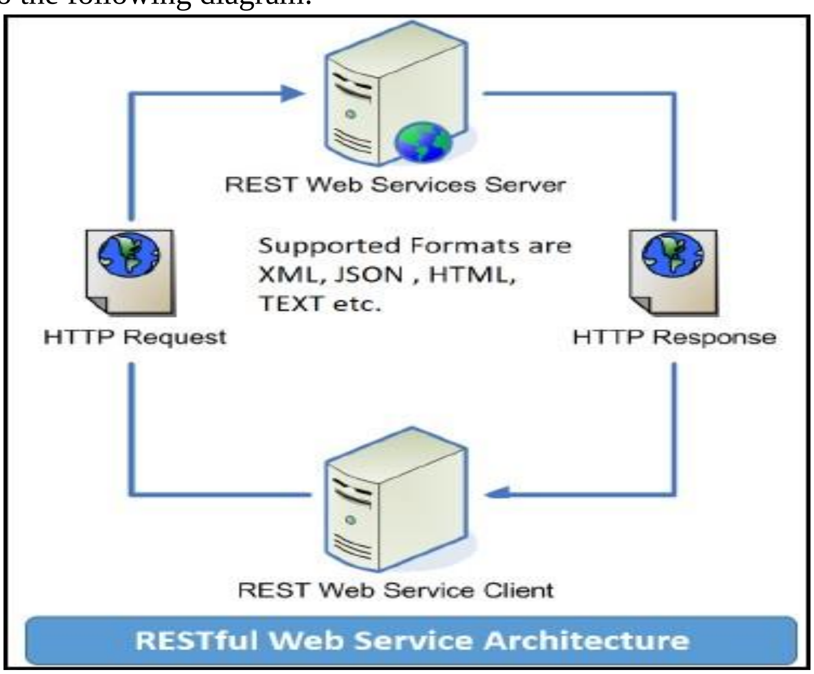
Universal Description, Discovery, and Integration (**UDDI**)*:* UDDI is an XMLbased framework for describing, discovering, and integrating web services. It acts as a directory of webservice interfaces described in the WSDL language.

Web Services Description Language (**WSDL**)*:* WSDL is an XML document 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's a protocol	REST can use SOAP web services
SOAP uses services interface to expose its business logic	REST uses URI (a kind of URL)to expose business logic
JAX –WS (security system) is the JAVA API for SOAP web services	JAX-RS (security system) is the JAVA API for RESTFUL web services
SOAP defines standards to be followed strictly	REST do not define to much standard like SOAP
SOAP requires more bandwidth and resources than REST	REST requires less bandwidth and resources
SOAP defines its own security	RESTFUL web services inherit security measures from the underlying transport means uses security of the network
SOAP permits XML data format only	REST permits different data formats 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 CalculatorWSand create New Web Service.
- IDE starts the glassfish server, builds the application and deploys the application on server.

2. Consuming the Webservice:

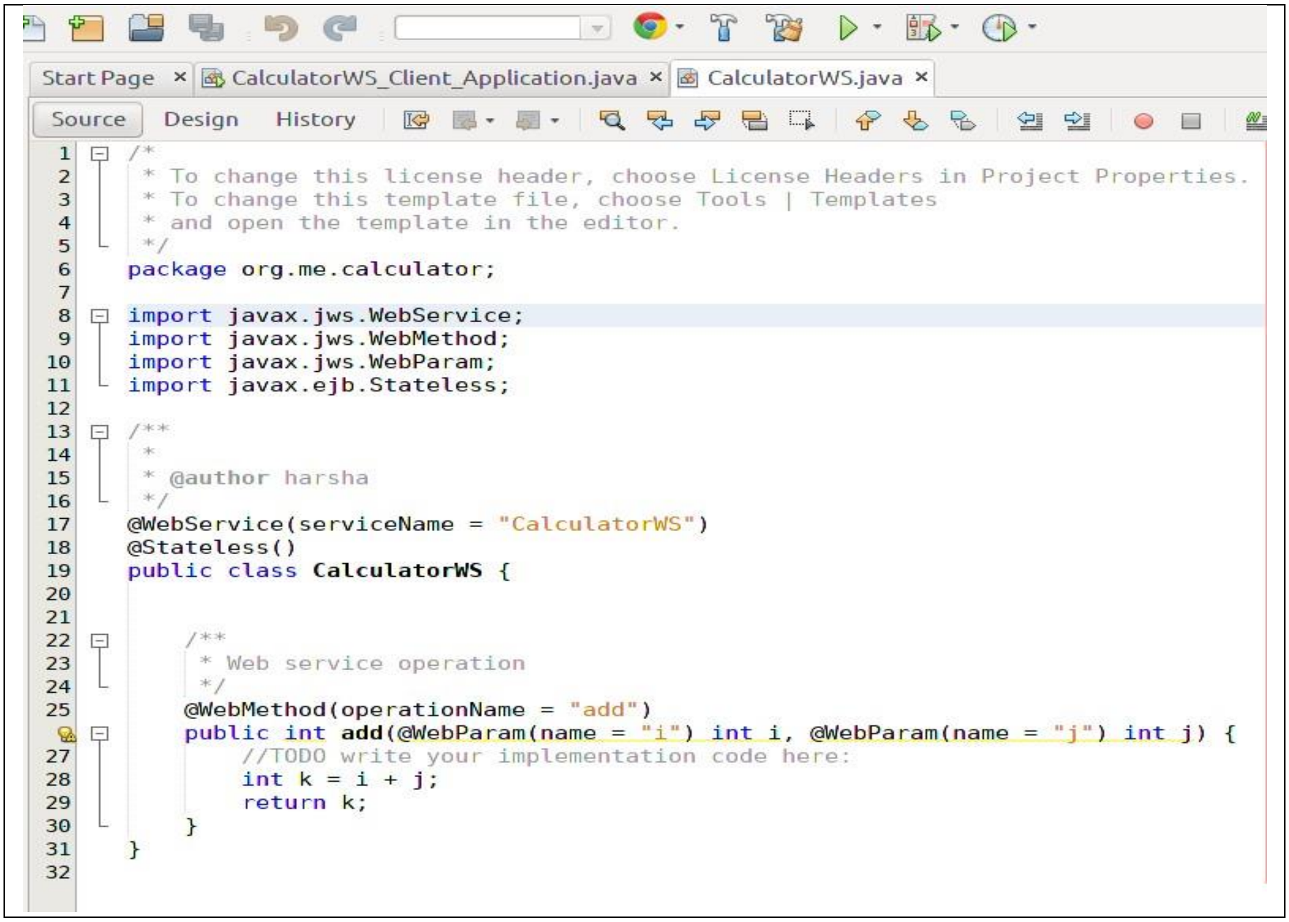
- Create a project with an CalculatorClient

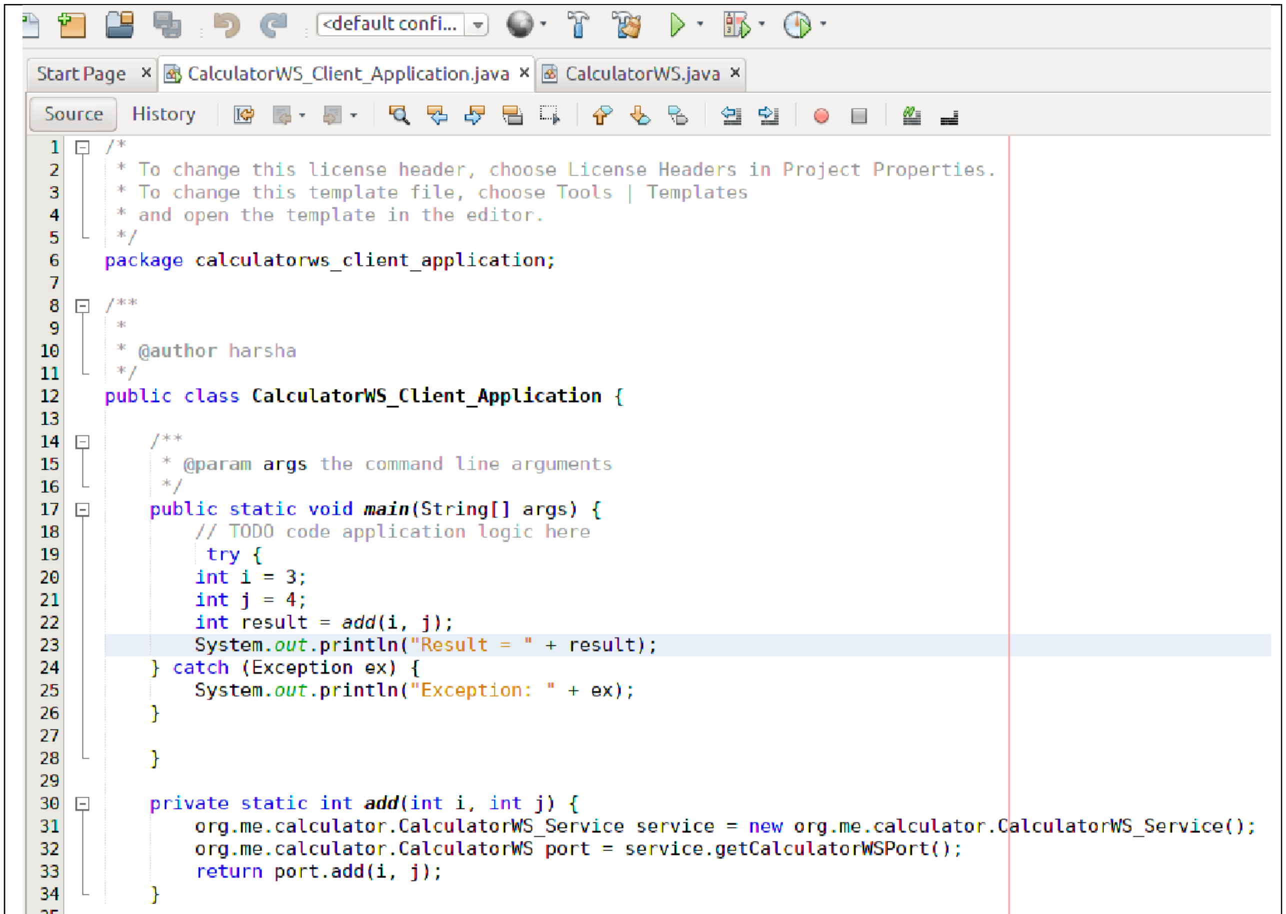
- Create package org.calculator.client;
- add java class CalculatorWS.java, addresponse.java, add.java, CalculatorWSService.java and ObjectFactory.java

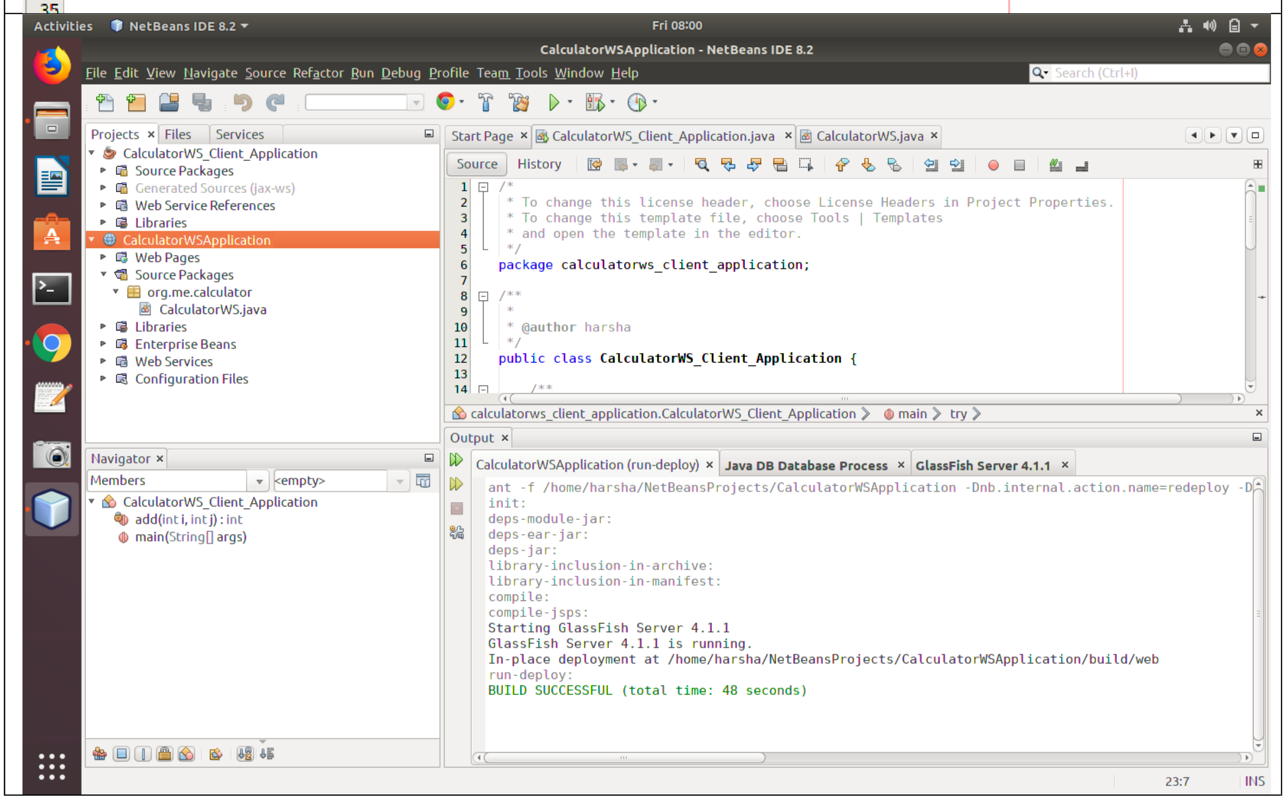
3. Creating servlet in web application

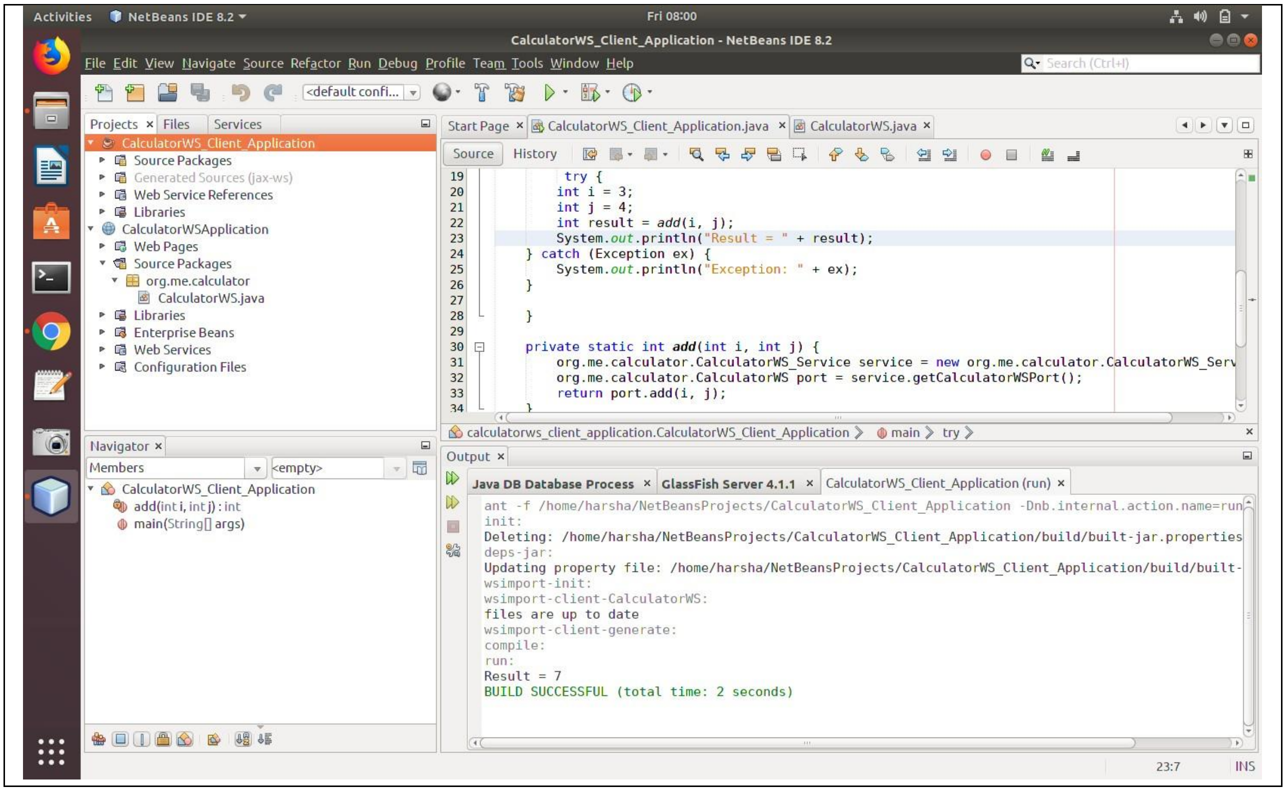
- Create new jsp page for creating user interface.

Writing the source code:



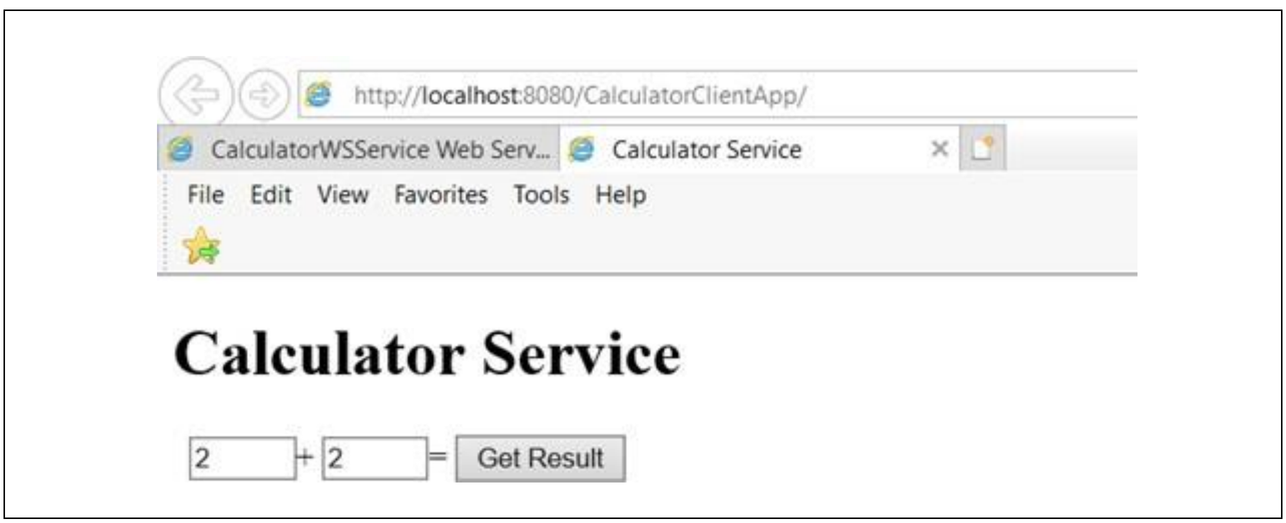

Start Page
CalculatorWS_Client_Application.java
CalculatorWS.java
Source
History
/*
* To change this license header, choose License Headers in Project Properties.
* To change this template file, choose Tools | Templates
* and open the template in the editor.
*/
package calculatorws_client_application;
/**
*
* @author harsha
*/
public class CalculatorWS_Client_Application {
/**
* @param args the command line arguments
*/
public static void main(String[] args) {
// TODO code application logic here
try {
int i = 3;
int j = 4;
int result = add(i, j);
System.out.println("Result = " + result);
} catch (Exception ex) {
System.out.println("Exception: " + ex);
}
}
private static int add(int i, int j) {
org.me.calculator.CalculatorWS_Service service = new org.me.calculator.CalculatorWS_Service();
org.me.calculator.CalculatorWS port = service.getCalculatorWSPort();
return port.add(i, j);
}

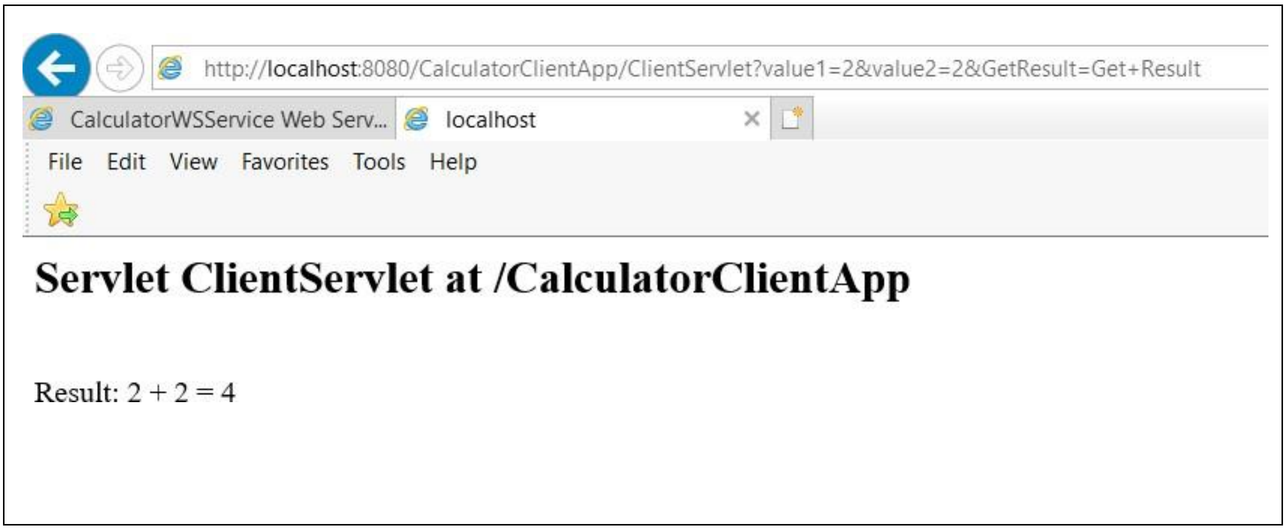

Activities
NetBeans IDE 8.2
Fri 08:00
CalculatorWSApplication - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Projects Files Services
CalculatorWS_Client_Application
Source Packages
Generated Sources (jax-ws)
Web Service References
Libraries
CalculatorWSApplication
Web Pages
Source Packages
org.me.calculator
CalculatorWS.java
Libraries
Enterprise Beans
Web Services
Configuration Files
Navigator
Members
<empty>
CalculatorWS_Client_Application
add(int i, int j) : int
main(String[] args)
Output
CalculatorWSApplication (run-deploy)
Java DB Database Process
GlassFish Server 4.1.1
ant -f /home/harsha/NetBeansProjects/CalculatorWSApplication -Dnb.internal.action.name=redeploy -D
init:
deps-module-jar:
deps-ear-jar:
deps-jar:
library-inclusion-in-archive:
library-inclusion-in-manifest:
compile:
compile-jsps:
Starting GlassFish Server 4.1.1
GlassFish Server 4.1.1 is running.
In-place deployment at /home/harsha/NetBeansProjects/CalculatorWSApplication/build/web
run-deploy:
BUILD SUCCESSFUL (total time: 48 seconds)
23:7
INS



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.8

Problem Statement:

Mini Project: A Distributed Application for Interactive Multiplayer Games

Theory:

As the gaming industry continues to grow in size, the need to create a unique and dynamic userexperience has become even more mandatory. Because of its fandom, businesses have to maximize the multiplayer gaming experience to drive customer acquisition and retention. However, companies are faced with a number of obstacles when trying to scale multiplayer games, all of which can be solved through Redis.

Personalized interactions and high-speed reactions are at the heart of creating a unique userexperience. Redis provides game publishers with a powerful database that can support low-latency gaming use cases. Recently a Launchpad App built a unique application that could have only beendeployed by Redis due to its unmatched speed in transmitting data.

This is crucial because participants play from all around the world. Interactions between players must be executed in real-time to support gameplay, requiring the latency to be less than onemillisecond.

- Students can form a group of 3 to 4 students and can develop the Interactive MultiplayerGame.
- Students must use Open-Source Operating Systems, tools, Multi-core CPU supportingVirtualization and Multi-Threading.

Students can refer the following links for project:-

- https://redis.com/blog/how-to-build-a-real-time-geo-distributed-multiplayer-top-down-arcade-shooting-game-using-redis/
- https://www.smashingmagazine.com/2021/10/real-time-multi-user-game/
- https://ably.com/blog/peer-to-peer-game-with-pub-sub