Laboratory Manual

Computer Laboratory-V
(2019 Course)

Final Year - Information Technology

Teaching Scheme	Examination Scheme
Theory : ——	Term Work: 50 Marks
Practical : 04 Hrs/Week	Practical : 25 Marks
	Oral : ——

Prepared By

Mr.Laxman.J.Deokate

Department of Information Technology

Sinhgad Academy of Engineering, Kondhwa(bk)
Pune – 411048

Savitribai Phule Pune University, Pune

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 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. Candidate is expected to know the theory involved in the experiment.
3. The practical examination should be conducted if the journal of the Student is completed in all respects and certified by concerned faculty and head of the department.

Guidelines for Lab /TW Assessment
The term work based on performance of students considering the parameters such as
1)timely conduction of practical assignment,
2)methodology adopted for implementation of practical assignment,
3)timely submission of assignment in the form of handwritten/printed write-up along with results of implemented assignment,
5)attendance etc.
6)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
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.

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.

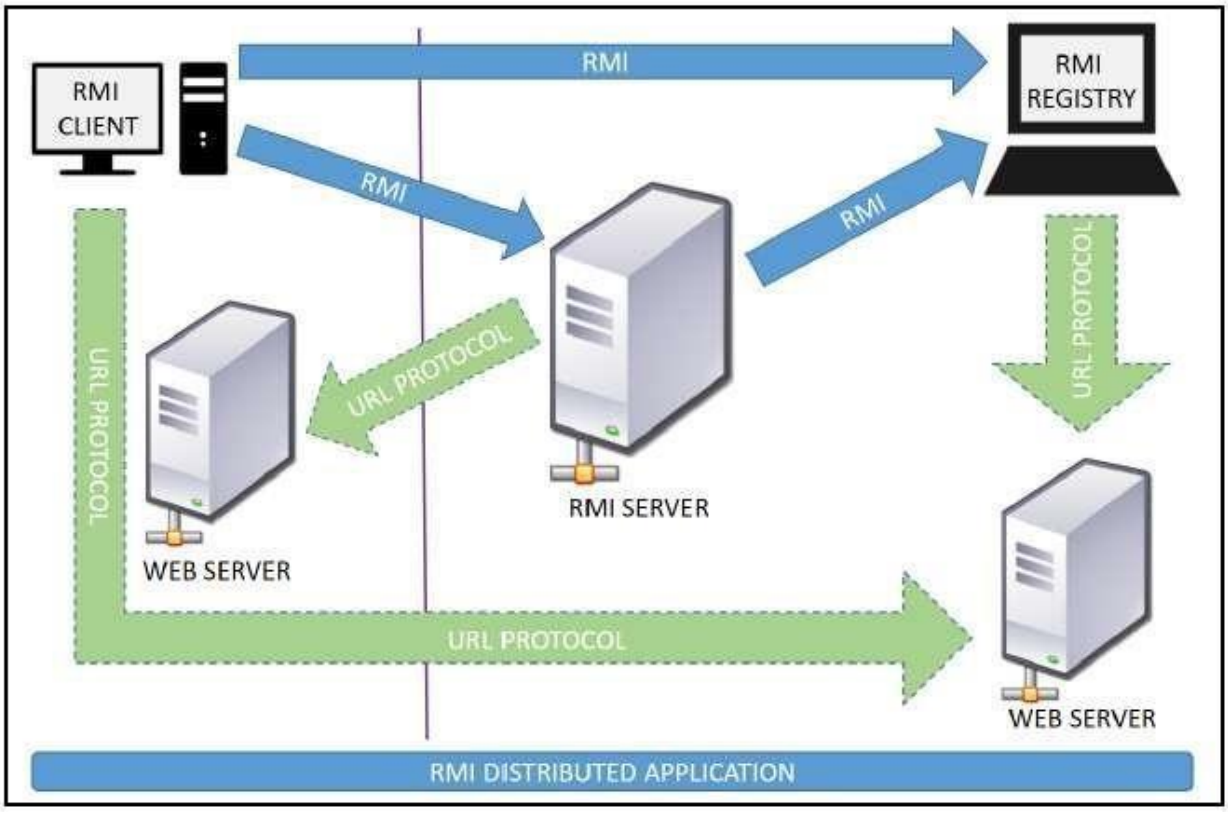
Tools / Environment:

Java Programming Environment, jdk 1.8, rmiregistry

Related Theory:

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

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



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

Key terminologies of RMI:

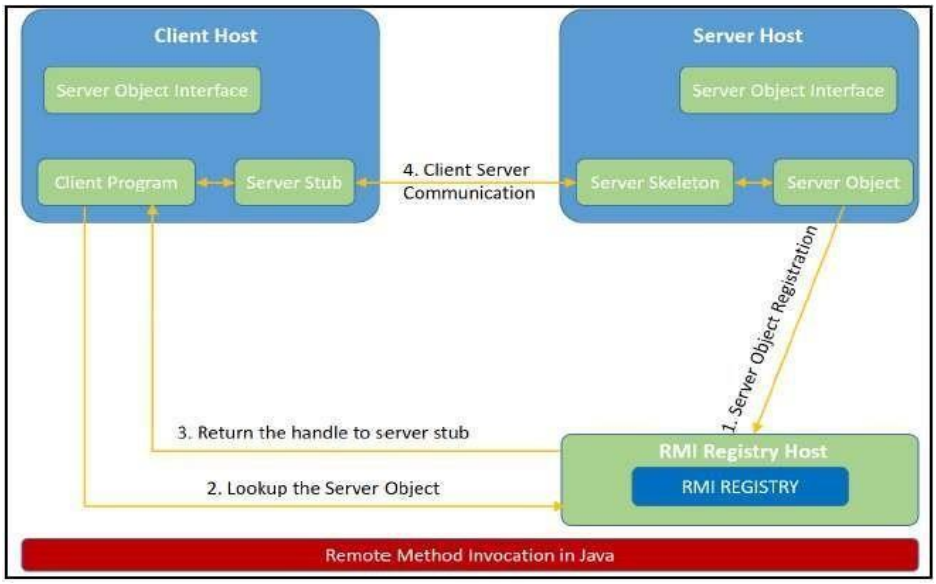
The following are some of the important terminologies used in a Remote Method Invocation. **Remote object**: This is an object in a specific JVM whose methods are exposed so they could be invoked by another program deployed on a different JVM.
Remote interface: This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.
RMI: This is a way of invoking a remote object's methods with the help of a remote interface. It can be carried with a syntax that is similar to the local method invocation.
Stub: This is a Java object that acts as an entry point for the client object to route any outgoing requests. It exists on the client JVM and represents the handle to the remote object.
If any object invokes a method on the stub object, the stub establishes RMI by following these steps:

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

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

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

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



Designing the solution:

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

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

Remote interface definition: The purpose of defining a remote interface is to declare the methods that should be available for invocation by a remote client.
Programming the interface instead of programming the component implementation is an essential design principle adopted by all modern Java frameworks, including Spring. In the same pattern, the definition of a remote interface takes importance in RMI design as well.
2. **Remote object implementation**: Java allows a class to implement more than one interface at a time. This helps remote objects implement one or more remote interfaces. The remote object class may have to implement other local interfaces and methods that it is responsible for. Avoid adding complexity to this scenario, in terms of how the arguments or return parameter values of such component methods should be written.
3. **Remote client implementation**: Client objects that interact with remote server objects can be written once the remote interfaces are carefully defined even after the remote objects are deployed.

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

Implementation:

Consider building an application to perform diverse mathematical operations.

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

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

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

The second source file, **`AddServerImpl.java`**, implements the remote interface. The implementation of the **`add()`** method is straightforward. All remote objects must extend

UnicastRemoteObject, which provides functionality that is needed to make objects available from remote machines.

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

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

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

2. Generate a stub

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

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

```
rmic AddServerImpl.
```

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

3. Install Files on the Client and Server Machines

Copy **AddClient.class, AddServerImpl_Stub.class**, **AddServerIntf.class**

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

4. Start the RMI Registry on the Server Machine

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

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

AddServerImpl.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

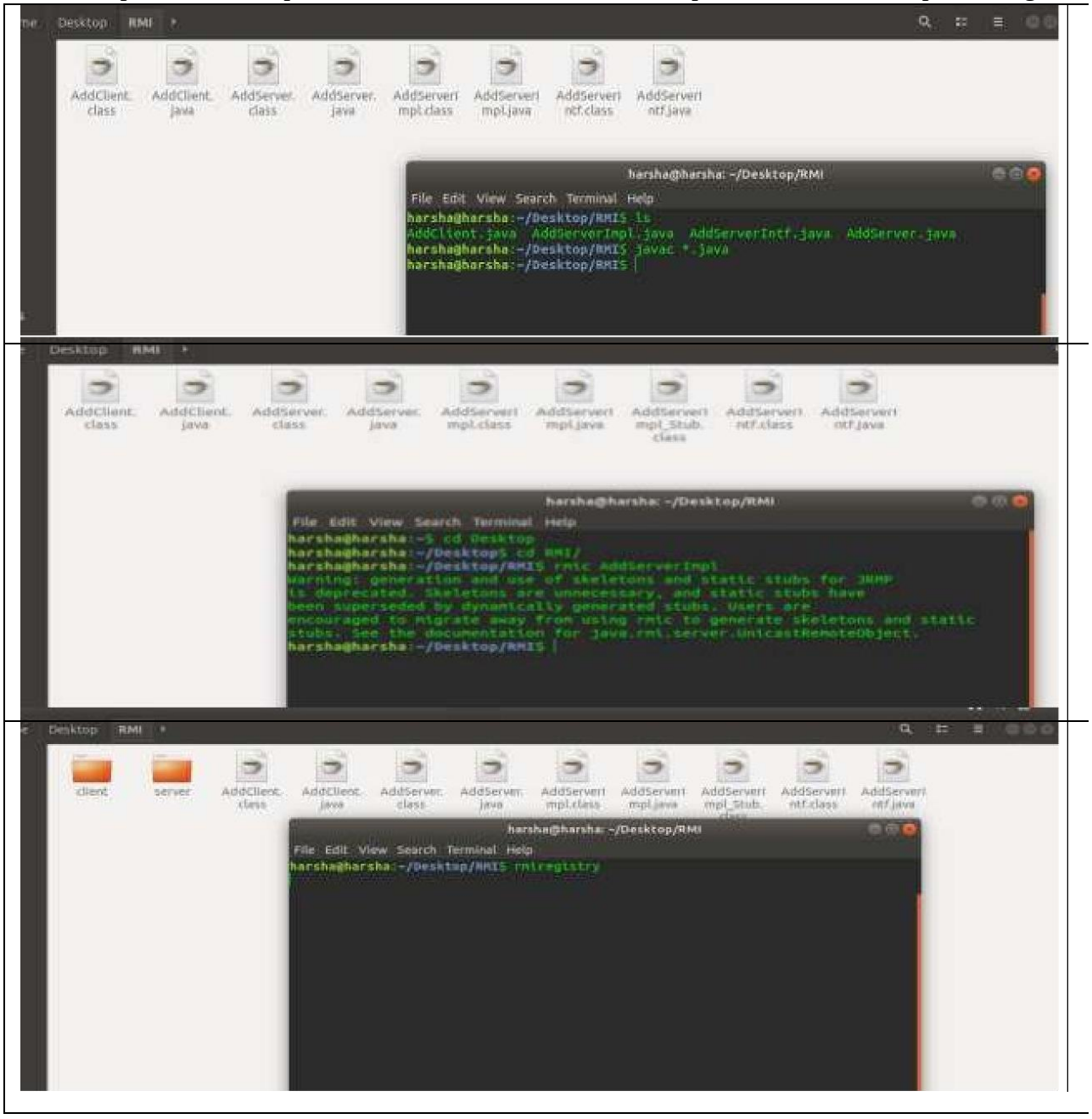
```
import java.net.*;
import java.rmi.*;
public class AddServer {
public static void main(String args[]) {
try {
AddServerImpl addServerImpl = new AddServerImpl();
Naming.rebind("AddServer", addServerImpl);
}
catch(Exception e) {
System.out.println("Exception: " + e);
}
}
}
```

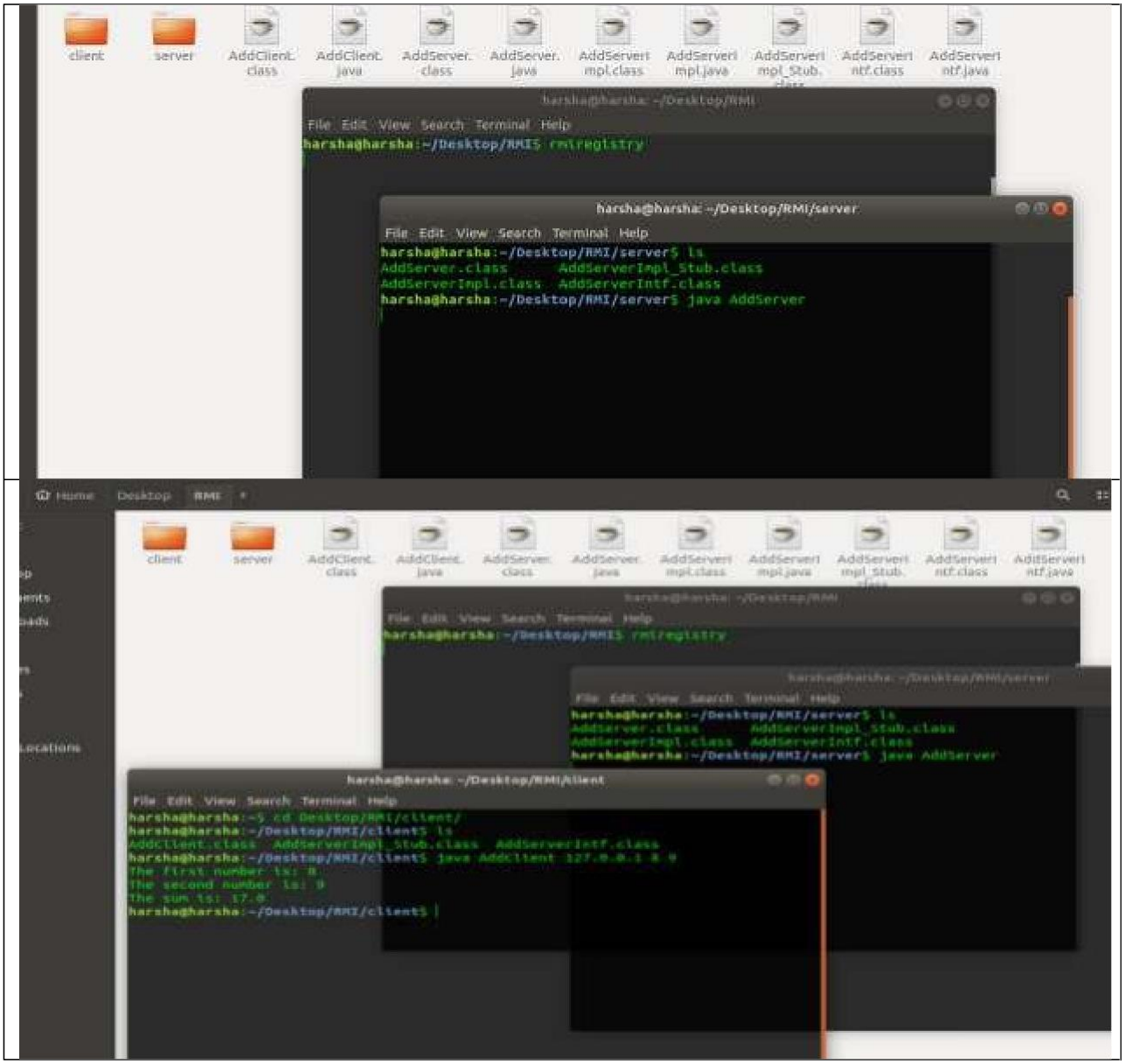
AddClient.java

```
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:





server commands:

Connect computer to network
1. Sudo su: It will ask admin password= @@dmin123
2. sudo apt-get update
3. sudo apt-get install openjdk-17-jdk
After installation of jdk execute all commands sequentially as given below

To change from admin to root use command **sudo su** on terminal

base) comp@SL15:~$ javac AddServerIntf.java
(base) comp@SL15:~$ javac AddServerImpl.java
(base) comp@SL15:~$ javac AddServer.java

root@SL17:/home/admin1# rmic AddServerImpl
Command 'rmic' not found, but can be installed with:

Above message can be displayed when you compile rmic compiler.

By using follwing Commands, one can install rmic compiler in machine

apt install openjdk-11-jdk-headless # version 11.0.20.1+1-0ubuntu1~22.04, or
apt install openjdk-8-jdk-headless   # version 8u382-ga-1~22.04.1

(base) comp@SL15:~$ rmic AddServerImpl //for generation of stub and skeleton
Warning: generation and use of skeletons and static stubs for JRMP
is deprecated. Skeletons are unnecessary, and static stubs have
been superseded by dynamically generated stubs. Users are
encouraged to migrate away from using rmic to generate skeletons and static
stubs. See the documentation for java.rmi.server.UnicastRemoteObject.
(base) comp@SL15:~$
(base) comp@SL15:~$ rmiregistry 1099 &
[1] 13965

Exit from the root by using ctrl+d or exit command

(base) comp@SL15:~$ java AddServer

Blinks cursor i.e. server is ready to accept request

client commands:

(base) comp@SL15:~$ javac AddClient.java
(base) comp@SL15:~$ java AddClient 6 7
The first number is: 7
Exception: java.lang.ArrayIndexOutOfBoundsException: 2
(base) comp@SL15:~$ javac AddClient.java
(base) comp@SL15:~$ java AddClient 6 4
The first number is: 6
The second number is:4
The sum is:10.0
(base) comp@SL15:~$

AddServerIntf.java

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

```
}
```

AddServerImpl.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

```
import java.net.*;
import java.rmi.*;
public class AddServer {
public static void main(String args[]) throws ConnectIOException
 {
try {
AddServerImpl addServerImpl = new AddServerImpl();

Naming.rebind("AddServer", addServerImpl);
}
catch(Exception e) {
System.out.println("Exception: " + e);
}
}
}
```

Need to connect to client without internet for that one can refer following server steps as a reference

```
import  java.rmi.*;
import  java.rmi.NotBoundException;
import  java.rmi.RemoteException;
import  java.rmi.registry.LocateRegistry;
import  java.rmi.registry.Registry;
import  java.io.IOException;
import  java.rmi.server.UnicastRemoteObject;
import  java.io.*;
import java.rmi.*;
public class AddServer {
public static void main(String args[]) throws ConnectIOException
{
try {
AddServerImpl addServerImpl = new AddServerImpl();//create object of add method resides in addServerImpl
Naming.rebind("AddServer", addServerImpl);
}
catch(Exception e) {
```

```
System.out.println("Exception: " + e);
}
}
}
```

AddClient.java

```
import java.rmi.*;
public class AddClient {
public static void main(String args[])throws ConnectIOException
 {
try {
String addServerURL = "rmi://" + "172.16.236.15" + "/AddServer";   //enter machines IP address in server's URL
AddServerIntf addServerIntf =(AddServerIntf)Naming.lookup(addServerURL);
System.out.println("The first number is: " + args[0]);
double d1 = Double.valueOf(args[0]).doubleValue();
System.out.println("The second number is:"+ args[1]);
double d2 = Double.valueOf(args[1]).doubleValue();
System.out.println("The sum is:"+ addServerIntf.add(d1, d2));
}
catch(Exception e)
{
System.out.println("Exception: " + e);
}
}
}
```

Need to add some header files in client program to run client and to connect Server without Internet. Can use following steps for reference

```
import java.rmi.*;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class AddClient {
public static void main(String args[])throws ConnectIOException
{
try {
//String addServerURL = "rmi://" + "172.16.7.170" + "/AddServer"; //
String addServerURL = "rmi://" + "localhost" + "/AddServer"; //
AddServerIntf addServerIntf =(AddServerIntf)Naming.lookup(addServerURL);
System.out.println("The first number is: " + args[0]);
double d1 = Double.valueOf(args[0]).doubleValue();
System.out.println("The second number is:"+ args[1]);
double d2 = Double.valueOf(args[1]).doubleValue();
System.out.println("The sum is:"+ addServerIntf.add(d1, d2));
}
catch(Exception e)
```

```
{
System.out.println("Exception: " + e);
}
}
}
```

Conclusion:

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

ASSIGNMENT NO. 2

Problem Statement:

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

Tools / Environment:

Java Programming Environment, JDK 1.8

Related Theory:

Common Object Request Broker Architecture (CORBA):

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

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

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

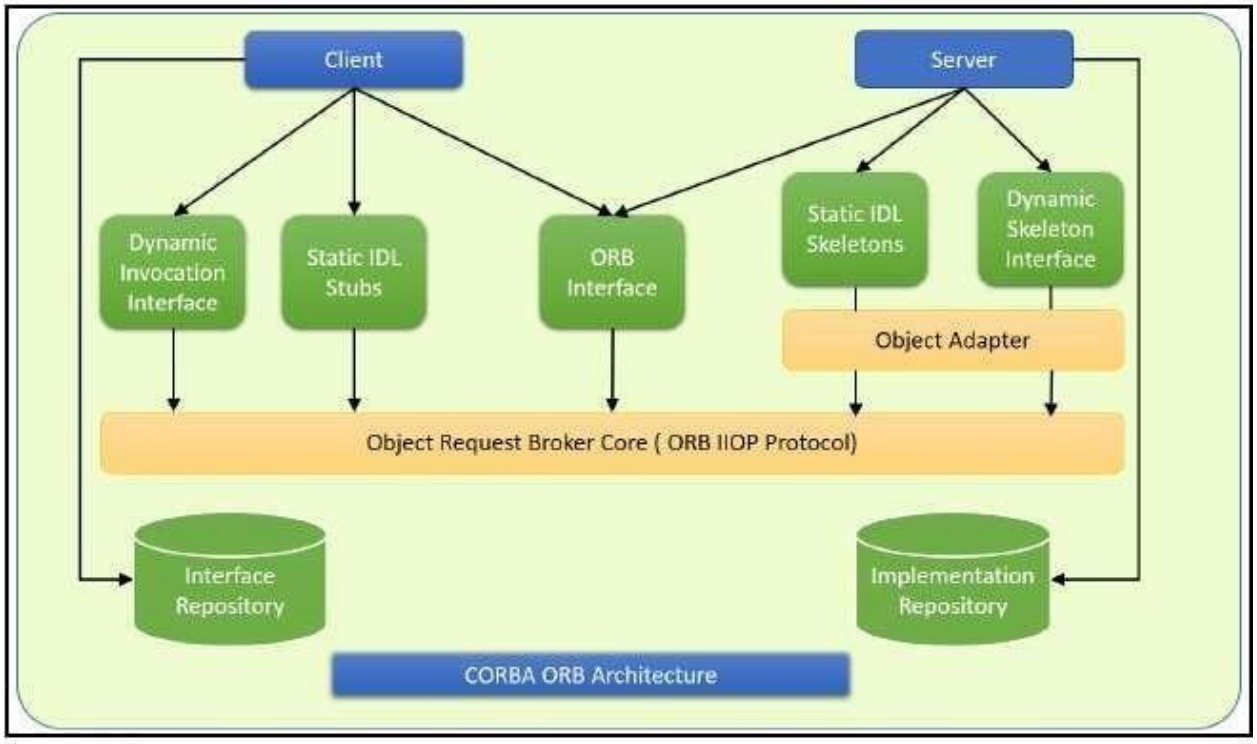
Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR&Benefits" maintain an object model with details of the organization, employees with demographic information, benefits, payroll, and deductions. They are only different in the way they handle the details, based on the country and region they are operating for. For each object type, similar to the HR&Benefits systems, we can define an interface using the **Interface Definition Language** (**OMG IDL**).

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

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

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

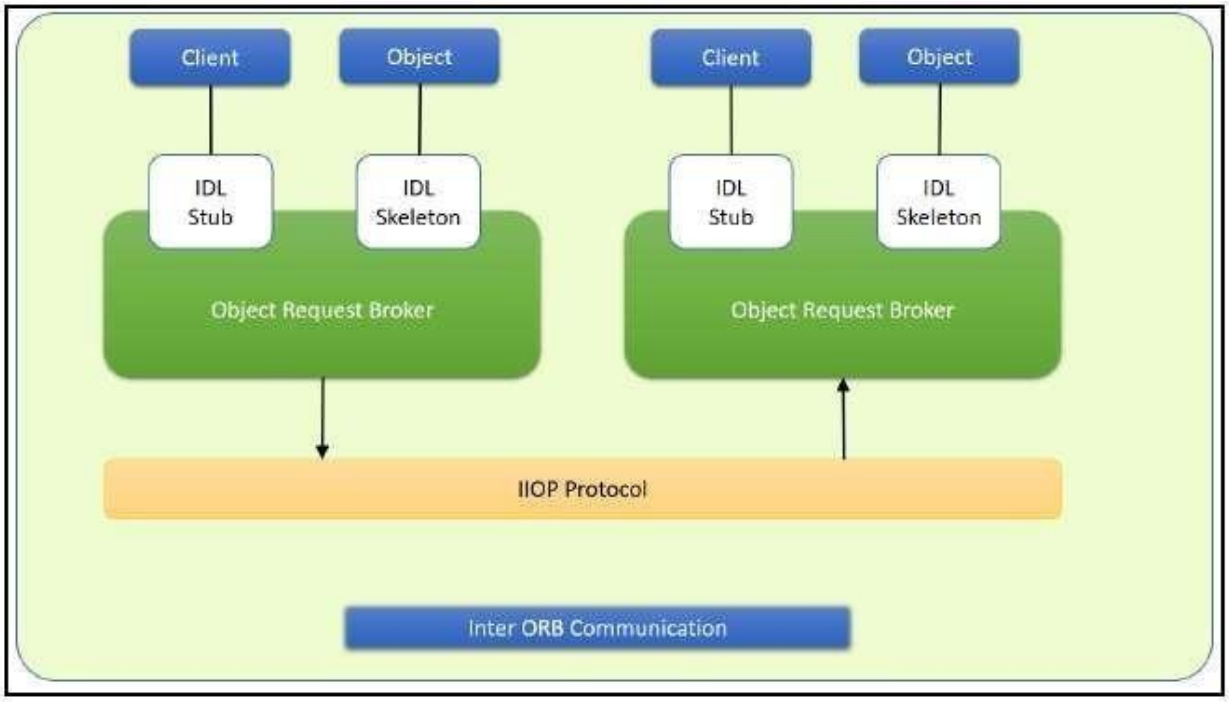
The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons, The objects are written (on the right) and a client for it (on the left), as represented in the diagram. The client and server use stubs and skeletons as proxies, respectively.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



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

Inter-ORB communication

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



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

Java Support for CORBA

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

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

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

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

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

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

The IDL Programming Model:

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

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

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

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

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

Designing the solution:

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

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

1. Creating CORBA Objects using Java IDL:

In order to distribute a Java object over the network using CORBA, one has to define 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.

Modules

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

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

Interfaces

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

```
interface  PrintServer  :  Server  {
...
```

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

Data members and methods

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

```
readonly attribute string myString;
```

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

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

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

A complete IDL example

Now let's tie all these basic elements together. Here's a complete IDL example that declares a module within another module, which itself contains several interfaces:

```
module OS  {
module services {
interface Server {
readonly      attribute      string
serverName;
boolean init(in string sName);
};

interface Printable {
boolean print(in string header);
};

interface PrintServer : Server {
boolean printThis(in Printable p);
          };
     };
};
```

The first interface, `Server`, has a single read-only `string`attribute and an `init()` method

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

2. Turning IDL Into Java

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

A Java interface with the same name as the IDL interface. This can act as the basis for a Java implementation of the interface (but you have to write it, since IDL doesn't provide any details about method implementations).
A *helper* class whose name is the name of the IDL interface with "Helper" appended to it (e.g., `ServerHelper`). The primary purpose of this class is to provide a static `narrow()` method that can safely cast CORBA `Object`references to the Java interface type. The helper class also provides other useful static methods, such as `read()` and `write()` methods that allow you to read and write an object of the corresponding type using I/O streams.
A *holder* class whose name is the name of the IDL interface with "Holder" appended to it (e.g., `ServerHolder`). This class is used when objects with this interface are used as `out`or `inout`arguments in remote CORBA methods. Instead of being passed directly into the remote method, the object is wrapped with its holder before being passed. When a remote method has parameters that are declared as `out`or `inout`, the method has to be able to update the argument it is passed and return the updated value. The only way to guarantee this, even for primitive Java data types, is to force `out`and `inout`arguments to be wrapped in Java holder classes, which are filled with the output value of the argument when the method returns.

The `idltoj` tool generate 2 other classes:

A client *stub* class, called *_interface-name*`Stub`, that acts as a client-side implementation of the interface and knows how to convert method requests into ORB requests that are forwarded to the actual remote object. The stub class for an interface named `Server`is called `_ServerStub`.
A server *skeleton* class, called *_interface-name*`ImplBase`, that is a base class for a server-side implementation of the interface. The base class can accept requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named `Server`is called `_ServerImplBase`.

So, in addition to generating a Java mapping of the IDL interface and some helper classes for the

Java interface, the *idltoj* compiler also creates subclasses that act as an interface between a CORBA client and the ORB and between the server-side implementation and the ORB.This creates the five Java classes: a Java version of the interface, a helper class, a holder class, a client stub, and a server skeleton.

3. Writing the Implementation

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

Implementing the solution:

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

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

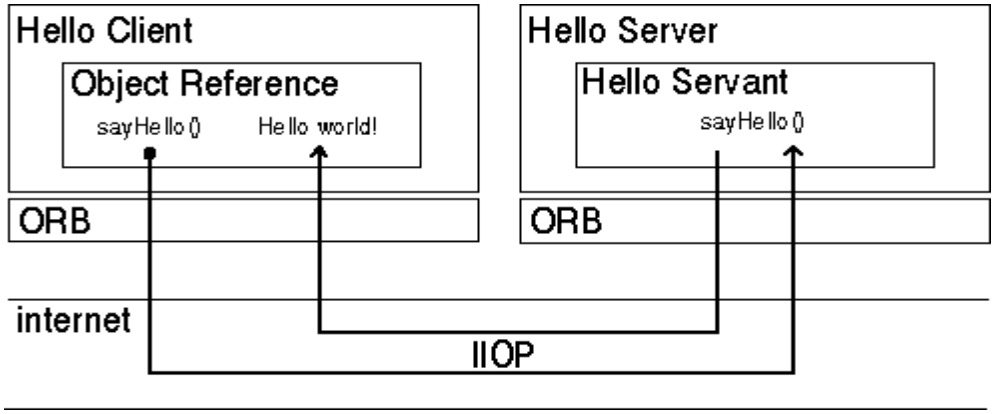
The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL).To complete the application, you simply provide the server **(`HelloServer.java`)** and client **(`HelloClient.java`)** implementations.

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

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

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

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



3. Implementing the Client Application (`HelloClient.java`)

The example application client that follows:

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

Writing the source code:

ReverseModule.idl
~/Desktop/IDL CORBA

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

ReverseClient.java
~/Desktop/IDL CORBA

```
// 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);
```

Open

ReverseImpl.java
~/Desktop/IDL CORBA

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

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

Open

ReverseServer.java
~/Desktop/IDL CORBA

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

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

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

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

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

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

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

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

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

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

//ReverseClient.java

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

class ReverseClient
{

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

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

            org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

            String name = "Reverse";
            //Helper class provides narrow method that cast corba object reference
(ref) into the java interface
           // System.out.println("Step2");
           // Look ups "Reverse" in the naming context
            ReverseImpl = ReverseHelper.narrow(ncRef.resolve_str(name));

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

            String tempStr= ReverseImpl.reverse_string(str);

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

//ReverseImpl.java

```
import ReverseModule.ReversePOA;
import java.lang.String;
class ReverseImpl extends ReversePOA
{
```

```
    ReverseImpl()
    {
        super();
        System.out.println("Reverse Object Created");
    }

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

//ReverseModule.idl

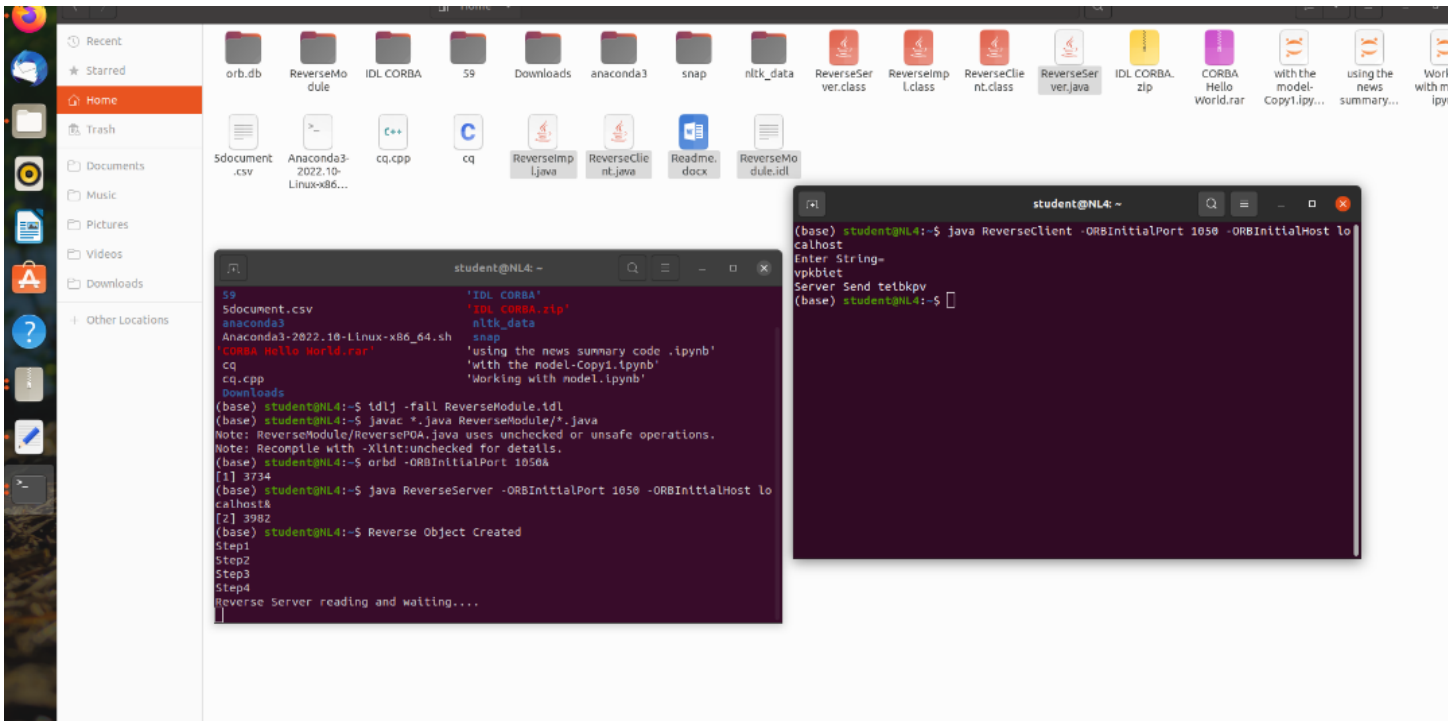
```
module ReverseModule   //module ReverseModule is the name of the module
{
    interface Reverse
    {
        string reverse_string(in string str);
    };
};
```

// ReverseServer.java

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

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

            // initialize the portable object adaptor (BOA/POA) connects client
request using object reference
            //uses orb method as resolve_initial_references
            POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootPOA.the_POAManager().activate();

            // creating an object of ReverseImpl class

            ReverseImpl rvr = new ReverseImpl();
            //server consist of 2 classes ,servent and server. The servent is the
subclass of ReversePOA which is generated by the idlj compiler
```

```
            // The  servent ReverseImpl is the implementation of the ReverseModule
idl interface
            // get the object reference from the servant class
            //use root POA class and its method servant_to_reference
            org.omg.CORBA.Object ref = rootPOA.servant_to_reference(rvr);

           // System.out.println("Step1");
            Reverse h_ref = ReverseModule.ReverseHelper.narrow(ref);// Helper class
provides narrow method that cast corba object reference (ref) into the java
interface
           // System.out.println("Step2");
           // orb layer uses resolve_initial_references method to take initial
reference as NameService
            org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
            //Register new object in the naming context under the Reverse

           // 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);
            //Server run and waits for invocations of the new object from the client

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

Result-



Building and Executing the solution:

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

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

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

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

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

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

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

option, are:

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

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

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

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

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

- `HelloOperations.java`

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

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

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

4. Start `orbd`.

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

```
orbd -ORBInitialPort 1050&
```

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

5. Start the `HelloServer`:

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

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

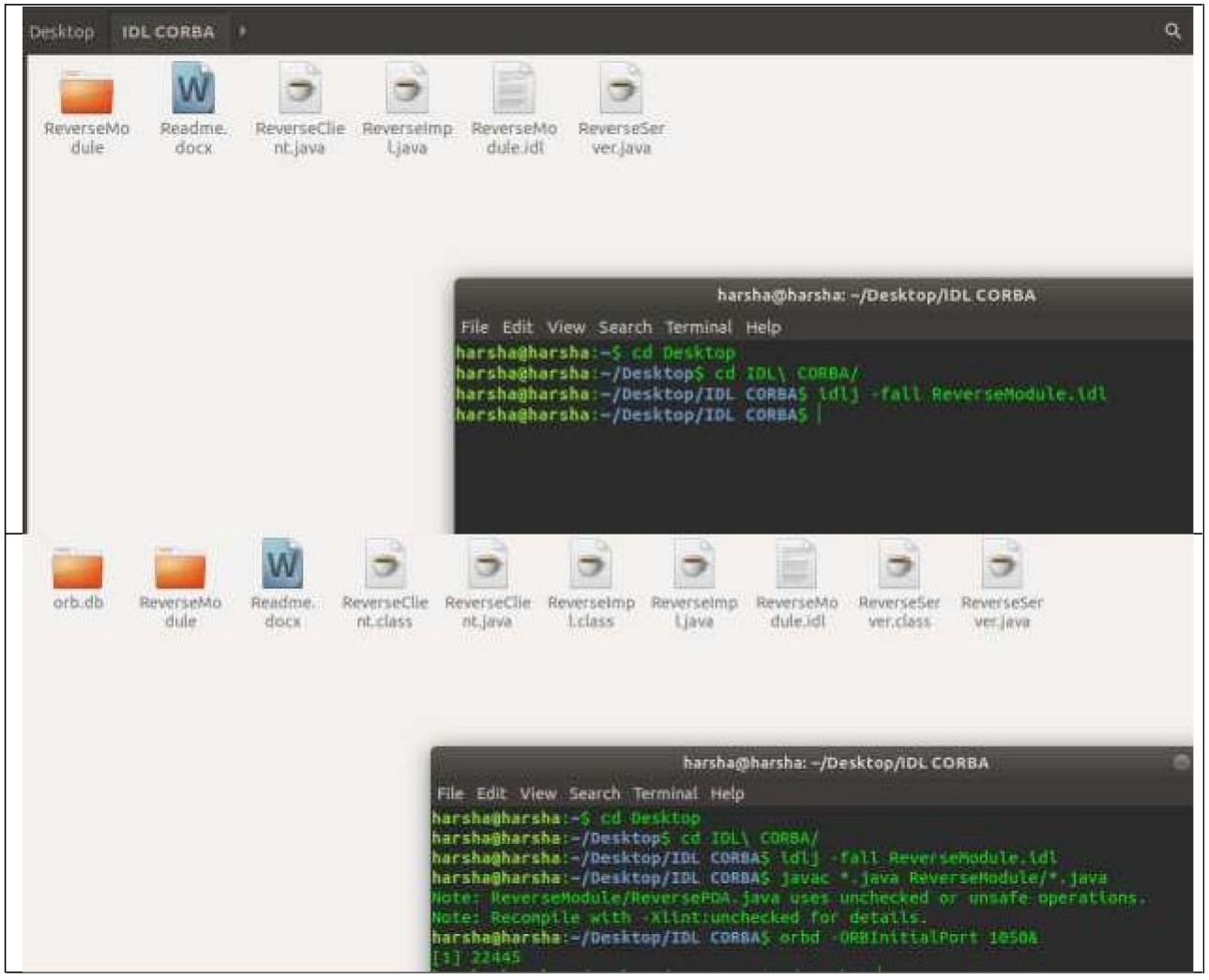
You will see `HelloServer`ready and waiting... when the server is started.

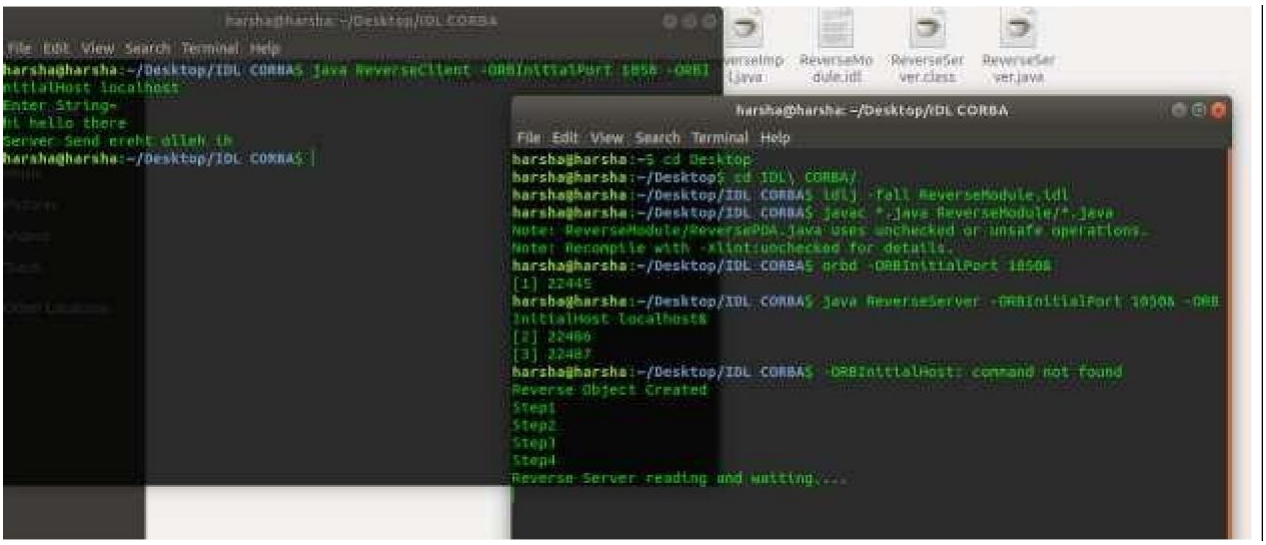
6. Run the client application:

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

When the client is running, you will see a response such as the following on your terminal: `Obtained a handle on server object: IOR: (binary code)`
`Hello World! HelloServer exiting...`

`After completion kill the name server (orbd).`





Conclusion:

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

ASSIGNMENT NO. 3

Problem Statement:

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

Tools / Environment:

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

Related Theory:

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

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

Message-Passing Interface Basics:

Every MPI program must contain the preprocessor directive:

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

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

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

The principal weakness of this limited form of processing is that the processes on different nodes run entirely independent of each other. It cannot enable capability or coordinated computing. **To get the different processes to interact, the concept of communicators is needed.** MPI programs are made up of concurrent processes executing at the same time that in almost all cases

are also communicating with each other. To do this, an object called the "communicator" is provided by MPI. Thus the user may specify any number of communicators within an MPI program, each with its own set of processes. "**MPI_COMM_WORLD**" communicator contains all the concurrent processes making up an MPI program.
The size of a communicator is the number of processes that makes up the particular communicator. The following function call provides the value of the number of processes of the specified communicator:

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

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

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

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

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

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

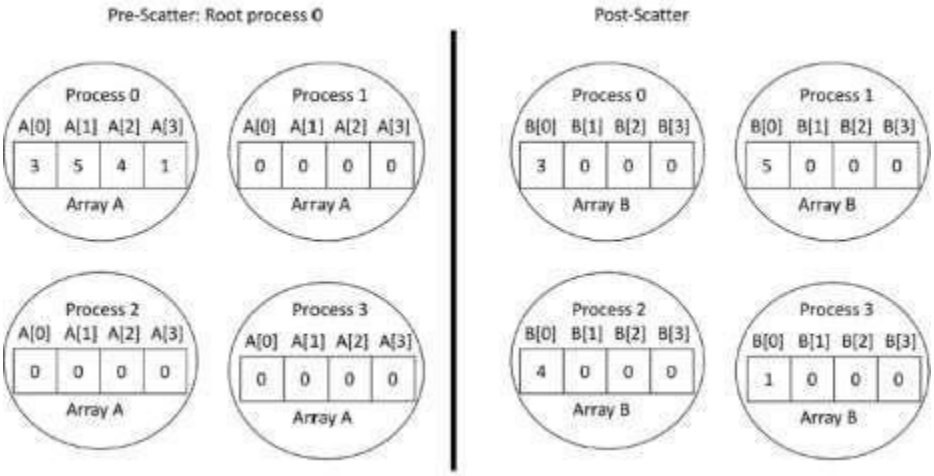
The `receive` command (`MPI_Recv`) describes both the data to be transferred and the connection to be established. The MPI_Recv construct is structured as follows:

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

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

Communication Collectives: Communication collective operations can dramatically expand interprocess communication from point-to-point to n-way or all-way data exchanges.
The scatter operation: The scatter collective communication pattern, like broadcast, shares data of one process (the root) with all the other processes of a communicator. But in this case it partitions a set of data of the root process into subsets and sends one subset to each of the processes. Each receiving process gets a different subset, and there are as many subsets as there are processes. In this example the send array is A and the receive array is B. B is initialized to 0. The root process (process 0 here) partitions the data into subsets of length

1 and sends each subset to a separate process.



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

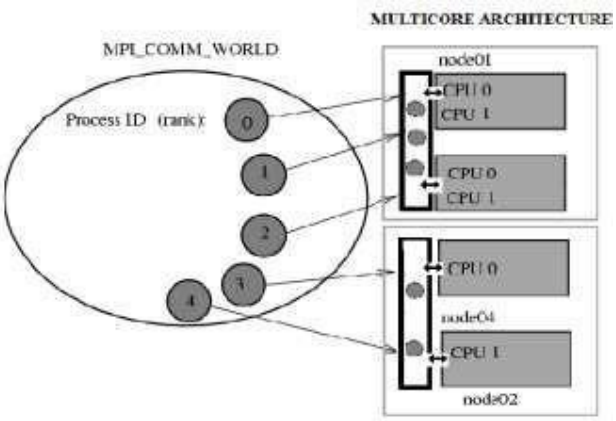
MPJ Express is essentially a middleware that supports communication between individual processors of clusters. **The programming model followed by MPJ Express is Single Program Multiple Data (SPMD).**

The multicore configuration is meant for users who plan to write and execute parallel Java

applications using MPJ Express on their desktops or laptops which contains shared memory and multicore processors. In this configuration, users can write their message passing parallel application using MPJ Express and it will be ported automatically on multicore processors. We 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 Hello World parallel Java program and save it as `HelloWorld.java (Asign2.java)`.
3. Compile a simple Hello World `(Asign)` parallel 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>
```

Steps for mpi path set up
Go to following path and copy path from properties option
paste path on terminal and write /configure --prefix=/home/it/LP5/openmpi-4.1.4/ as shown below

it@PGDERP1:~$ /home/it/LP5/openmpi-4.1.4/configure --prefix=/home/it/LP5/openmpi-4.1.4/

it@PGDERP1:~$ make all
it@PGDERP1:~$ make install

it@PGDERP1:~$ echo "export PATH=\$path:\$HOME/opt/openmpi/bin">>$HOME/.bashrc
it@PGDERP1:~$ echo "export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:\$HOME/opt/openmpi/lib">>$HOME/.bashrc
it@PGDERP1:~$ gedit hello.c

it@PGDERP1:~$ mpicc hello.c

it@PGDERP1:~$ echo "export PATH=\$path:\$HOME/opt/openmpi/bin">>$HOME/.bashrc

it@PGDERP1:~$ echo "export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:\$HOME/opt/openmpi/lib">>$HOME/.bashrc
it@PGDERP1:~$ gedit hello.c
it@PGDERP1:~$ mpicc hello.c
mpicc: error while loading shared libraries: libopen-pal.so.40: cannot open shared object file: No such file or directory
it@PGDERP1:~$ sudo ldconfig
[sudo] password for it:
it@PGDERP1:~$ mpicc hello.c
it@PGDERP1:~$ mpirun -np 2 ./a.out
Hello, world, I am 0 of 2

Hello, world, I am 1 of 2
it@PGDERP1:~$

Otherwise use following commands from configure

Configure command

/home/comp/Downloads/LP5/openmpi-4.1.4/configure --prefix=/home/comp/Downloads/LP5/openmpi-4.1.4/

make all
make install

(base) comp@SL19:~/Downloads$ echo "export PATH=\$path:\$HOME/opt/openmpi/bin">>$HOME/.bashrc
(base) comp@SL19:~/Downloads$ echo "export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:\$HOME/opt/openmpi/lib">>$HOME/.bashrc

(base) comp@SL19:~/Downloads$ gedit hello.c
(base) comp@SL19:~/Downloads$ mpicc hello.c

Command 'mpicc' not found, but can be installed with:

sudo apt install lam4-dev # version 7.1.4-6build2, or
sudo apt install mpich # version 3.3.2-2build1
sudo apt install openmpi-bin # version 4.0.3-0ubuntu1
Choose sudo apt install openmpi-bin # version 4.0.3-0ubuntu1 command to install compiler

sudo apt install openmpi-bin # version 4.0.3-0ubuntu1

Type Hello world program compile using mpicc compiler and run using mpirun command

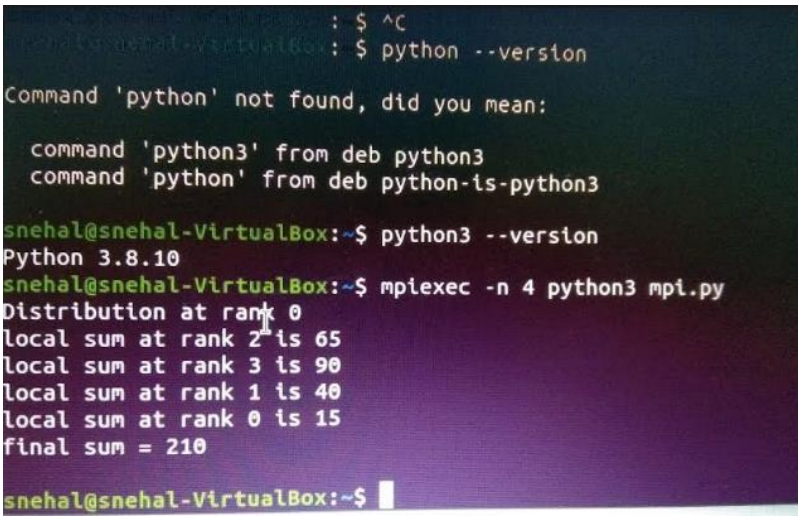
(base) comp@SL19:~/Downloads$ mpicc hello.c
(base) comp@SL19:~/Downloads$ mpirun -np 2 ./a.out
Hello,world, I am 1 of 2
Hello,world, I am 0 of 2
(base) comp@SL19:~/Downloads$

Execute Addition of 20 numbers program by using python on Ubntu Terminal

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

num = [i + 1 for i in range(20)] # N=20, n=4

if rank == 0:
    s = [0] * 4
```

```
    print("Distribution at rank", rank)
    for i in range(1, 4):
        comm.send(num[i * 5: (i + 1) * 5], dest=i, tag=1) # N/n i.e. 20/4=5
    local_sum = sum(num[:5])
    for i in range(1, 4):
        s[i] = comm.recv(source=i, tag=1)
    print("local sum at rank", rank, "is", local_sum)
    total_sum = local_sum + sum(s[1:])
    print("final sum =", total_sum, "\n")

else:
    k = comm.recv(source=0, tag=1)
    local_sum = sum(k)
    print("local sum at rank", rank, "is", local_sum)
    comm.send(local_sum, dest=0, tag=1)

MPI.Finalize()
```

Output:-



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:

Java Programming Environment, JDK1.8 or higher version (Eclipse or Netbeans)

Related Theory:

Berkeley's Algorithm is an algorithm that is used for clock Synchronization in distributed systems. This algorithm is used in cases when some or all systems of the distributed network have one of these issues −

- A. The machine does not have an accurate time source.
- B. The network or machine does not have a UTC server.

The algorithm is designed to work in a network where clocks may be running at slightly different rates, and some computers may experience intermittent communication failures.

The basic idea behind Berkeley's Algorithm is that each computer in the network periodically sends its local time to a designated "master" computer, which then computes the correct time for the network based on the received timestamps. The master computer then sends the correct time back to all the computers in the network, and each computer sets its clock to the received time.

Distributed system contains multiple nodes that are physically separated but are linked together using a network.

Berkeley's Algorithm

In this algorithm, the system chooses a node as master/ leader node. This is done from pool nodes in the server.

The algorithm is −

- An election process chooses the master node in the server.
- The leader then polls followers that provide their time in a way similar to Cristian's Algorithm, this is done periodically.
- The leader then calculates the relative time that other nodes have to change or adjust to synchronize to the global clock time which is the average of times that are provided to the leader node.

Let's sum-up steps followed to synchronize the clock using the Berkeley algorithm,

Nodes in the distributed system with their clock timings −

N1 -> 14:00 (master node)

N2 -> 13: 46

N3 -> 14: 15

Step 1 − The Leader is elected, node N1 is the master in the system.

Step 2 − leader requests for time from all nodes.
N1 -> time : 14:00
N2 -> time : 13:46
N3 -> time : 14:20
Step 3 − The leader averages the times and sends the correction time back to the nodes.
N1 -> Corrected Time 14:02 (+2)
N2 -> Corrected Time 14:02 (+16)
N3 -> Corrected Time 14:02 (-18)
This shows how the synchronization of nodes of a distributed system is done using Berkeley's algorithm.

1)This Module shows Berkeley Algorithm Working using server(Master) with two Clients(Slaves)

```
import java.util.Date;
import java.util.Scanner;
import java.text.ParseException;
import java.text.SimpleDateFormat;

public class berkeley {
   public static void berkeleyAlgo(String servertime, String time1, String time2) {
      System.out.println("Server Clock   = " + servertime);
      System.out.println("Client Clock 1 = " + time1);
      System.out.println("Client Clock 2 = " + time2);
      SimpleDateFormat sdf = new SimpleDateFormat("mm:ss");
      try {
         /* Converting time to Milliseconds */
         long s = sdf.parse(servertime).getTime(); //Server Time
         long t1 = sdf.parse(time1).getTime();  //Client1 Time
         long t2 = sdf.parse(time2).getTime(); //Client2 Time
         /* Calcuating time differences w.r.t server */
        //  long s = s – s=0;
         long st1 = t1 - s;  //Client1-Server time in milisecond((1 sec=1000 milisecond so divide by 1000)
         System.out.println("t1 - s = "+st1/1000);
         long st2 = t2 - s; //client2 –server time in millisecond(1 sec=1000 milisecond so divide by 1000)
         System.out.println("t2 - s = "+st2/1000);
         /*Calculate Average of all millisecond times here server time –server time=0 in millisecond therefore take 0
value i.e. Fault tolerant Average */
         long aveg = (st1 + st2 + 0) / 3;
         System.out.println("(st1 + st2 + 0)/3 = "+aveg/1000); //print average time in milisecond
         /* Adjust clock values by adding or substracting average clock value from client1 time and client1 time
Adjustment */
         long adjserver = aveg+s;
         long adj_t1 = aveg-st1;
         long adj_t2 = aveg-st2;
         System.out.println("t1 adjustment = "+adj_t1/1000);
         System.out.println("t2 adjustment = "+adj_t2/1000);
```

```
            /* sync clock values of clients and server */
            System.out.println("Synchronized Server Clock  = "+sdf.format(new Date(adjserver)));
            System.out.println("Synchronized Client1 Clock = "+sdf.format(new Date(t1+adj_t1)));
            System.out.println("Synchronized Client2 Clock = "+sdf.format(new Date(t2+adj_t2)));
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter server time (mm:ss): ");
        String servertime = input.next();
        System.out.print("Enter client clock 1 (mm:ss): ");
        String time1 = input.next();
        System.out.print("Enter client clock 2 (mm:ss): ");
        String time2 = input.next();

        berkeleyAlgo(servertime, time1, time2);
        input.close();
    }
}
```

Execution steps

$ edit Berkeley.java

$javac Berkeley.java

$ `java berkeley`

```
Output:-
Enter server time (mm:ss): 3:10
Enter client clock 1 (mm:ss): 2:50
Enter client clock 2 (mm:ss): 3:00
Server Clock   = 3:10
Client Clock 1 = 2:50
Client Clock 2 = 3:00
t1 - s = -20
t2 - s = -10
(st1 + st2 + 0)/3 = -10
t1 adjustment = 10
t2 adjustment = 0
Synchronized Server Clock  = 03:00
Synchronized Client1 Clock = 03:00
Synchronized Client2 Clock = 03:00
```

2)We can implement Berkeley Algorithm for server with Multiple Clients by using Multithreading

Server Module

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

public class BerkeleyServer {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(9876);
            System.out.println("Server is waiting for clients...");

            while (true) {
                Socket clientSocket = serverSocket.accept();
                new ServerThread(clientSocket).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ServerThread extends Thread {
    private Socket clientSocket;

    public ServerThread(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    @Override
    public void run() {
        try {
            BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

            // Get client's time
            long clientTime = Long.parseLong(in.readLine());
            long serverTime = System.currentTimeMillis();

            // Send server time to the client
            out.println(serverTime);

            // Calculate the offset
            long offset = serverTime - clientTime;

            // Send offset to the client for clock adjustment
            out.println(offset);

            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Client Module:-

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

public class BerkeleyClient {
    public static void main(String[] args) {
        try {
            Socket socket = new Socket("localhost", 9876);

            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

            // Send client's time to the server
            out.println(System.currentTimeMillis());

            // Receive server time
            long serverTime = Long.parseLong(in.readLine());

            // Receive offset and adjust client's clock
            long offset = Long.parseLong(in.readLine());
            long adjustedTime = System.currentTimeMillis() + offset;

            System.out.println("Server time: " + serverTime);
            System.out.println("Adjusted client time: " + adjustedTime);

            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

ASSIGNMENT NO. 5

Problem Statement:

Implement token ring based mutual exclusion algorithm.

Tools / Environment:

Java Programming Environment, JDK 8, Netbeans IDE

Related Theory:

Mutual exclusion is a common term appearing frequently in computer sciences. In essence, it's a mechanism of concurrency control allowing exclusive access to some resource (or "critical region"). Token passing is an algorithm for distributed mutual exclusion (DME) and will be our focus in this post.

DME specifications usually make the following assumptions:

- Network delivers message in order, e.g. TCP (sometimes)
- Every message is eventually delivered (usually)
- Messages are never duplicated. Duplication may result granting resources to multiple clients, which is not what mutual exclusion demands (usually)

DME specifications are:

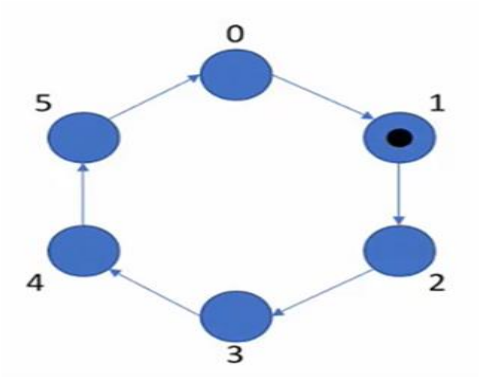
- Mutual exclusion, at most one client is in a critical section (always)
- Non-starvation. A requesting client enters critical section eventually (usually)
- Non-overtaking. A client cannot enter critical section more than once while another client waits (usually)

In a token ring, a client holds a token and then sends it to the next one after exiting its critical section. When we make assumptions about a token ring, we

- do not need to have network delivering messages in order, because at any given time in a token, there is at most one message in transit.
- ensure every message is eventually delivered. Otherwise, the system won't make progress, and we will not have non-starvation guarantee.
- Need non-duplication for messages. Otherwise, we violate the fundamental properties of this

protocol, or no mutual exclusion.

• clients don't spuriously release. This will be clear later when we demonstrate what happens if clients release multiple times.



We want to guarantee that

• Mutual exclusion holds.

• Non-starvation

• Non-overtaking, because token will get through every client in the network first because repetition happens.

To analyze token performance, we use the above performance metrics (message complexity, response time, and throughput)

• Message complexity: when the system is under low load, the message complexity is unbounded because there may be an arbitrary number of messages being sent throughout the network where no one is in the critical section. When system is under high load, the message complexity is 1.

• Response time: when the system is under low load, there could be messages times (where is the total number of clients). When under high load, the response time would be 1 message time.

• Throughput: the maximum throughput would be 1/(message time + CS time)

Token Ring algorithm achieves mutual exclusion in a distributed system by creating a bus network of processes. A logical ring is constructed with these processes and each process is assigned a position in the ring. Each process knows who is next in line after itself.

The algorithm works as follows:

When the ring is initialized, process 0 is given a token.

The token circulates around the ring.

When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region.

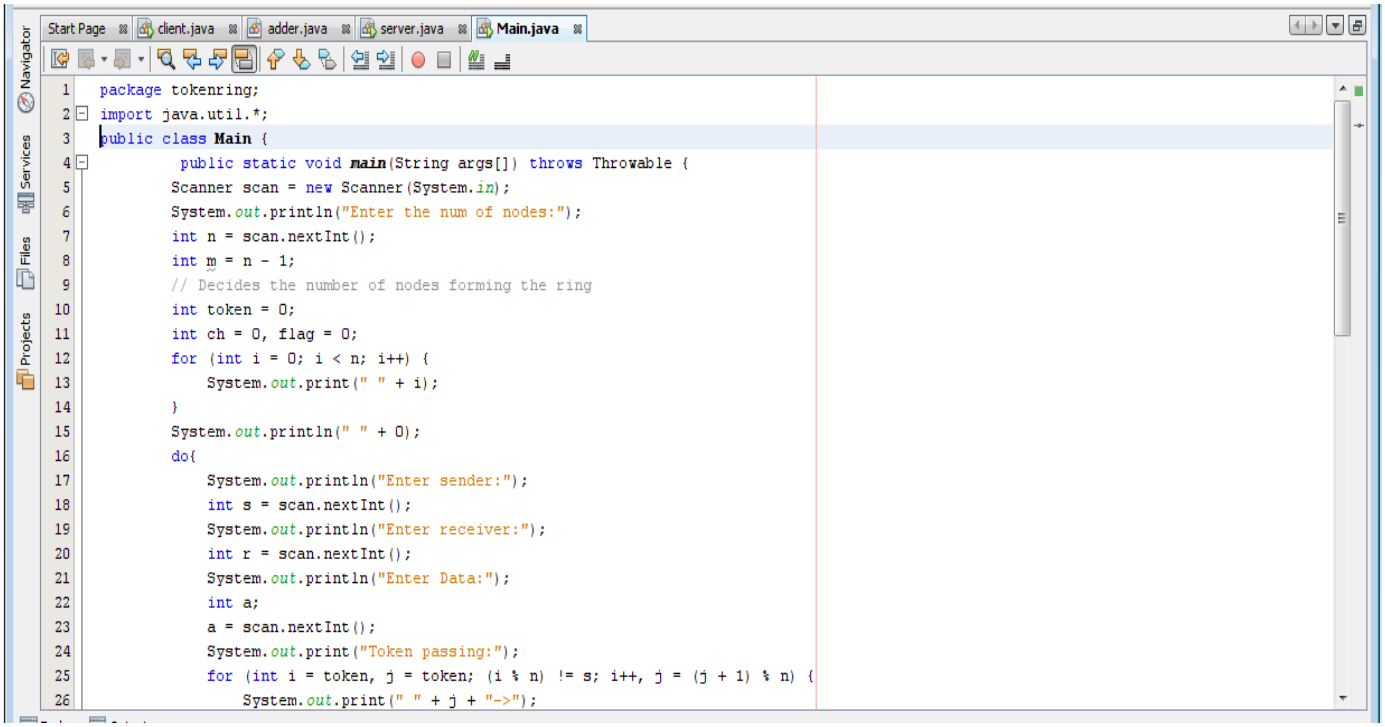
If so, the process enters the region, does all the work it needs to, and leaves the region.

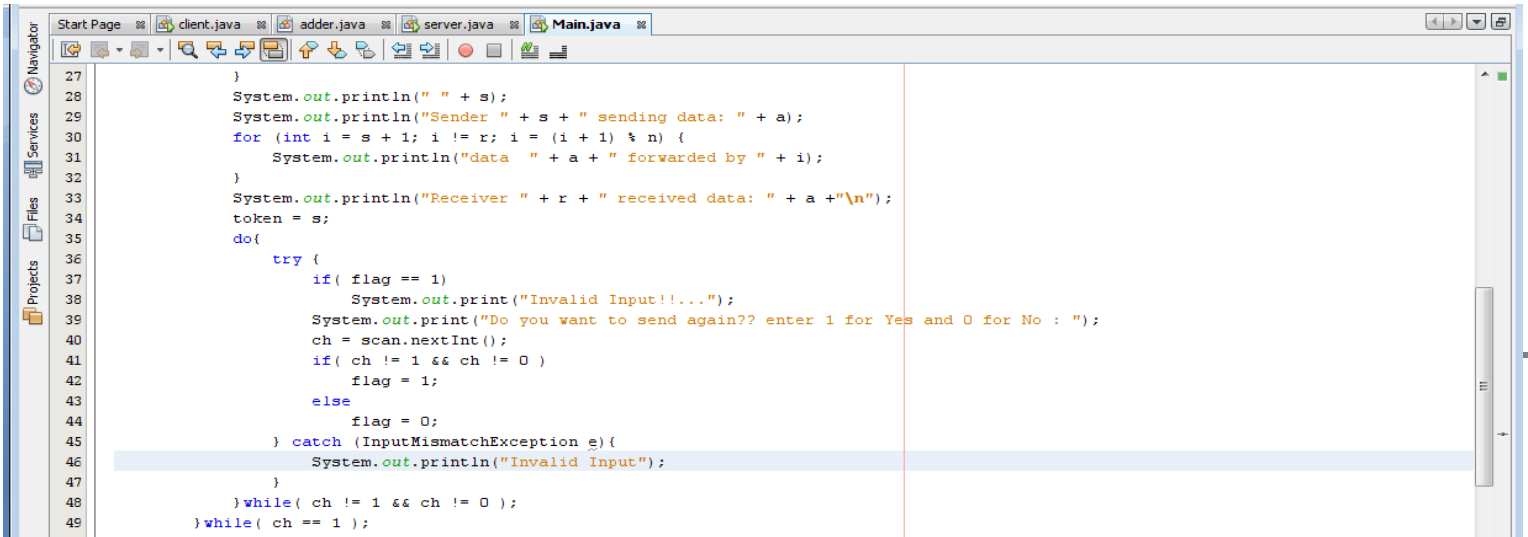
After it has exited, it passes the token to the next process in the ring.

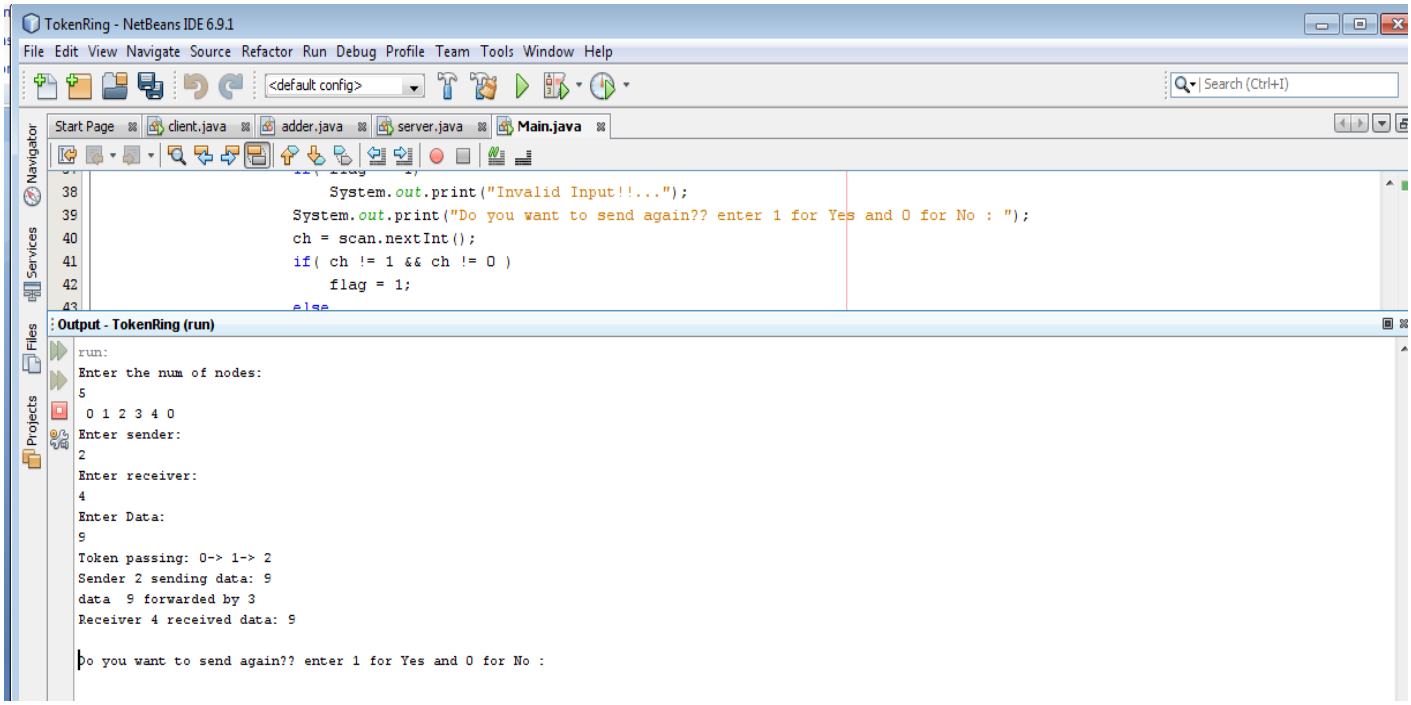
It is not allowed to enter the critical region again using the same token.

If a process is handed the token by its neighbor and is not interested in entering a critical region,

it just passes the token along to the next process.








```
package tokenring;

import java.util.*;

public class TokenRing {
    public static void main(String args[]) throws Throwable {
      Scanner scan = new Scanner(System.in);
      System.out.println("Enter the num of nodes:");
      int n = scan.nextInt();
      int m = n - 1;
      // Decides the number of nodes forming the ring
      int token = 0;
      int ch = 0, flag = 0;
      for (int i = 0; i < n; i++) {
         System.out.print(" " + i);
```

```
        }
        System.out.println(" " + 0);
        do{
            System.out.println("Enter sender:");
            int s = scan.nextInt();
            System.out.println("Enter receiver:");
            int r = scan.nextInt();
            System.out.println("Enter Data:");
            int a;
            a = scan.nextInt();
            System.out.print("Token passing:");
            for (int i = token, j = token; (i % n) != s; i++, j = (j + 1) % n) {
                System.out.print(" " + j + "->");
            }
            System.out.println(" " + s);
            System.out.println("Sender " + s + " sending data: " + a);
            for (int i = s + 1; i != r; i = (i + 1) % n) {
                System.out.println("data  " + a + " forwarded by " + i);
            }
            System.out.println("Receiver " + r + " received data: " + a +"\n");
            token = s;
            do{
                try {
                    if( flag == 1)
                            System.out.print("Invalid Input!!...");
                    System.out.print("Do you want to send again?? enter 1 for Yes and 0 for No : ");
                    ch = scan.nextInt();
                    if( ch != 1 && ch != 0 )
                            flag = 1;
                    else
                            flag = 0;
                } catch (InputMismatchException e){
                    System.out.println("Invalid Input");
                }
            }while( ch != 1 && ch != 0 );
        }while( ch == 1 );
    }

}
```

Output:-

```
Enter the num of nodes:
5
 0 1 2 3 4 0
Enter sender:
2
```

Enter receiver:
4
Enter Data:
9
Token passing: 0-> 1-> 2
Sender 2 sending data: 9
data 9 forwarded by 3
Receiver 4 received data: 9

Do you want to send again?? enter 1 for Yes and 0 for No :

ASSIGNMENT NO.6

Problem Statement:

Implement Bully and Ring algorithm for leader election.

Tools / Environment:

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

Related Theory:

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

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

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

Bully Algorithm:

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

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

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

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

Ring Algorithm:

Initiation:
1. When a process notices that coordinator is not functioning:
2. Another process (initiator) initiates the election by sending "ELECTION" message (containing its own process number)
Leader Election:
3. Initiator sends the message to it's successor (if successor is down, sender skips over it and goes to the next member along the ring, or the one after that, until a running process is located).
4. At each step, sender adds its own process number to the list in the message.
5. When the message gets back to the process that started it all: Message comes back to initiator.
In the queue the **process with maximum ID Number wins**.
Initiator announces the winner by sending another message around the ring.

Designing the solution:

A. For Ring Algorithm
Initiation:

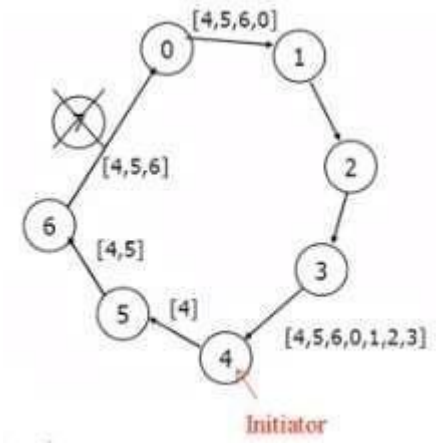
1. Consider the Process 4 understands that Process 7 is not responding.

2.Process 4 initiates the Election by sending "ELECTION" message to it's successor (or next alive process) with it's ID.

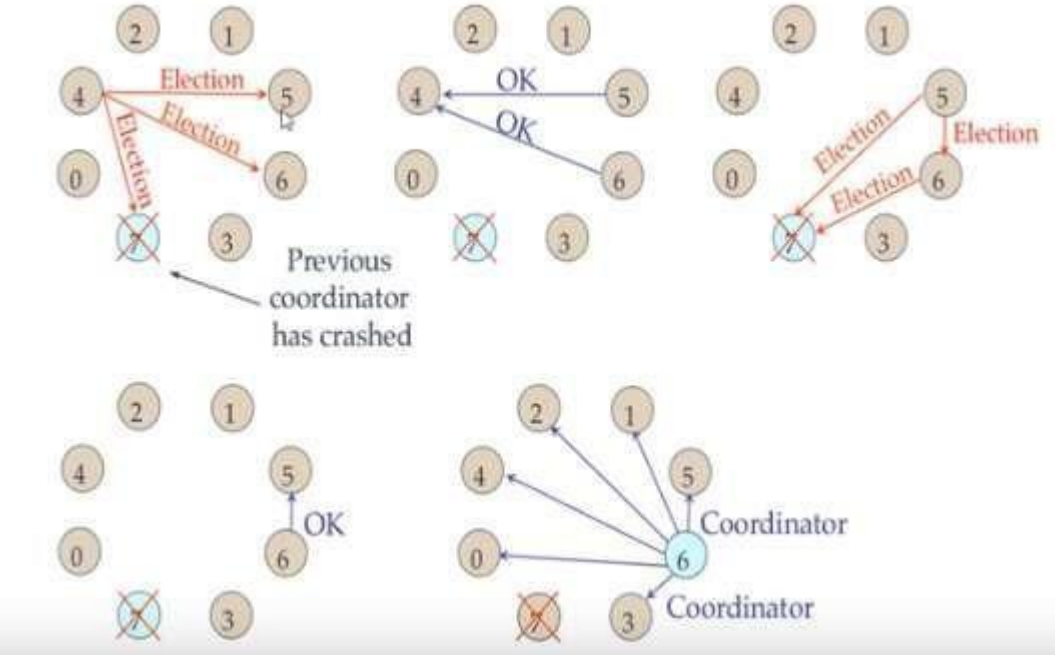
Leader Election:

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



B. For Bully Algorithm:



Implementing the solution:

For Ring Algorithm:

1. Creating Class for Process which includes
   i) State: Active / Inactive
   ii) Index: Stores index of process.
   iii) ID: Process ID
2. Import Scanner Class for getting input from Console

3. Getting input from User for number of Processes and store them into object of classes.

4. Sort these objects on the basis of process id.

5. Make the last process id as "inactive".

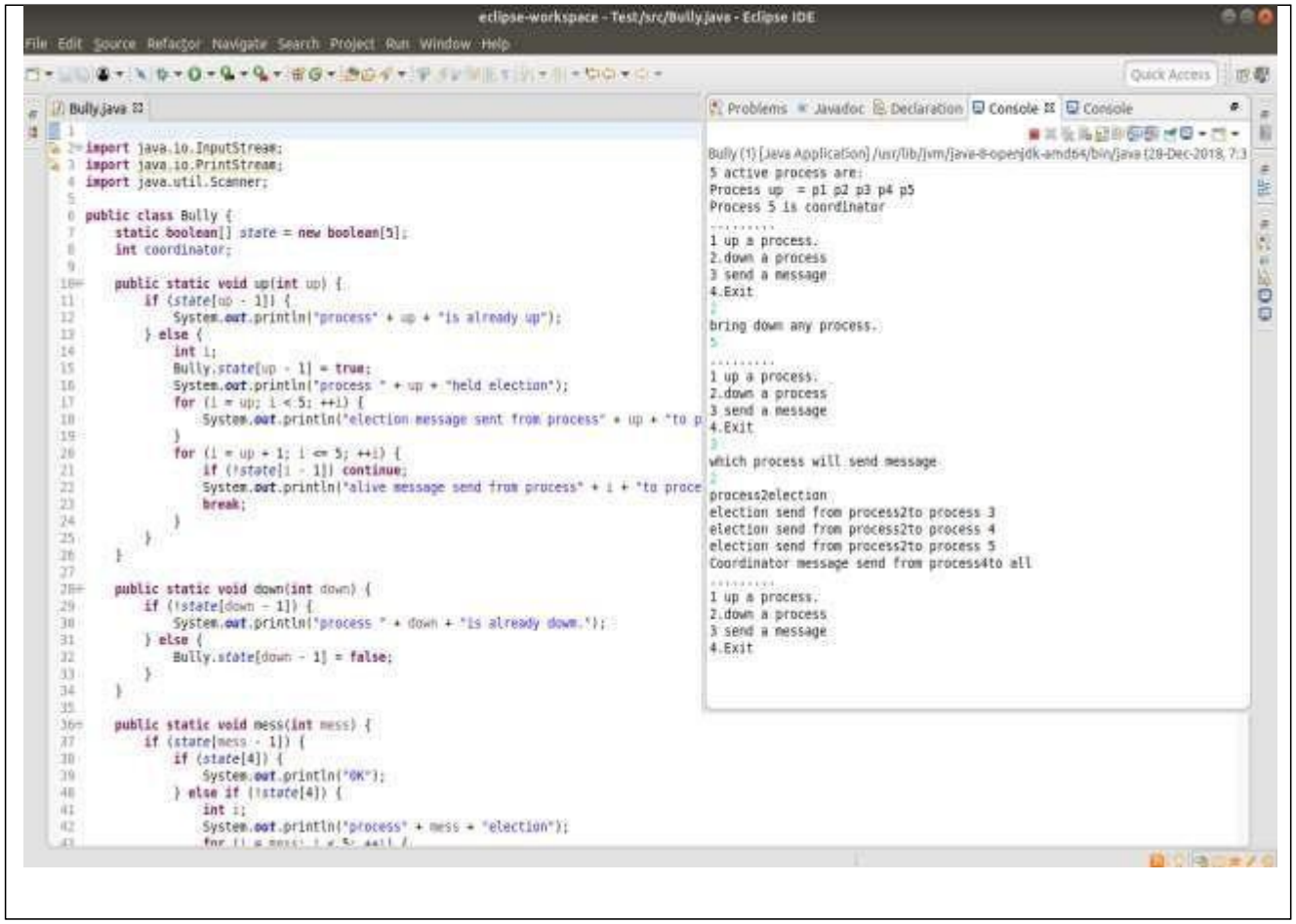
6. Ask for menu 1.Election 2.Exit

7. Ask for initializing election process.

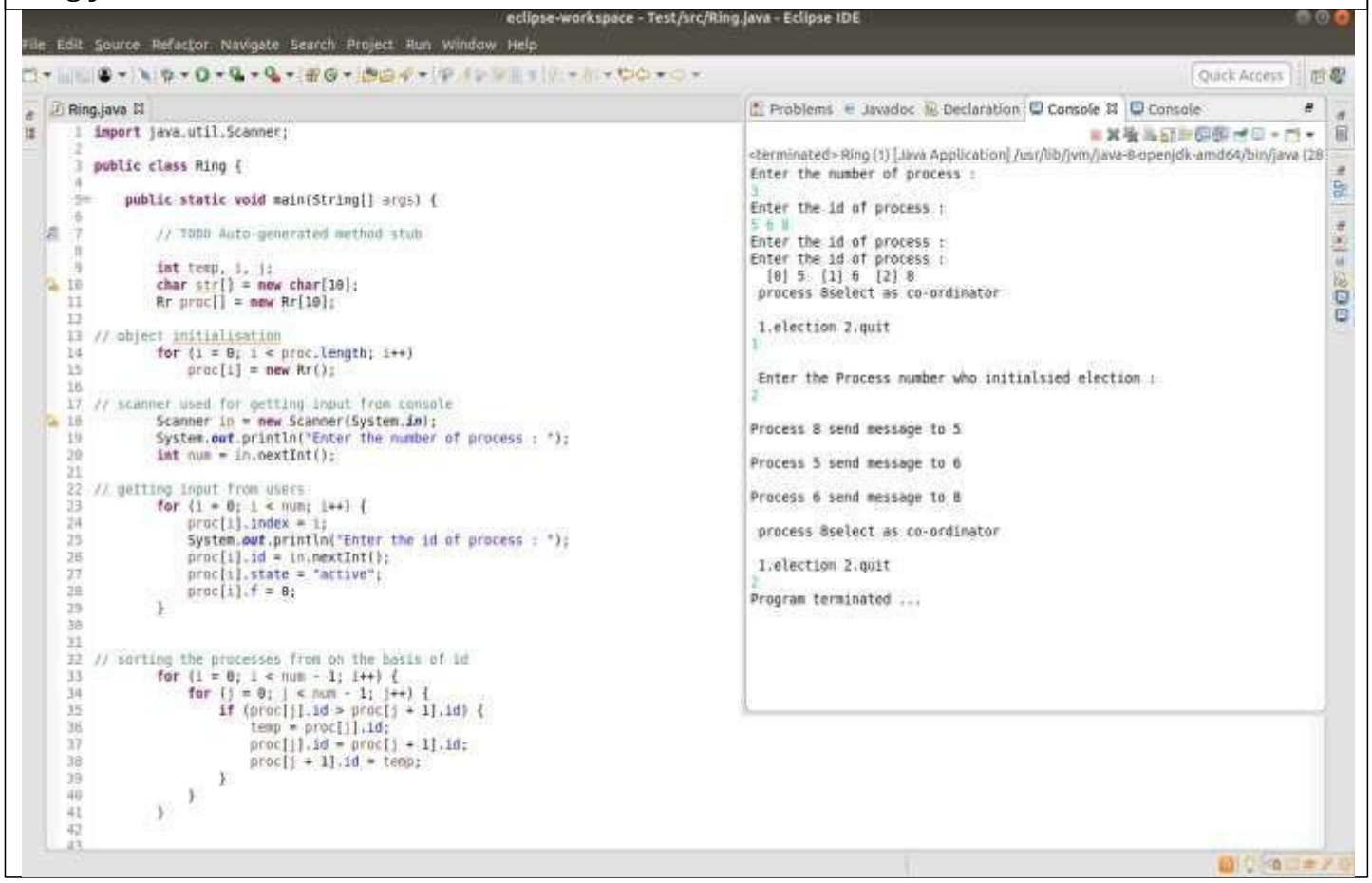
8.These inputs will be used by Ring Algorithm.

Writing the source code:

Bully.java



Ring.java




```java
import java.util.Scanner;

public class Ring {

    public static void main(String[] args) {

        // TODO Auto-generated method stub

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

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

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

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


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

Compiling and Executing the solution:

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

The output is associated in the above section.

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

ASSIGNMENT NO. 7

Problem Statement:

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

Tools / Environment:

Java Programming Environment, JDK 8, Netbeans IDE(8.0.1/8.2) 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:

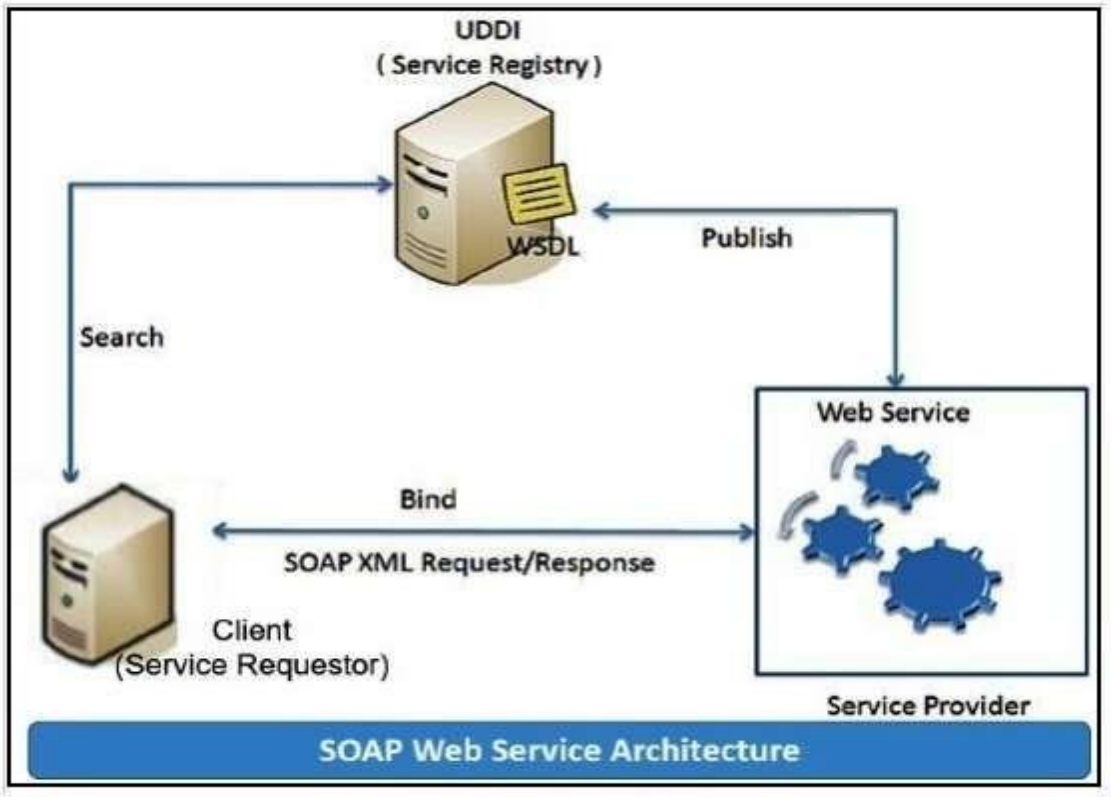
1. The client program that wants to interact with another application prepares its request content as a SOAP message.
2. Then, the client program sends this SOAP message to the server web service as an HTTP POST request with the content passed as the body of the request.
3. The web service plays a crucial 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 theSOAP HTTP request invoked by the client program with this response.
6. The client program web service reads the SOAP response message to receive the outcome of the server program for the request content it sent as a request.

SOAP web services:

Simple Object Access Protocol (**SOAP**) is an XML-based protocol for accessing web services. It is a W3C recommendation for communication between two applications, and it is a platform- and language-independent technology in integrated distributed applications.
While XML and HTTP together make the basic platform for web services, the following are the key components of standard SOAP web services:

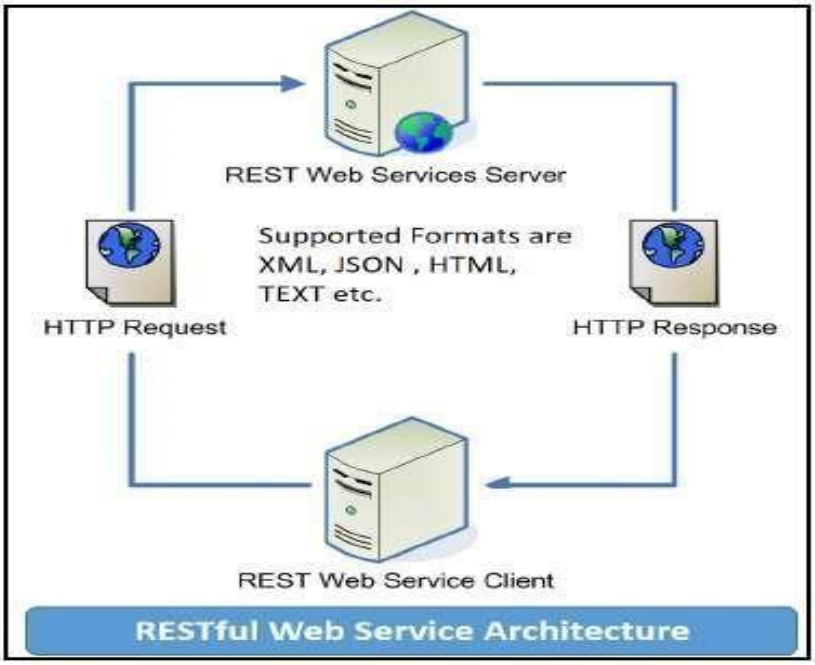
Universal Description, Discovery, and Integration (**UDDI**)*:* UDDI is an XMLbased framework for describing, discovering, and integrating web services. It acts as a directory of web service interfaces described in the WSDL language.

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



RESTful web services

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



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

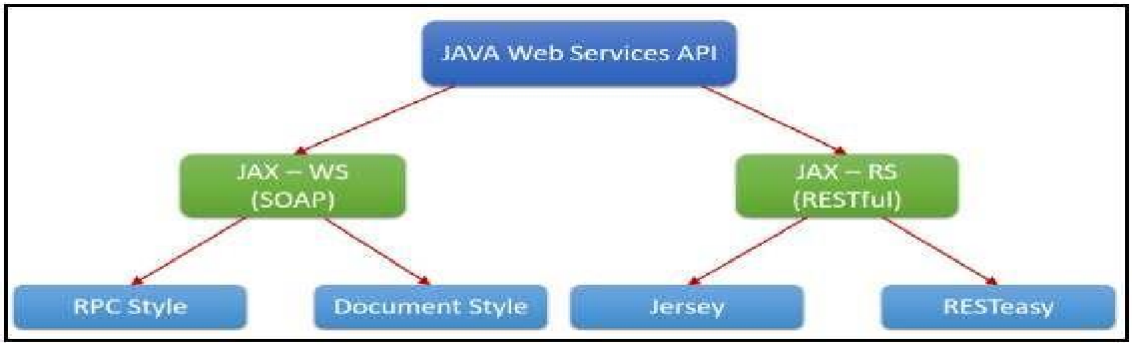
SOAP	REST
SOAP is a protocol.	REST is an architectural style.
SOAP stands for Simple Object Access Protocol.	REST stands for REpresentational State Transfer.
SOAP can't use REST because it is a protocol.	REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.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

Designing the solution:

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

1. **JAX-WS**: JAX-WS stands for Java API for XML Web Services. JAX-WS is XML based Java API to build web services server and client application.
2. **JAX-RS**: Java API for RESTful Web Services (JAX-RS) is the Java API for creating REST web services. JAX-RS uses annotations to simplify the development and deployment of web services.

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



Implementing the solution:

1. **Creating a web service container CalculatorWSApplication:**

For Container Create New Project for `CalculatorWSApplication`.

With following steps

File->New project->select categpry as Java Web and projects as Web Application.->Give Project Name as `CalculatorWSApplication`->Choose Server As GlassFish Server 4.1->select next and click Finish button.

2) On left panel right click on `CalculatorWSApplication->new->choose Web Service->Give web Service name as CalculatorWS->`Create a package `org.me.calculator->choose option as create Web Service from Scratch->Tick mark on Implement Web Service as State less Session Bean(State less does not tracks information about the stateof a connectionor application.Stateless system sends a request to the server and relays the response (state)back without storing any information)->Click on Finish button.`

`In this way,We have created a container. Now we add an operation to the web service. For that on the left panel ,open web services ->Right click on CalculatorWS->choose Add operations ->Name the operation as add and Return type will be an integer->add parameters by clicking on add button ->Name the parameter as i and Type as int ->Again Click on Add button to add Another Parameter as j and Type as int->Click on OK button.`

`Web Service operation Code will automatically add in web Service Container.(Can delete default Hello operation code)`

`In Web Service Operation add following code`

```
Public class CalculatorWS//default it will be there
{
Public int add(){//default it will be there
int k=i+j;
return k;
}
}
```

`To deploy web service Right click on` **`CalculatorWSApplication->choose Deply option`**

`Can See GlassFish Server is Successfully running`

`GlassFish Server-Open Source Fully Java EE compliant Application Server. It Implements set of Java Technologies that simplify developing and using Web Services .Glass Fish`

Server comes along with the test client but Apachi tom cat doesnot come with the test client So we have to use Netbeans along with GlassFish Servers purposefully

2)IDE starts the glassfish server, builds the application and deploys the application on server.

To Test Web Service right click on CalculatorWS->Choose Test Web Service->In Running window copy link from glassfish Server and paste it in browser.
Open browser and check with link as localhost:8080CalculatorWS/CalculatorWS?Tester.
In Methods ->enter first and second number ->click on add button.

In this way Web Service is Successfully Tested .

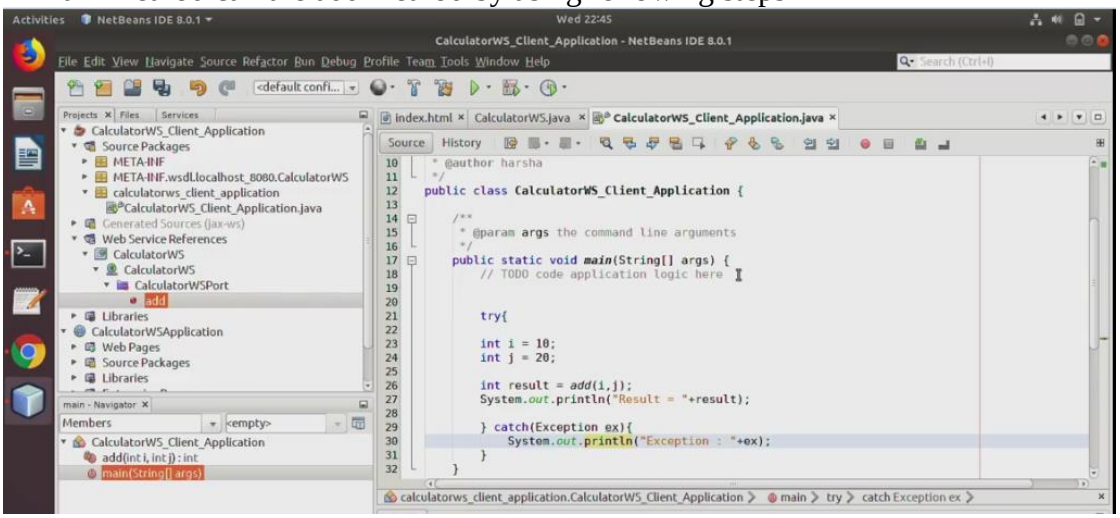
3)Create Client to Consume Web Service
Now create one client to test that web service by using following steps.

Choose File->New Project->Select Category as Java and project as Java Application->Give Project name as CalculatorWS_Client_Application->Click on Finish button.

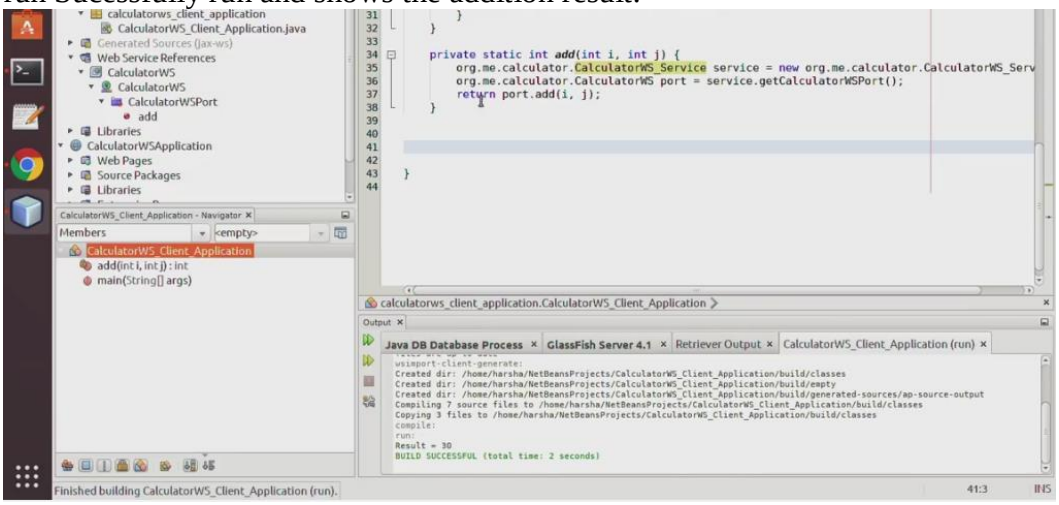
On left panel right click on CalculatorWS_Client_Application->choose New->Select Web Service Client.
In New Web Service Client Window select WSDL and Client location browse project as Calculator

WSApplication->Open it and choose CalculatorWS->Click OK->Click Finish button.
On left Panel Open Web Service References till add method .Below main Function drag the add method from the left panel and drop it in program.

In main method call the add method by using following steps



In left panel right click on Calculator_Client_Application java project and choose run option Client will run Sucessfully run and shows the addition result.



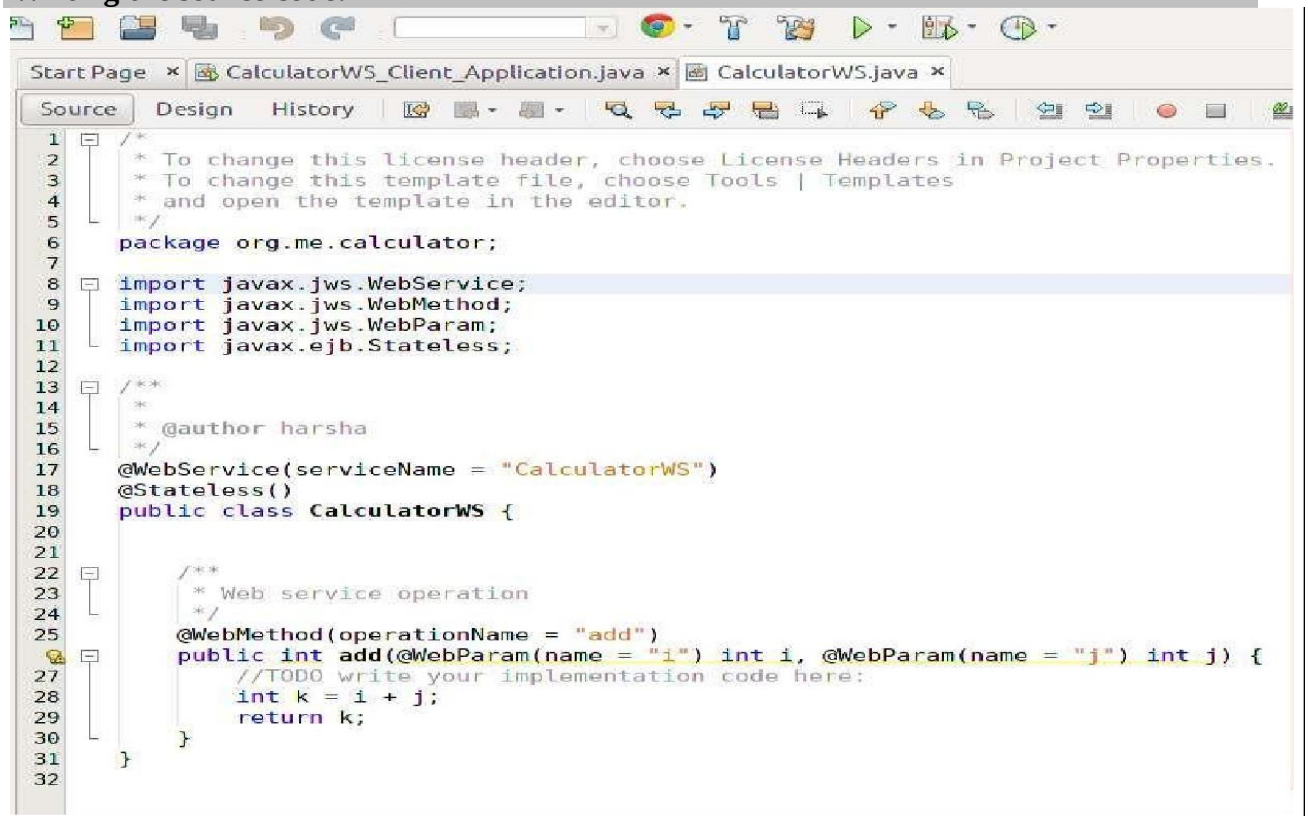
2. Consuming the Webservice:

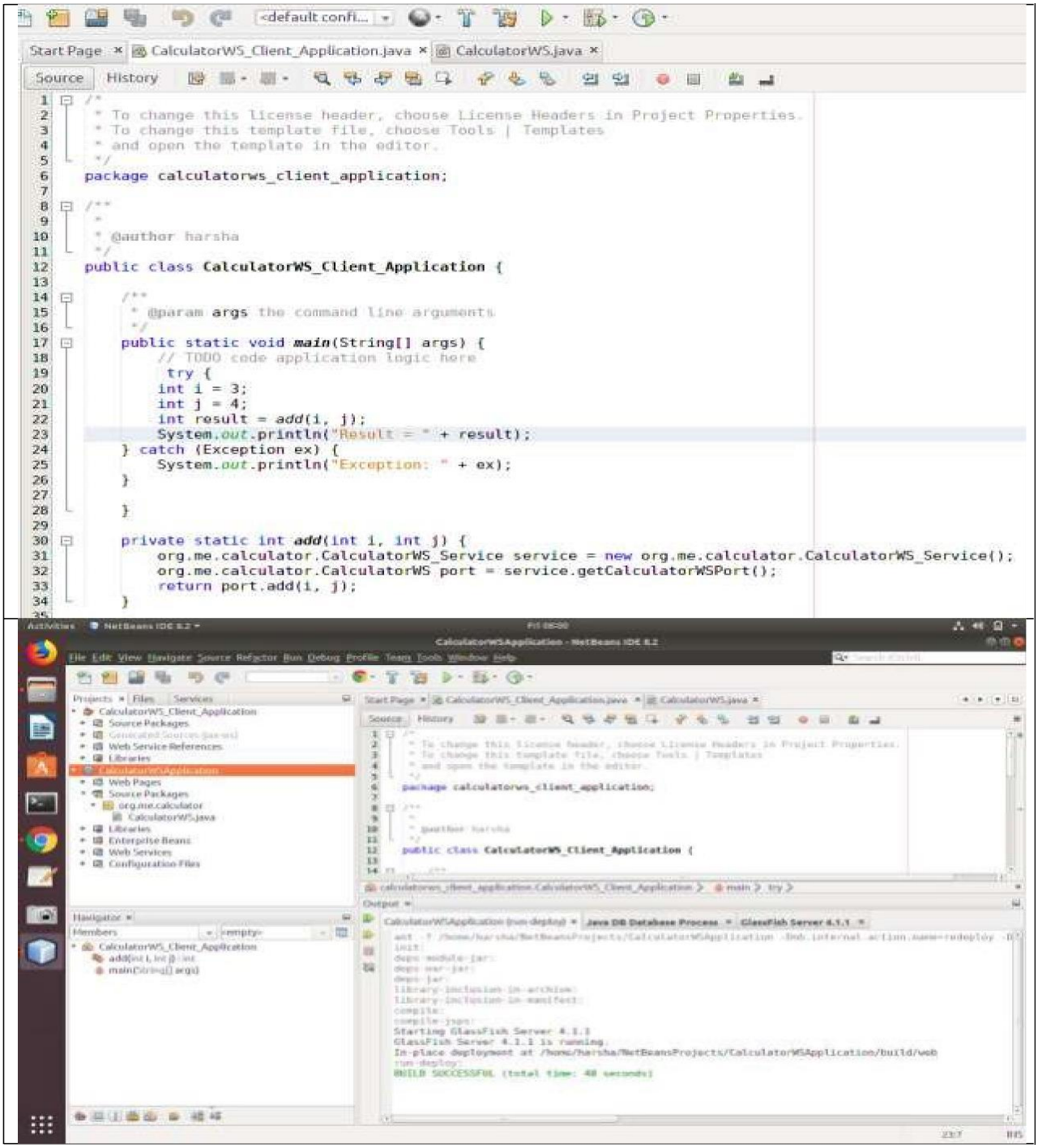
Create a project with an `CalculatorClient`

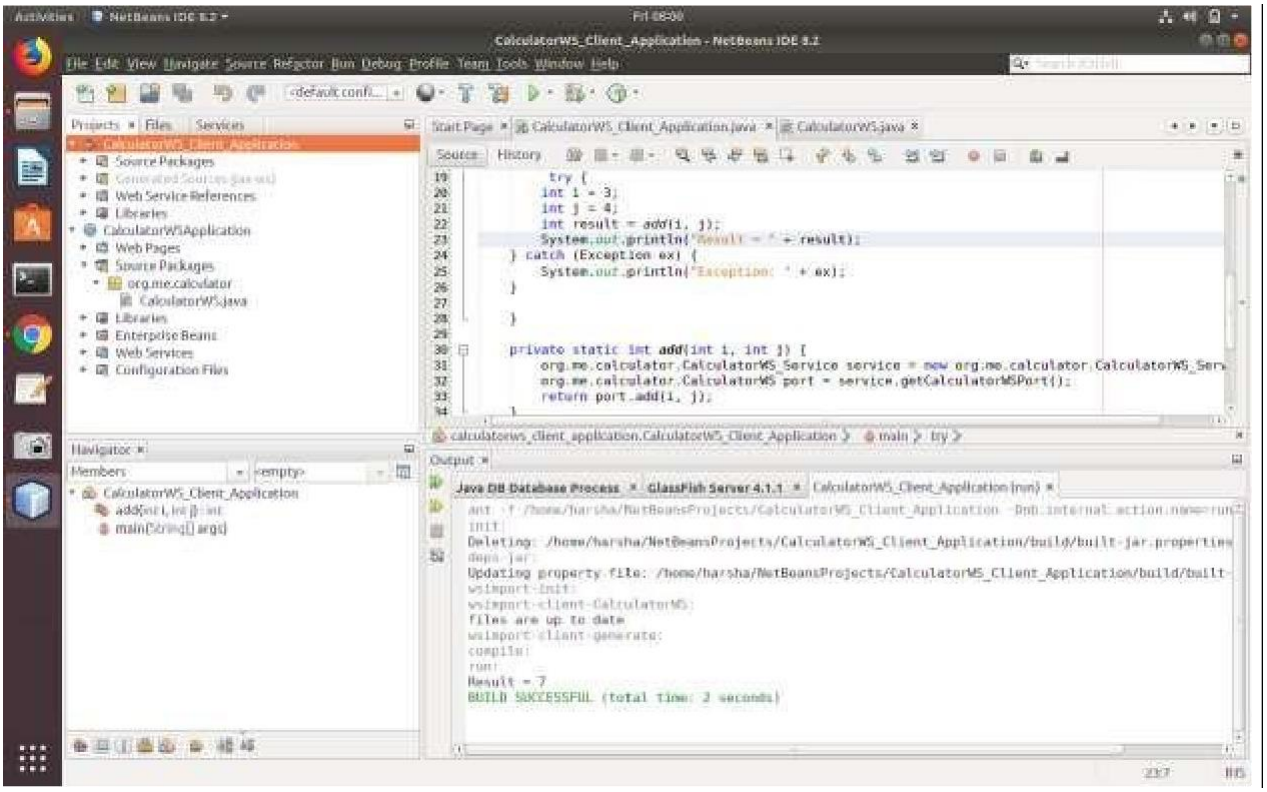
Create package `org.calculator.client`;

add java class `CalculatorWS.java, addresponse.java, add.java, CalculatorWSService.java` and `ObjectFactory.java`

Writing the source code:

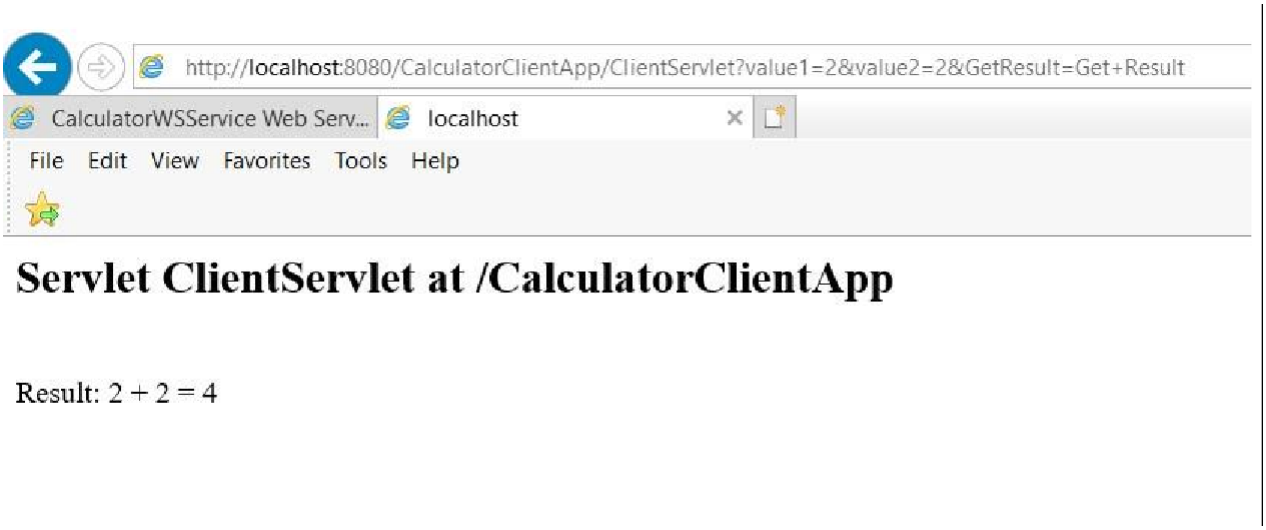



Start Page
CalculatorWS_Client_Application.java
CalculatorWS.java
Source
History
/*
* To change this license header, choose License Headers in Project Properties.
* To change this template file, choose Tools | Templates
* and open the template in the editor.
*/
package calculatorws_client_application;
/**
*
* @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);
}



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.