

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

INDEX

Sr. No.	Title	Page No.
1.	Implement multi-threaded client/server Process communication using RMI.	
2.	Develop any distributed application using CORBA to demonstrate object brokering. (Calculator or String operations).	
3.	Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.	
4.	Implement Berkeley algorithm for clock synchronization.	
5.	Implement token ring based mutual exclusion algorithm.	
6.	Implement Bully and Ring algorithm for leader election.	
7.	Create a simple web service and write any distributed application to consume the web service.	
8.	Mini Project (In group): A Distributed Application for Interactive Multiplayer Games	

Assignment No: 1

Title / Objective: Client/Server Process Communication using RMI

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

Course Outcome:

CO No.	CO Statement	BTL
--------	--------------	-----

C414454.1	Demonstrate knowledge of the core concepts and techniques in distributed systems.	Apply
-----------	---	-------

Pre-requisites: Computer Network Technology

Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

A. For Prerequisites –

B. For Assignment -

Requirements: Java Programming Environment, rmiregistry, jdk 1.8, Eclipse IDE.

Theory:

Socket:

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

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

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

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

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

- **TCP** – TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP** – UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

Socket programming for TCP:

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

- The server instantiates a `ServerSocket` object, denoting which port number communication is to occur on.
- The server invokes the `accept()` method of the `ServerSocket` class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a `Socket` object, specifying the server name and the port number to connect to.
- The constructor of the `Socket` class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a `Socket` object capable of communicating with the server.
- On the server side, the `accept()` method returns a reference to a new socket on the server that is connected to the client's socket.
- After the connections are established, communication can occur using I/O streams. Each socket has both an `OutputStream` and an `InputStream`. The client's `OutputStream` is connected to the server's `InputStream`, and the client's `InputStream` is connected to the server's `OutputStream`.
- TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.

Socket programming for UDP:

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

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

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

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

RMI :

Remote Method Invocation (RMI) is an API which allows an object to invoke a method of an object that exists in another address space, which could be on the same machine or on a remote machine.

Through RMI, object running in a JVM present on a computer (Client side) can invoke methods on an object present in another JVM (Server side).

RMI creates a public remote server object that enables client and server side communications through simple method calls on the server object.

The communication between client and server is handled by using two intermediate objects: Stub object (on client side) and Skeleton object (on server side).

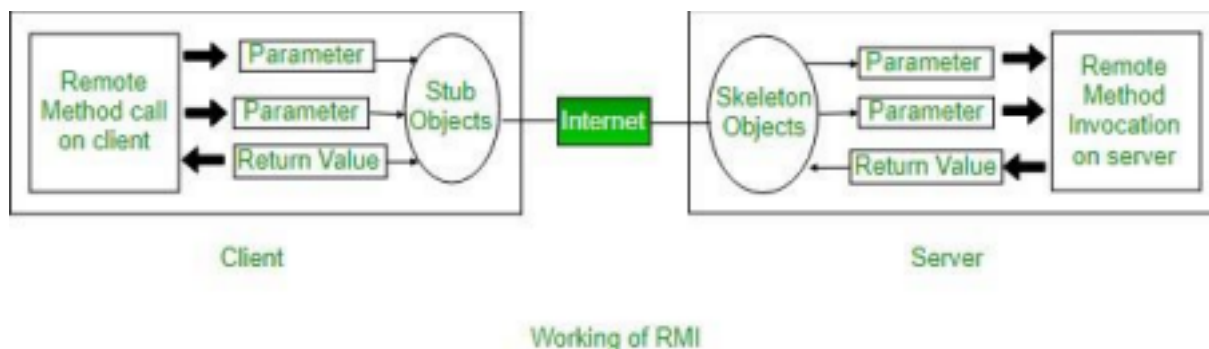
- Stub Object:

The stub object on the client machine builds an information block and sends this information to the server. The block consists of: An identifier of the remote object to be used Method name which is to be invoked Parameters to the remote JVM.

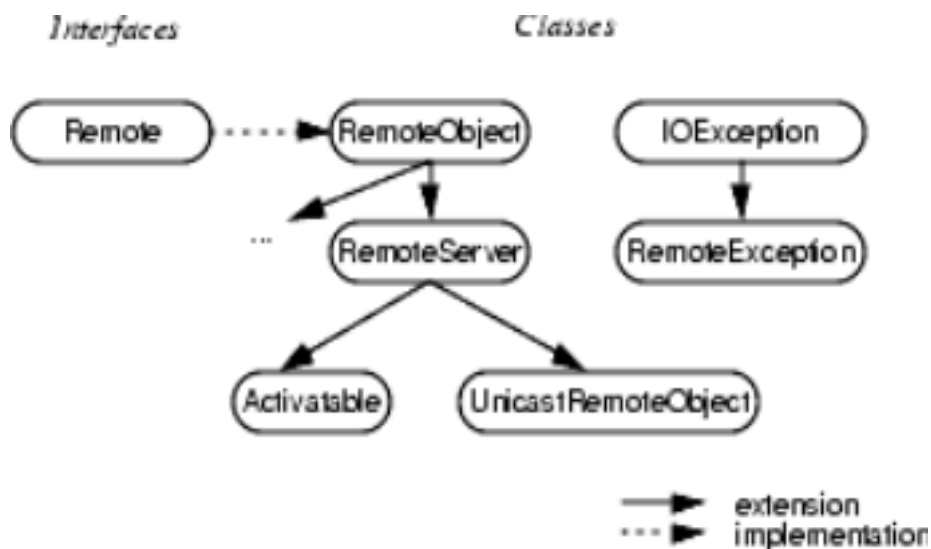
- Skeleton Object

The skeleton object passes the request from the stub object to the remote object. It performs following tasks:

- It calls the desired method on the real object present on the server.
- It forwards the parameters received from the stub object to the method



The interfaces and classes that are responsible for specifying the remote behavior of the RMI system are defined in the java.rmi package hierarchy. The following figure shows the relationship between several of these interfaces and classes:



REMOTE METHOD INVOCATION (RMI)

- `java.rmi.Remote` **Interface:**
- In RMI, a *remote* interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine.
- **The RemoteObject Class and its Subclasses:**
- RMI server functions are provided by `java.rmi.server.RemoteObject` and its subclasses, `java.rmi.server.RemoteServer` and `java.rmi.server.UnicastRemoteObject`.
- The class `java.rmi.server.RemoteObject` provides implementations for the `java.lang.Object` methods that are sensible for remote objects.
- The methods needed to create remote objects and make them available to remote clients are provided by the class `UnicastRemoteObject`.
- The `java.rmi.server.UnicastRemoteObject` class defines a singleton (unicast) remote object whose **references are valid only while the server process is alive.**

Algorithm / Methods / Steps:

Steps to implement RMI:

1. Defining a remote interface
2. Implementing the remote interface
3. Creating Stub and Skeleton objects from the implementation class using `rmic` (rmi compiler)
4. Start the `rmiregistry`
5. Create and execute the server application program
6. Create and execute the client application program.

Remote interfaces: Every remote object has a remote interface that specifies which of its methods can be invoked remotely.

Implementation:

STEP 1: Defining the remote interface:

- To create an interface which will provide the description of the methods that can be invoked by remote clients.
- This interface should extend the `Remote` interface and the method prototype within the interface should throw the `RemoteException`.

```
// Creating a Search interface (Search.java)
import java.rmi.*;
public interface Search extends Remote
{
    // Declaring the method prototype
    public String query(String search) throws
    RemoteException;
}
```

STEP 2: Implementing the remote interface

- To implement the remote interface, the class should extend to UnicastRemoteObject class of java.rmi package.

```
// Java program to implement the Search interface (SearchQuery.java)
import java.rmi.*;
import java.rmi.server.*;
public class SearchQuery extends UnicastRemoteObject
                                implements Search
{
    // Implementation of the query interface
    public String query(String search)
                                throws RemoteException
    {
        String result;
        if (search.equals("Reflection in Java"))
            result = "Found";
        else
            result = "Not Found";
        return result; } }
```

STEP 3: Creating Stub and Skeleton objects from the implementation class using rmic •

The rmic tool is used to invoke the rmi compiler that

creates the Stub and Skeleton objects. Its prototype rmic class name. The command need to be executed at the command prompt.

```
# rmic SearchQuery
```

STEP 4: Start the rmiregistry

- Start the registry service by issuing the command at the command prompt :

```
# start rmiregistry
```

STEP 5: Create and execute the server application program

- To create the server application program and execute it on a separate command prompt. • The server program uses createRegistry method of LocateRegistry class to create rmiregistry within the server JVM with the port number passed as argument. • The rebind method of Naming class is used to bind the remote object to the new name.

STEP 6: Create and execute the client application program

- The last step is to create the client application program and execute it on a separate command prompt .
- The lookup method of Naming class is used to get the reference of the Stub object.

```
//program for server application (SearchServer.java)
import java.rmi.*;
import java.rmi.registry.*;
public class SearchServer
{
    public static void main(String args[])
    {
        try
        {
            // Create an object of the interface
            // implementation class
            Search obj = new SearchQuery();
            // rmiregistry within the server JVM with port number 1900
            LocateRegistry.createRegistry(1900);
            // Binds the remote object by the name LP-V
            Naming.rebind("rmi://localhost:1900"+
                        "/LP-V",obj);
        }
        catch(Exception ae)
        {
            System.out.println(ae);
        }
    }
}
```

```
//program for client application (ClientRequest.java)
import java.rmi.*;
public class ClientRequest
{
    public static void main(String args[])
    {
        String answer,value= "RMI in Java";
        try
        {
            // lookup method to find reference of remote object
            Search access =
            (Search)Naming.lookup("rmi://localhost:1900/cl9");
            answer = access.query(value);
            System.out.println("Article on " + value +
                                " " + answer+" at
cl9");
        }
        catch(Exception ae)
        {
            System.out.println(ae);
        }
    }
}
```

Execution:

Compile and execute application programs:

```
#Javac SearchQuery.java
#rmic SearchQuery
#rmiregistry on console
```

On console-1:

Compile Server Application:

```
#javac SearchServer.java
#java SearchServer
```

On console-2:

Compile ClientRequest Application:

```
#Javac ClientRequest.java
#java ClientRequest
```

Inference:

The client-server communication through different protocols and sockets implemented successfully. The implementation of RMI with Java support through the socket API for TCP and UDP programming.

Oral questions:

1. What is RMI?
2. What is the method that is used by the RMI client to connect to remote RMI servers?
3. How distributed garbage collector does manages the disconnections detected on the client side?
4. What is the relationship between the RMI and CORBA?
5. What is Remote object?
6. What is Server object?
7. What is rmiregistry?
8. What are the different types of classes that are used in RMI
9. What is the main purpose of Distributed object applications in RMI?
10. How does the communication with remote objects occur in RMI?
11. What are the steps that are involved in RMI distributed applications?
12. What is the use of java.rmi.Remote Interface in RMI?
13. Write a program to show the remote interface using RMI.
14. Why are stubs used in RMI?
15. Why is the function or role of skeleton in RMI?
16. How dynamic class loading does happens in RMI?

6.	Implement Bully and Ring algorithm for leader election.	
7.	Create a simple web service and write any distributed application to consume the web service.	
8.	Mini Project (In group): A Distributed Application for Interactive Multiplayer Games	

Assignment No: 2

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

Problem Statement: Development of distributed application using CORBA to demonstrate object brokering

Course Outcome:

CO Number	Applicable CO	Blooms Taxonomy Category

Pre-requisites: Computer Network Technology

Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

A. For Prerequisites –

B. For Assignment -

Requirements: JavaIDL, a core package of JDK1.3+

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.

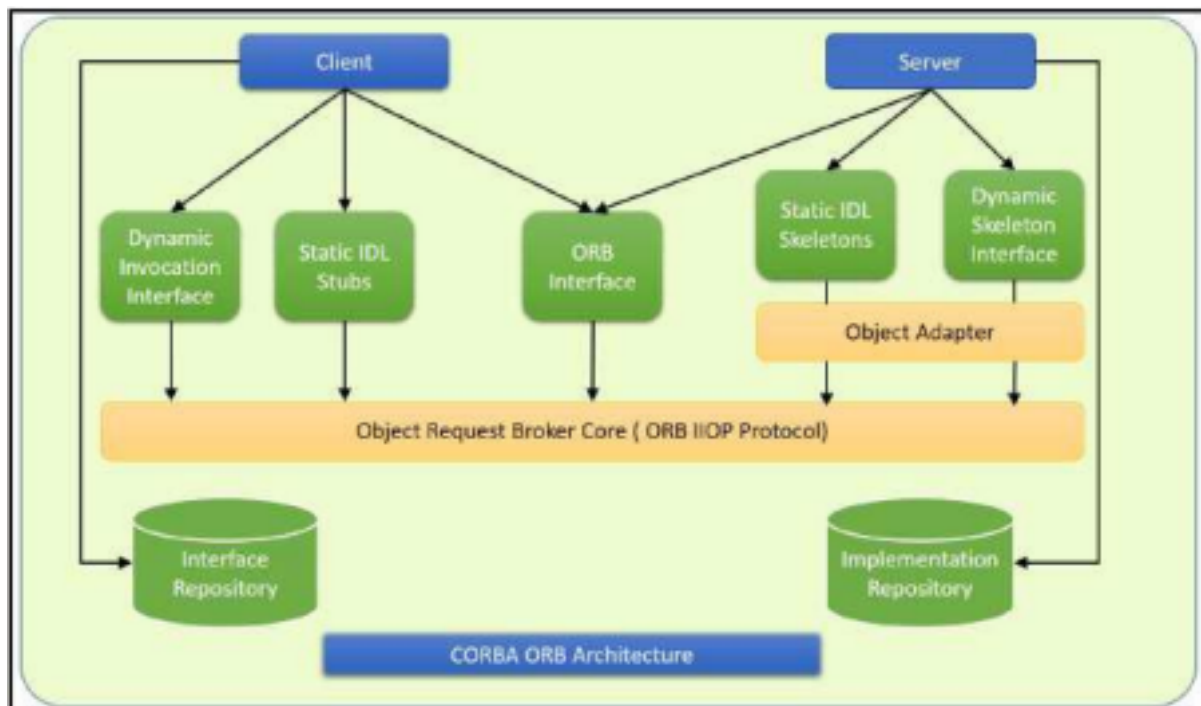
Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR&Benefits" maintain an object model with details of the organization, employees with demographic information, benefits, payroll, and deductions.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

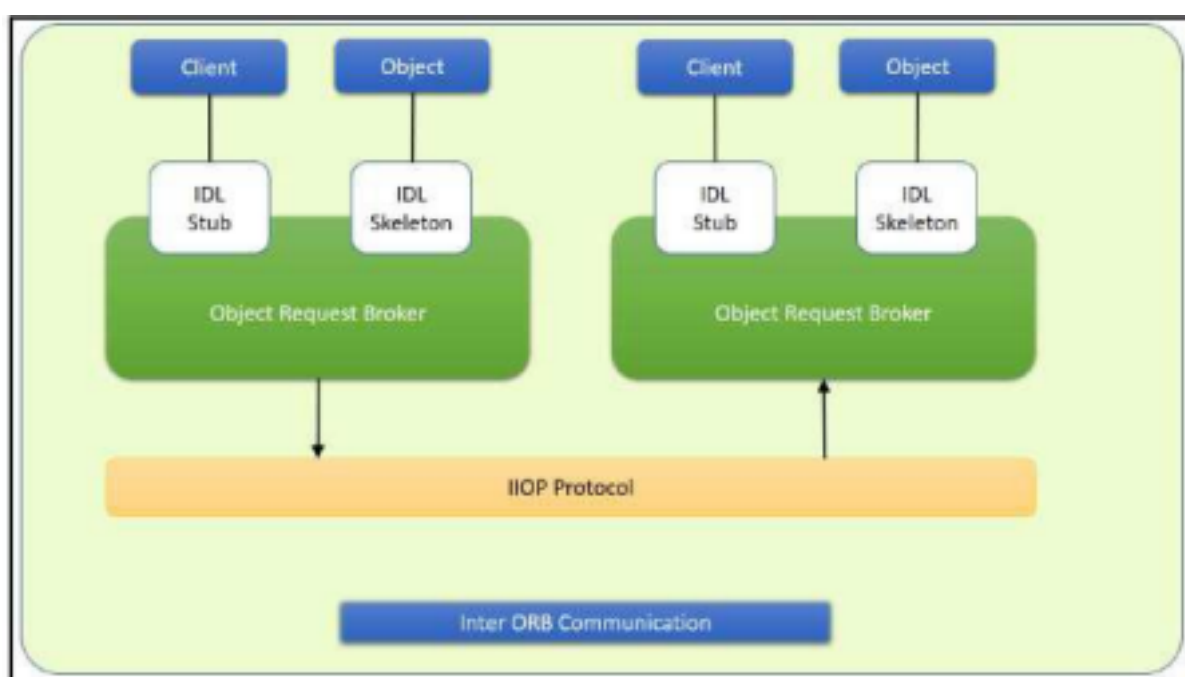
The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons. The objects are written (on the right) and a client for it (on the left), as represented in the diagram. The client and server use stubs and skeletons as proxies, respectively.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

Algorithm / Methods / Steps:

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:

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

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

Interfaces in IDL are declared much like interfaces in Java.

1.2. Modules

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

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

1.3. 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.

1.4 Data members and methods:

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

```
readonly attribute string myString;
```

The method can be declared by specifying its name, return type, and parameters, at a minimum. *string parseString(in string buffer);*

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

1.5 A complete IDL example:

Now let's tie all these basic elements together. Here's a complete IDL example 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-nameStub*, that acts as a client-side implementation of the interface and knows how to convert method requests into ORB requests that are forwarded to the actual remote object. The stub class for an interface named *Server* is called *_ServerStub*.
- A **server skeleton class**, called *_interface-nameImplBase*, that is a base class for a server side implementation of the interface. The base class can accept requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named *Server* is called *_ServerImplBase*.

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

Implementation :

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,

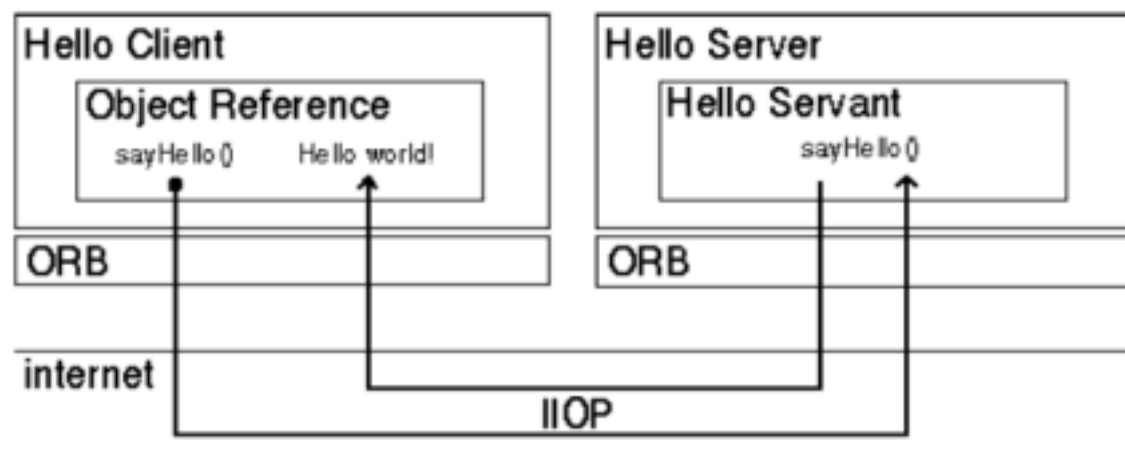
you simply provide the server (*HelloServer.java*) and client (*HelloClient.java*) implementations.

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.

Working:

Open ▾  ReverseModule.idl
~/Desktop/IDL CORBA

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

Open ▾  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);

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

```

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));
    }
}

```

```

ReverseServer.java
~/Desktop/IDL CORBA

import org.omg.CORBA.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 POA/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();
        }
    }
}

```

Execution:

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

- *HelloHolder.java*

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

- *HelloOperations.java*

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

3. Compile the *.java* files, including the stubs and skeletons (which are in the directory *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:

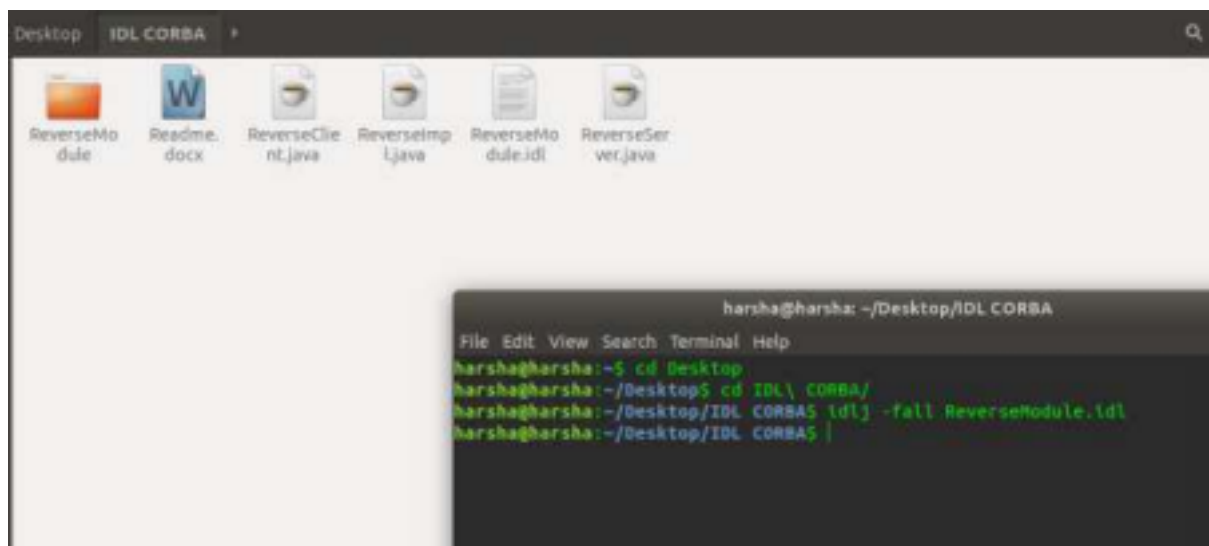
```
java HelloClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

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

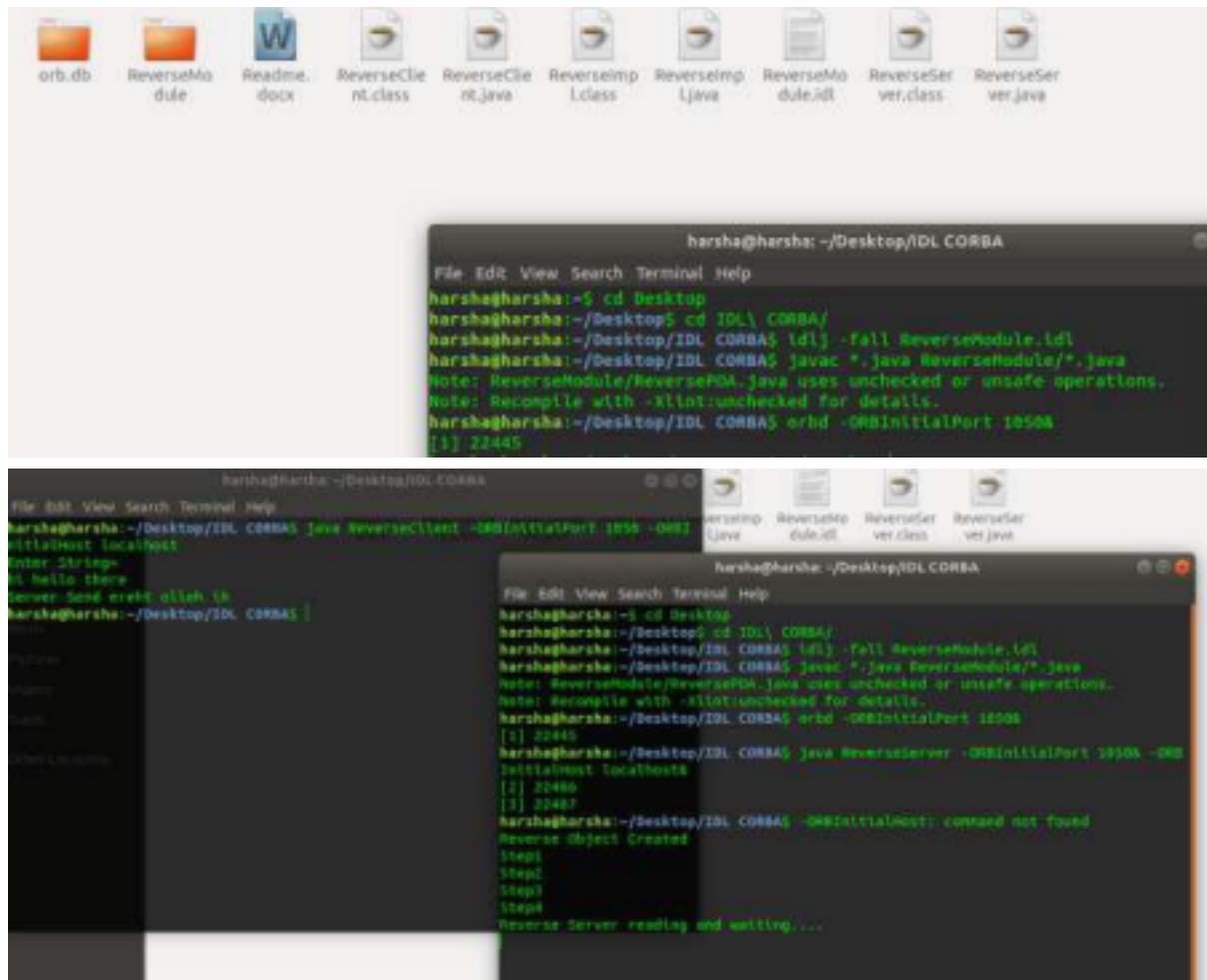
Obtained a handle on server object: IOR: (binary code)

Hello World! HelloServer exiting...

After completion kill the name server (orbd).



Input and Output:



Inference:

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.

Oral questions:

1. What is CORBA?
2. What does CORBA provide?
3. What does Java offer CORBA Programmers?
4. Which protocol is used for invoking methods on CORBA objects over the internet?
5. Name some of the CORBA development tools?
6. Explain Naming Service in CORBA?
7. What are the ORBlets?
8. What is an Object Implementation defined in ORB Architecture?
9. What is OMG IDL?
10. What is Call Back Mechanism?
11. What is Event Service in CORBA?
12. What is Callback Mechanism in CORBA?

13. What is the main difference between RMI and CORBA?
14. Draw the class hierarchy in CORBA application?
15. What does CORBA offer Java Programmers?
16. What types of event channel models does the Event Service provide?
17. Give the diagrammatic representation of Pull and Push models?
18. What is URL Naming Service?
19. Explain the Bootstrapping technique in RMI?
20. What are the CORBA services?
21. What are all the requirements needed to build a CORBA

program?

Sr. No.	Title	Page No.
1.	Implement multi-threaded client/server Process communication using RMI.	
2.	Develop any distributed application using CORBA to demonstrate object brokering. (Calculator or String operations).	
3.	Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.	
4.	Implement Berkeley algorithm for clock synchronization.	
5.	Implement token ring based mutual exclusion algorithm.	
6.	Implement Bully and Ring algorithm for leader election.	
7.	Create a simple web service and write any distributed application to consume the web service.	
8.	Mini Project (In group): A Distributed Application for Interactive Multiplayer Games	

Assignment No: 3

Title / Objective: MPI

Problem Statement: Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.

Course Outcome:

CO No.	CO Statement	BTL
C414454.1	Demonstrate knowledge of the core concepts and techniques in distributed systems.	Apply

Pre-requisites: Computer Network Technology

Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

A. For Prerequisites –

B. For Assignment -

Requirements: openmpi-4.1.4.tar.bz2.

Theory:

OpenMP:

OpenMP is an Application Program Interface (API) that may be used to explicitly direct *multi-threaded, shared memory parallelism* in C/C++ programs. It allows developers to easily write shared-memory parallel applications. A shared memory model runs on a single system and utilizes its multiple processing units or cores to achieve concurrent computation using the same memory space. OpenMP provides high-level compiler directives.

It is not intrusive on the original serial code in that the OpenMP instructions are made in pragmas interpreted by the compiler.

OpenMP uses the fork-join model of parallel execution. All OpenMP programs begin with a single master thread which executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK). When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN).

This includes `#pragma omp parallel`, used to identify which parts of the code need to run concurrently `#pragma omp critical` to identify critical sections. It then automatically deals with many of the low-level complications like handling threads (each core's line of execution) and locks (constructs used to avoid problems like race and deadlock). It even handles reductions and compiles the results of each thread into a final answer.

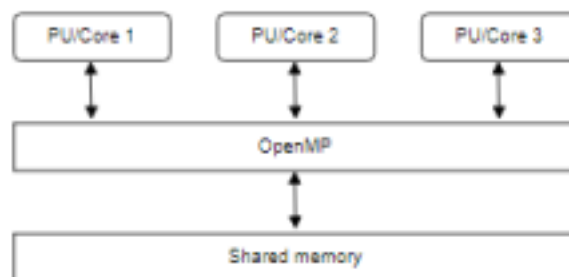


Fig – Open MP Structure

Algorithm / Methods / Steps:

1. Download openmpi-4.1.4.tar.bz2 from <http://www.open-mpi.org> in a folder say LP5.
2. Goto the terminal (Command prompt)
3. Update using

```
sudo apt-get update
sudo apt install gcc {if not already installed}
```
4. Goto the directory which contains the downloaded file
5. Extract the files using

```
tar -jxf openmpi-4.1.4.tar.bz2
```
6. The directory openmpi-4.1.4 is created
7. Configure, compile and install by executing the following commands

```
./configure --prefix=$HOME/opt/openmpi
make all
make install
```
8. Now openmpi folder is created in 'opt' folder of Home directory.
9. Now the folder LP5 can be deleted (optional)
10. Update the PATH and LD_LIBRARY_PATH environment variable using

```
echo "export PATH=$PATH:$HOME/opt/openmpi/bin" >> $HOME/.bashrc
echo "export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/opt/openmpi/lib" >> $HOME/.bashrc
```


11. Compile the program using
mpicc name of the program
12. Execute the program using
mpirun -np N ./a.out

Implementation and Execution:

Hello world program

nllabc2d22@nllabc2d-22:~/opt/openmpi/bin\$ gedit hello.c

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
    int rank, size, len;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello, world, I am %d of %d\n",rank, size);
    MPI_Finalize();
    return 0;
}
```

Compile the program

```
nllabc2d22@nllabc2d-22:~/opt/openmpi/bin$ mpicc hello.c
```

Execute the program using 2 cores

```
nllabc2d22@nllabc2d-22:~/opt/openmpi/bin$ mpirun -np 2 ./a.out
```

```
Hello, world, I am 0 of 2
```

```
Hello, world, I am 1 of 2
```

Execute the program using 4 cores

```
nllabc2d22@nllabc2d-22:~/opt/openmpi/bin$ mpirun -np 4 ./a.out
```

```
Hello, world, I am 0 of 4
```

```
Hello, world, I am 3 of 4
```

```
Hello, world, I am 1 of 4
```

```
Hello, world, I am 2 of 4
```

Program to transfer data from core 0 to core 1.

```
#include <stdio.h>
```

```
#include "mpi.h"
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    int rank, size, len;
```

```
    int num=10;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
    if(rank == 0)
```

```
    {
```

```
        printf("Sending message containing: %d from rank %d\n", num, rank);
```

```
        MPI_Send(&num, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
```

```
    }
```

```
    else
```

```
    {
```

```
        printf(" at rank %d\n", rank);
```

```
        MPI_Recv(&num, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
        printf("Received message containing: %d at rank %d\n", num, rank);
```

```
    }
```

```
    MPI_Finalize();
```

```
    return 0;
```

```
}
```

Sending message containing: 10 from rank 0

at rank 1

at rank 3

Received message containing: 10 at rank 1

at rank 2

*/****** The cores 2 and will be in waiting mode ... Press Ctrl+C to end the execution *****/*

Assignment program: Add 20 numbers in an array using 4 cores

```
#include <stdio.h>
```

```
#include "mpi.h"
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    int rank, size;
```

```
    int num[20]; //N=20, n=4
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
    for(int i=0; i<20; i++)
```

```
        num[i]=i+1;
```

```
    if(rank == 0){
```

```
        int s[4];
```

```
        printf("Distribution at rank %d \n", rank);
```

```
        for(int i=1; i<4; i++)
```

```
            MPI_Send(&num[i*5], 5, MPI_INT, i, 1, MPI_COMM_WORLD); //N/n i.e. 20/4=5
```

```
        int sum=0, local_sum=0;
```

```
        for(int i=0; i<5; i++)
```

```
        {
```

```
            local_sum=local_sum+num[i];
```

```
        }
```

```
        for(int i=1; i<4; i++)
```

```
        {
```

```
            MPI_Recv(&s[i], 1, MPI_INT, i, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
        }
```

```
        printf("local sum at rank %d is %d\n", rank, local_sum);
```

```
        sum=local_sum;
```

```
        for(int i=1; i<4; i++)
```

```
            sum=sum+s[i];
```

```
        printf("final sum = %d\n\n", sum);
```

```
    }
```

```

else
{
    int k[5];
    MPI_Recv(k, 5, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    int local_sum=0;
    for(int i=0;i<5;i++)
    {
        local_sum=local_sum+k[i];
    }
    printf("local sum at rank %d is %d\n", rank, local_sum);
    MPI_Send(&local_sum, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
MPI_Finalize();

return 0;
}

```

Proposed output:

```

Distribution at rank 0
local sum at rank 1 is 40
local sum at rank 2 is 65
local sum at rank 3 is 90
local sum at rank 0 is 15
final sum = 210

```

Inference:

A distributed system is developed, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP.

Oral questions:

1. What is MPI?
2. What is OpenMP?
3. What is difference between OpenMP and OpenMPI?
4. How can OpenMP threads permanently allocate memory on GPUs?
5. Is OpenMP multiprocessing or multithreading?
6. Is OpenMP parallel or concurrent?
7. How many threads can OpenMP use?
8. What are the different types of synchronization in OpenMP?
9. Does OpenMP use multiple cores?
10. What is the advantage of OpenMP?
11. How many cores does OpenMP use?
12. What is default in OpenMP?

13. What is the limit of threads?
14. How many threads can be active?
15. Can OpenMP be used more than one node?
16. Is OpenMP a compiler?
17. Which compiler supports OpenMP?
18. Is OpenMP shared memory or distributed memory?
19. How many processors does OpenMP have?
20. What is thread in OpenMP?
21. What is master thread in OpenMP?
22. What are the disadvantages of OpenMP?
23. What is the difference between thread and process in OpenMP?
24. What are the primary components of OpenMP?

Sr. No.	Title	Page No.
1.	Implement multi-threaded client/server Process communication using RMI.	
2.	Develop any distributed application using CORBA to demonstrate object brokering. (Calculator or String operations).	
3.	Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.	
4.	Implement Berkeley algorithm for clock synchronization.	
5.	Implement token ring based mutual exclusion algorithm.	
6.	Implement Bully and Ring algorithm for leader election.	
7.	Create a simple web service and write any distributed application to consume the web service.	
8.	Mini Project (In group): A Distributed Application for Interactive Multiplayer Games	

Assignment No: 4

Title / Objective: Berkeley Algorithm

Problem Statement: Implement Berkeley algorithm for clock synchronization.

Course Outcome:

CO No.	CO Statement	BTL
C414454.2	Learn how to apply principles of state-of-the-Art Distributed systems in practical application	Apply

Pre-requisites: Computer Network Technology

Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

A. For Prerequisites –

B. For Assignment -

Requirements: openmpi-4.1.4.tar.bz2.

Theory:

Berkeley Algorithm:

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

There are several variations of Berkeley's Algorithm that have been proposed, but a common version of the algorithm is as follows –

- Each computer starts with its own local time, and periodically sends its time to the master computer.
- The master computer receives timestamps from all the computers in the network. • The master computer computes the average time of all the timestamps it has received and sends the average time back to all the computers in the network.
- Each computer sets its clock to the time it receives from the master computer. • The process is repeated periodically, so that over time, the clocks of all the computers in the network will converge to the correct time.

One benefit of Berkeley's Algorithm is that it is relatively simple to implement and understand. However, It has a limitation that the time computed by the algorithm is based on the network conditions and time of sending and receiving timestamps which can't be very accurate and also it has a requirement of a master computer which if failed can cause the algorithm to stop working.

There is other more advance algorithm such as the Network Time Protocol (NTP) which use a more complex algorithm and also consider the network delay and clock drift to get a more accurate time.

Features of Berkeley's Algorithm:

Centralized time coordinator: Berkeley's Algorithm uses a centralized time coordinator, which is responsible for maintaining the global time and distributing it to all the client machines. **Clock adjustment:** The algorithm adjusts the clock of each client machine based on the difference between its local time and the time received from the time coordinator. **Average calculation:** The algorithm calculates the average time difference between the client machines and the time coordinator to reduce the effect of any clock drift. **Fault tolerance:** Berkeley's Algorithm is fault-tolerant, as it can handle failures in the network or the time coordinator by using backup time coordinators.

Accuracy: The algorithm provides accurate time synchronization across all the client machines, reducing the chances of errors due to time discrepancies.

Scalability: The algorithm is scalable, as it can handle a large number of client machines, and the time coordinator can be easily replicated to provide high availability. **Security:** Berkeley's Algorithm provides security mechanisms such as authentication and encryption to protect the time information from unauthorized access or tampering.

To use Berkeley's Algorithm, one needs to implement the algorithm on each computer in a network of computers. Here is a general overview of the steps you would need to take to implement the algorithm:

- Designate one computer in the network as the master computer. This computer will be responsible for receiving timestamps from all the other computers in the network and computing the correct time.
- On each computer in the network, set up a timer to periodically send the computer's local time to the master computer.
- On the master computer, set up a mechanism for receiving timestamps from all the computers in the network.
- On the master computer, implement the logic for calculating the average time based on the received timestamps.
- On the master computer, set up a mechanism for sending the calculated average time back to all the computers in the network.
- On each computer in the network, set up a mechanism for receiving the time from the master computer and setting the computer's clock to that time.
- Repeat the process periodically, for example, every 30 seconds or 1 minute, to ensure that the clocks in the network stay synchronized.

It's worth noting that this is a high-level description of the steps to implement the algorithm and in practice, many implementation details will depend on the programming language, operating system, and network infrastructure you are using. Additionally, as explained before, Berkeley Algorithm has some limitations. If you need to have a more accurate and robust solution, you may consider using other time synchronization protocols like NTP which have been designed to overcome some of the limitations of Berkeley's algorithm.

Algorithm / Methods / Steps:

- 1) An individual node is chosen as the master node from a pool of nodes in the network. This node is the main node in the network which acts as a master and the rest of the nodes act as slaves. The master node is chosen using an election process/leader election algorithm.
- 2) Master node periodically pings slave nodes and fetches clock time at them using Cristian's algorithm.
- 3) Master node calculates the average time difference between all the clock times received and the clock time given by the master's system clock itself. This average time difference is added to the current time at the master's system clock and broadcasted over the network.

Implementation and Execution:

Server Program:

```
# Python3 program imitating a clock server

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

```

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

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

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

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

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

        time.sleep(5)

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

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

        print(slave_address + " got connected successfully")

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

        current_thread.start()

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

    current_client_data = client_data.copy()

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

    sum_of_clock_difference = sum(time_difference_list, \
                                                                    datetime.timedelta(0, 0))

    average_clock_difference = sum_of_clock_difference \
                                                                    / len(client_data)

    return average_clock_difference

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

    while True:

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

        if len(client_data) > 0:

            average_clock_difference = getAverageClockDiff()

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

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

```

```

        except Exception as e:
            print("Something went wrong while " + \
                  "sending synchronized time " + \
                  "through " + str(client_addr))

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

    print("\n\n")

    time.sleep(5)

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

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

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

    master_server.bind(('', port))

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

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

    master_thread.start()

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

    sync_thread.start()

# Driver function
if __name__ == '__main__':

    # Trigger the Clock Server
    initiateClockServer(port = 8080)

```

Client Program:

```

# Python3 program imitating a client process

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

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

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

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

        time.sleep(5)

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

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

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

```

```

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

    slave_client = socket.socket()

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

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

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

# Driver function
if __name__ == '__main__':

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

```

Proposed output:

Client Output

New synchronization cycle started.
 Number of clients to be synchronized: 3

Client Data updated with: 127.0.0.1:57284

Client Data updated with: 127.0.0.1:57274

Client Data updated with: 127.0.0.1:57272

Server Output

Recent time sent successfully
 Synchronized time at the client is: 2023-04-17 15:49:31.166449

Inference:

The Berkeley algorithm is implemented for clock synchronization.

Oral questions:

1. What is Berkeley Algorithm?
2. What is importance of Berkeley Algorithm?
3. How Berkeley Algorithm is useful?

4. How clock synchronization is achieved through Berkeley algorithm?
5. Which algorithm is used for clock synchronization?
6. How does the Berkeley algorithm achieve fault tolerant average and give better synchronization of time?
7. What is the difference between Berkeley and Cristian's algorithm?
8. What is the function of clock synchronization?
9. What is the accuracy of clock synchronization?
10. What are the two methods used for time synchronization?
11. What is an example of clock synchronization?
12. What is Berkeley algorithm used for?
13. Is Berkeley algorithm active or passive?
14. What is the formula for propagation time for Berkeley algorithm?
15. What are the benefits of clock synchronization?

Course Objectives

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

Course Outcomes

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

INDEX

Sr. No.	Title	Page No.
1.	Implement multi-threaded client/server Process communication using RMI.	
2.	Develop any distributed application using CORBA to demonstrate object brokering. (Calculator or String operations).	
3.	Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.	
4.	Implement Berkeley algorithm for clock synchronization.	
5.	Implement token ring based mutual exclusion algorithm.	
6.	Implement Bully and Ring algorithm for leader election.	
7.	Create a simple web service and write any distributed application to consume the web service.	
8.	Mini Project (In group): A Distributed Application for Interactive Multiplayer Games	

Assignment No: 5

Title / Objective: Implement token ring based mutual exclusion algorithm.

Problem Statement: To implement token ring based mutual exclusion algorithm.

Course Outcome:

CO No.	CO Statement	BTL
C414454.2	Learn how to apply principles of state-of-the-Art Distributed systems in practical application	Apply

Pre-requisites: Computer Network Technology

Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

A. For Prerequisites –

B. For Assignment -

Requirements: Java Programming Environment, JDK 8

Theory:

Distributed Mutual exclusion:

Distributed Mutual exclusion is a concurrency control property, which is introduced to prevent race conditions. It is the requirement that a process cannot enter its critical section while another concurrent process is currently present or executing in its critical section i.e. only one process is allowed to execute the critical section at any given instance of time.

In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.

In Distributed systems, we have shared neither memory nor a common physical clock and there for we cannot solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.

A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.

Requirements of Mutual exclusion Algorithm:

- **No Deadlock:**
Two or more site should not endlessly wait for any message that will never arrive.
- **No Starvation:**
Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section
- **Fairness:**
Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e. critical section execution requests should be executed in the order of their arrival in the system.
- **Fault Tolerance:**
In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

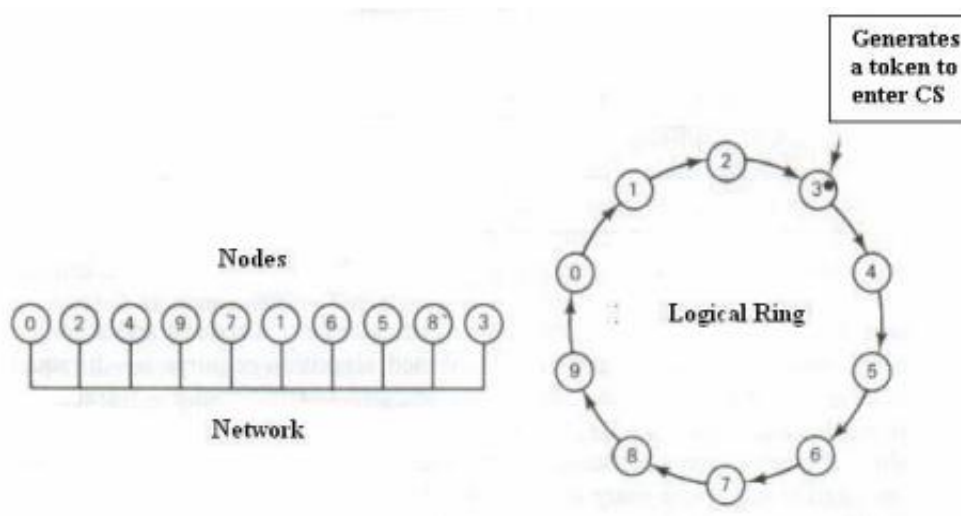
Solution to distributed mutual exclusion:

As we know shared variables or a local kernel cannot be used to implement mutual exclusion in distributed systems. Message passing is a way to implement mutual exclusion. Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:

1. Token Based Algorithm:

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.

- A unique **token** is shared among all the sites.
- If a site possesses the unique token, it is allowed to enter its critical section
- This approach uses sequence number to order requests for the critical section.
- Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
- This approach insures Mutual exclusion as the token is unique.



2. **Non-token based approach:**

- A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
- This approach use timestamps instead of sequence number to order requests for the critical section.
- Whenever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm, which follows non-token based approach, maintains a logical clock. Logical clocks get updated according to Lamport's scheme
- **Example:**
- Lamport's algorithm, Ricart–Agrawala algorithm

3. **Quorum based approach:**

- Instead of requesting permission to execute the critical section from all other sites, each site requests only a subset of sites, which is called a **quorum**.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion
- **Example:**
- Maekawa's Algorithm

Algorithm / Methods / Steps:

The algorithm works as follows:

1. When the ring is initialized, process 0 is given a token.
2. The token circulates around the ring. .
3. 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
4. After it has exited, it passes the token to the next process in the ring.
5. It is not allowed to enter the critical region again using the same token.
6. 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.

Implementation and Execution:

```
Import java.io.*;

import java.util.*;

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 );
}
}

```

Proposed Output:

Enter the num of nodes:

6

0 1 2 3 4 5 0

Enter sender:

1

Enter receiver:

5

Enter Data:

20

Token passing: 0->1

Sender 1 sending data: 20

data 20 forwarded by 2

data 20 forwarded by 3

data 20 forwarded by 4

Receiver 5 received data: 20

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

Inference:

This assignment, described the Token Ring algorithm approach to handle mutual exclusion in distributed systems.

Oral questions:

1. What is Mutual Exclusion?
2. What are the different mutual exclusion algorithms available in distributed systems?
3. What are the requirements of mutual exclusion?
4. Classify Distributed mutual exclusion algorithm.
5. Differentiate between token based algorithms and non-token based algorithms.
6. What are different non-token based algorithm are there in distributed system?
7. What are different token based algorithm are there in distributed system?
8. What are the different performance measure of mutual exclusion algorithms?
9. Give comparative performance analysis of mutual exclusion algorithms.

Assignment No: 6

Title / Objective: Bully and Ring Algorithm

Problem Statement: Implement Bully and Ring algorithm for leader election.

Course Outcome:

CO No.	CO Statement	BTL
C414454.2	Learn how to apply principles of state-of-the-Art Distributed systems in practical application	Apply

Pre-requisites: Computer Network Technology

Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

A. For Prerequisites –

B. For Assignment -

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

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 its successor (if successor is down, sender skips over it and goes to the next member along the ring, or the one after that, until a running process is located).
4. At each step, sender adds its own process number to the list in the message.
5. When the message gets back to the process that started it all: Message comes back to initiator.

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

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

Algorithm / Methods / Steps:

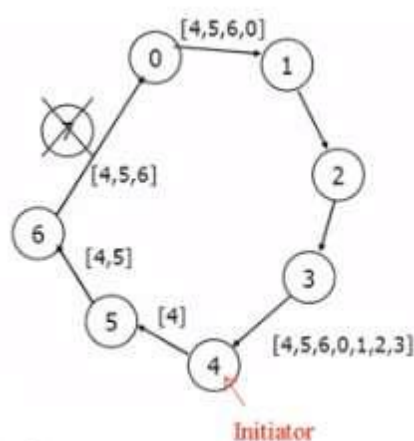
A. For Ring Algorithm

Initiation:

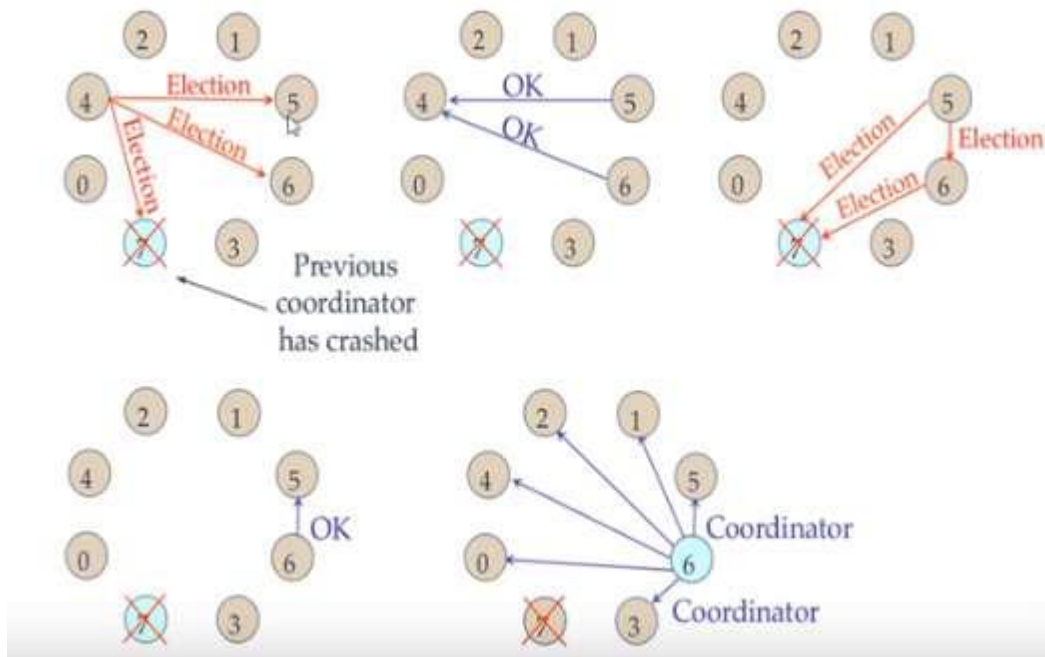
1. Consider the Process 4 understands that Process 7 is not responding.
2. Process 4 initiates the Election by sending "ELECTION" message to it's successor (or next alive process) with it's ID.

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
1) With highest process ID is 6. The initiator will announce that Process 6 is Coordinator.



B. For Bully Algorithm:



Implementation and Execution:

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.

Ring.java

The screenshot shows the Eclipse IDE with the file `Ring.java` open. The code implements a distributed election algorithm. The console output shows the execution flow: entering the number of processes (5), entering IDs (5, 6, 7, 8), selecting process 8 as the coordinator, and the program terminating.

```

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

```

Console Output:

```

<terminated> Ring(1) [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (28
Enter the number of process :
5
Enter the id of process :
5 6 7
Enter the id of process :
[0] 5 [1] 6 [2] 8
process 8select as co-ordinator
1.election 2.quit
Enter the Process number who initialised election :
2
Process 8 send message to 5
Process 5 send message to 6
Process 6 send message to 8
process 8select as co-ordinator
1.election 2.quit
Program terminated ...

```

Bully.java

The screenshot shows the Eclipse IDE with the file `Bully.java` open. The code implements a Bully algorithm for distributed election. The console output shows the election process: process 5 becomes the coordinator, and messages are sent to other processes.

```

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

```

Console Output:

```

Bully(1) [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java (28-Dec-2018, 7:3
5 active process are:
Process up = p1 p2 p3 p4 p5
Process 5 is coordinator
.....
1 up a process.
2 down a process
3 send a message
4.Exit
bring down any process.
5
.....
1 up a process.
2 down a process
3 send a message
4.Exit
which process will send message
2
process2election
election send from process2to process 3
election send from process2to process 4
election send from process2to process 5
Coordinator message send from process4to all
.....
1 up a process.
2 down a process
3 send a message
4.Exit

```


Inference:

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 processes are unidirectional and every process can message to the process on its right only

Oral questions:

1. What is Election Algorithm?
2. Name different election algorithms?
3. What is Bully and Ring Algorithm?
4. What election algorithm does?
5. Why election algorithms are normally needed in a distributed system?
6. What is leader election algorithm and why do we need this algorithm?
7. How many types of messages are there in election algorithm?
8. What are the features required for election algorithms?
9. What is the purpose of election system?
10. What is the time complexity of leader election?

Assignment No: 7

Title / Objective: Distributed application to consume the web service

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

Course Outcome:

CO No.	CO Statement	BTL
C414454.3	Design, build and test application programs on distributed systems	Create

Pre-requisites: Computer Network Technology

Study Material: (Blogs / Videos / Courses / Web Sites / Books / e-Books)

A. For Prerequisites –

B. For Assignment -

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

Theory:

Web Service:

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

A service can be treated as a web service if:

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

Types of Web Services:

There are two types of web services:

1. **SOAP:** SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it's XML based, it's platform and language independent. So, our server can be based on JAVA and client can be on .NET, PHP etc. and vice versa.
2. **REST:** REST (Representational State Transfer) is an architectural style for developing web services. It's getting popularity recently because it has small learning curve when compared to SOAP. Resources are core concepts of Restful web services and they are uniquely identified by their URIs.

Web service architectures:

As part of a web service architecture, there exist three major roles.

Service Provider is the program that implements the service agreed for the web service and exposes the service over the internet/intranet for other applications to interact with.

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

Service Registry acts as the directory to store references to the web services.

The following are the steps involved in a basic SOAP web service operational behavior:

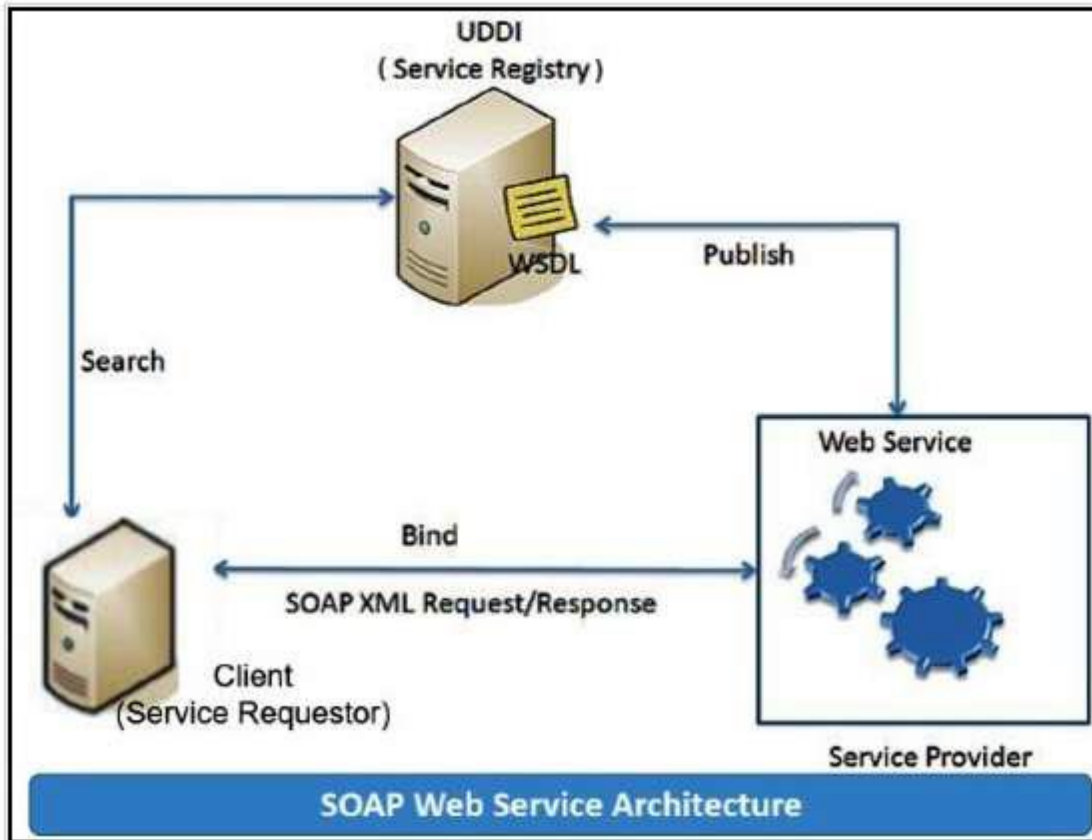
1. The client program that wants to interact with another application prepares its request content as a SOAP message.
2. Then, the client program sends this SOAP message to the server web service as an HTTP POST request with the content passed as the body of the request.
3. The web service plays a crucial role in this step by understanding the SOAP request and converting it into a set of instructions that the server program can understand.
4. The server program processes the request content as programmed and prepares the output as the response to the SOAP request.
5. Then, the web service takes this response content as a SOAP message and reverts to the SOAP HTTP request invoked by the client program with this response.
6. The client program web service reads the SOAP response message to receive the outcome of the server program for the request content it sent as a request.

SOAP web services:

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

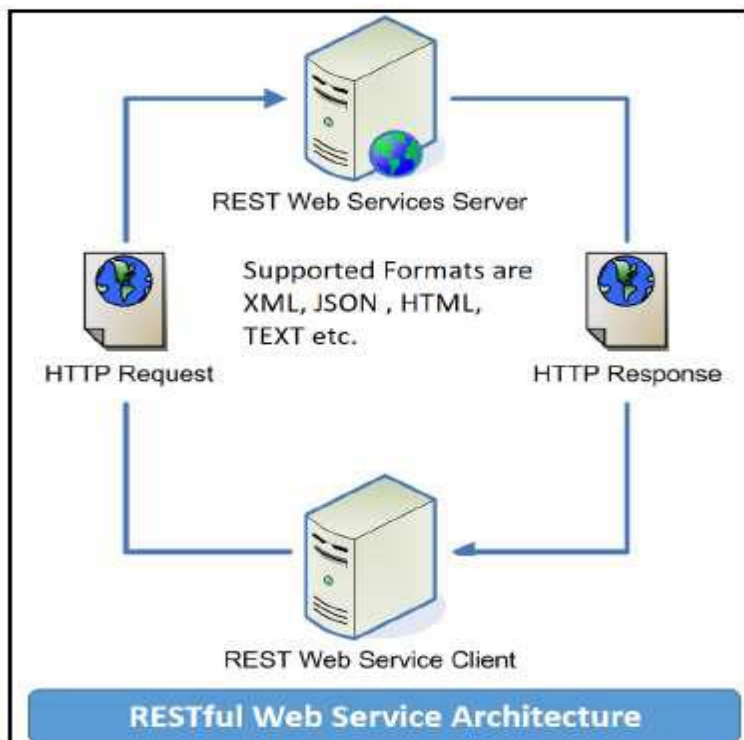
Universal Description, Discovery, and Integration (UDDI): UDDI is an XML based framework for describing, discovering, and integrating web services. It acts as a directory of web service interfaces described in the WSDL language.

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



RESTful web services

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



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

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

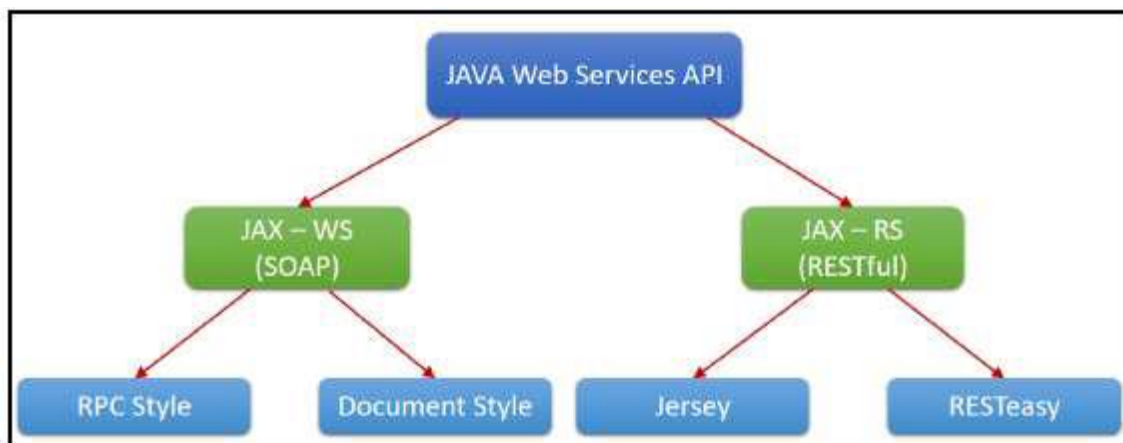
Algorithm / Methods / Steps:

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

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

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

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



Implementation and Execution:

1. Creating a web service CalculatorWSApplication:

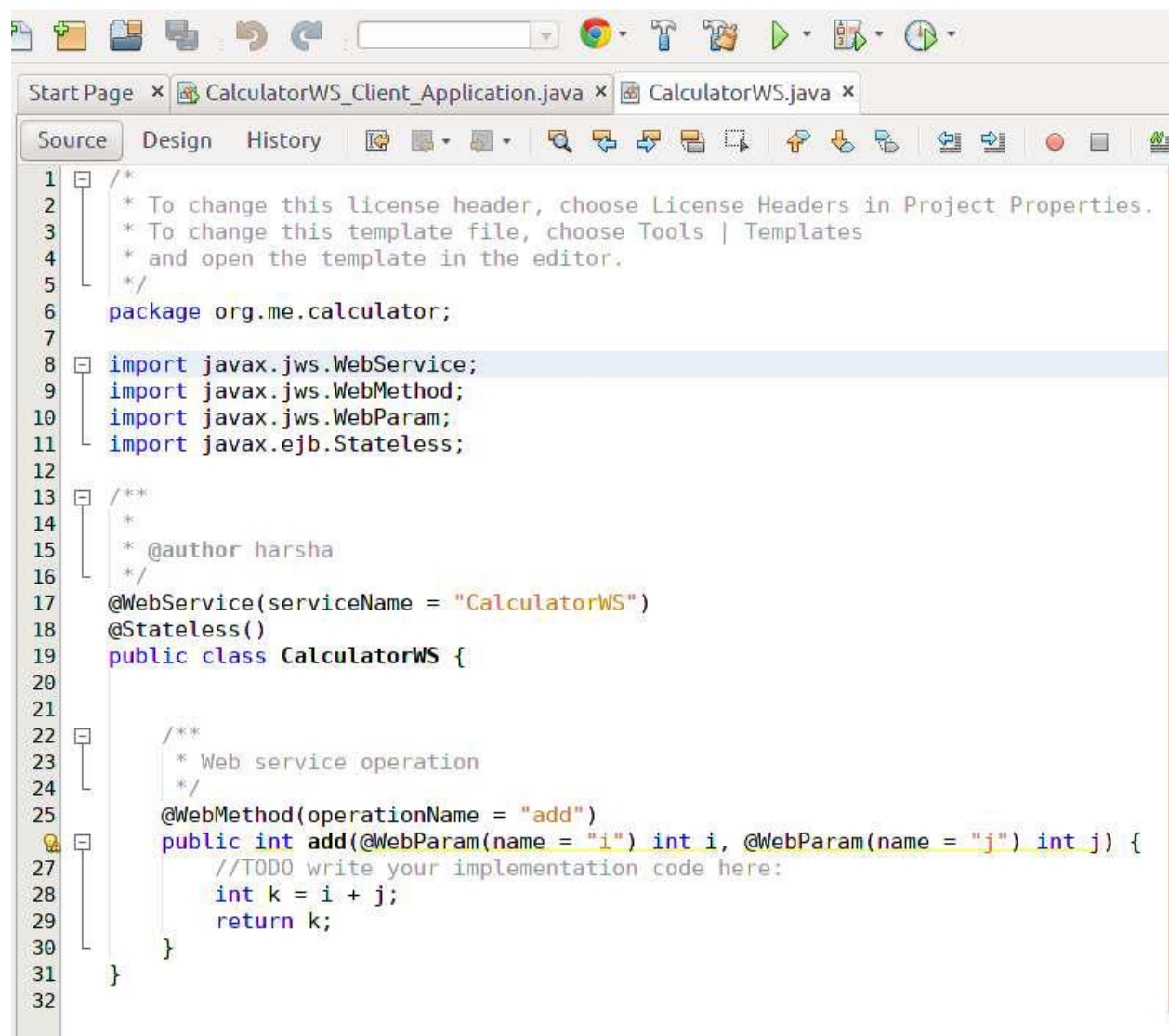
- Create New Project for CalculatorWSApplication.
- Create a package org.calculator
- Create class CalculatorWS.
- Right-click on the CalculatorWS and create New Web Service.
- IDE starts the glassfish server, builds the application and deploys the application on server.

2. Consuming the Webservice:

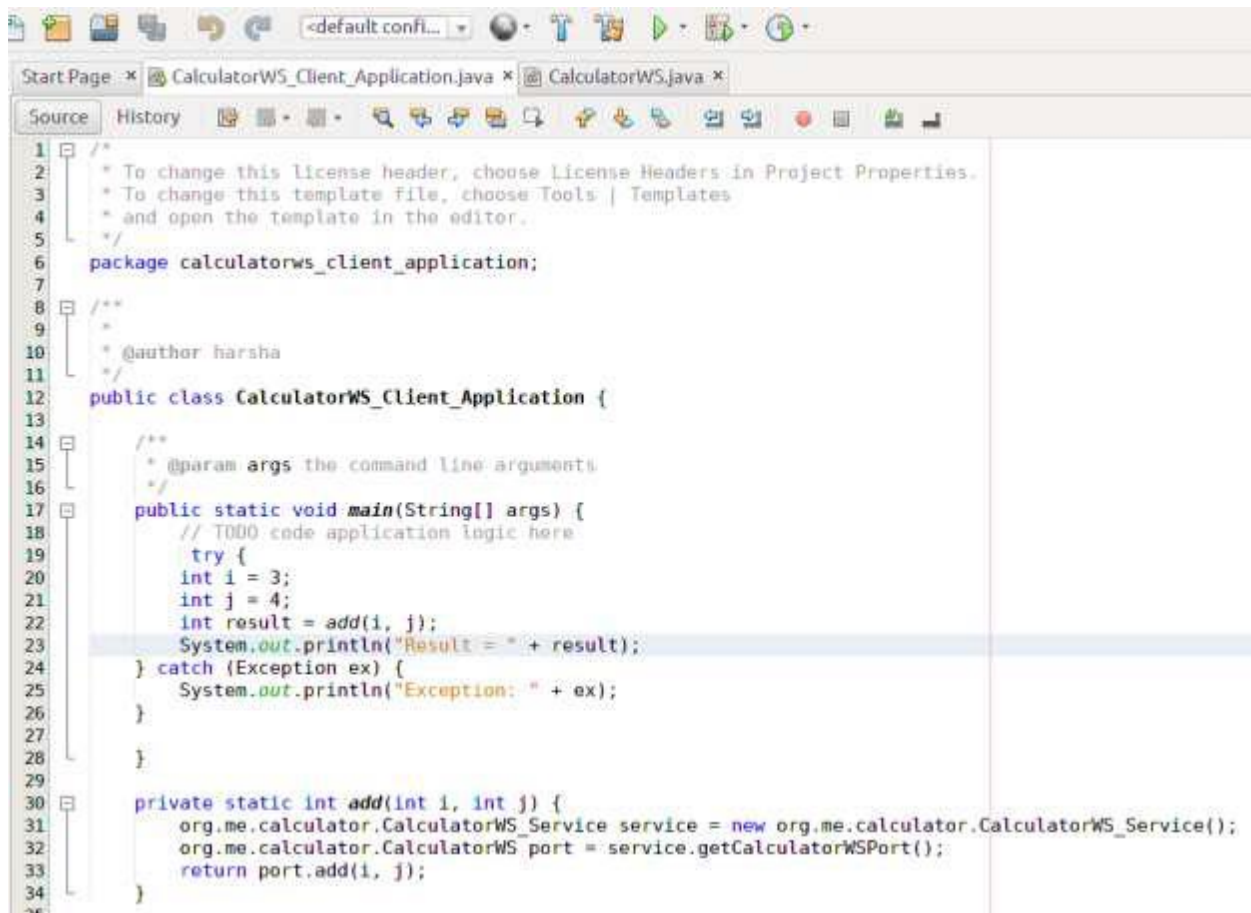
- Create a project with an CalculatorClient
- Create package org.calculator.client;
- add java class CalculatorWS.java, addresponse.java, add.java, CalculatorWSService.java and ObjectFactory.java

3. Creating servlet in web application

- Create new jsp page for creating user interface.

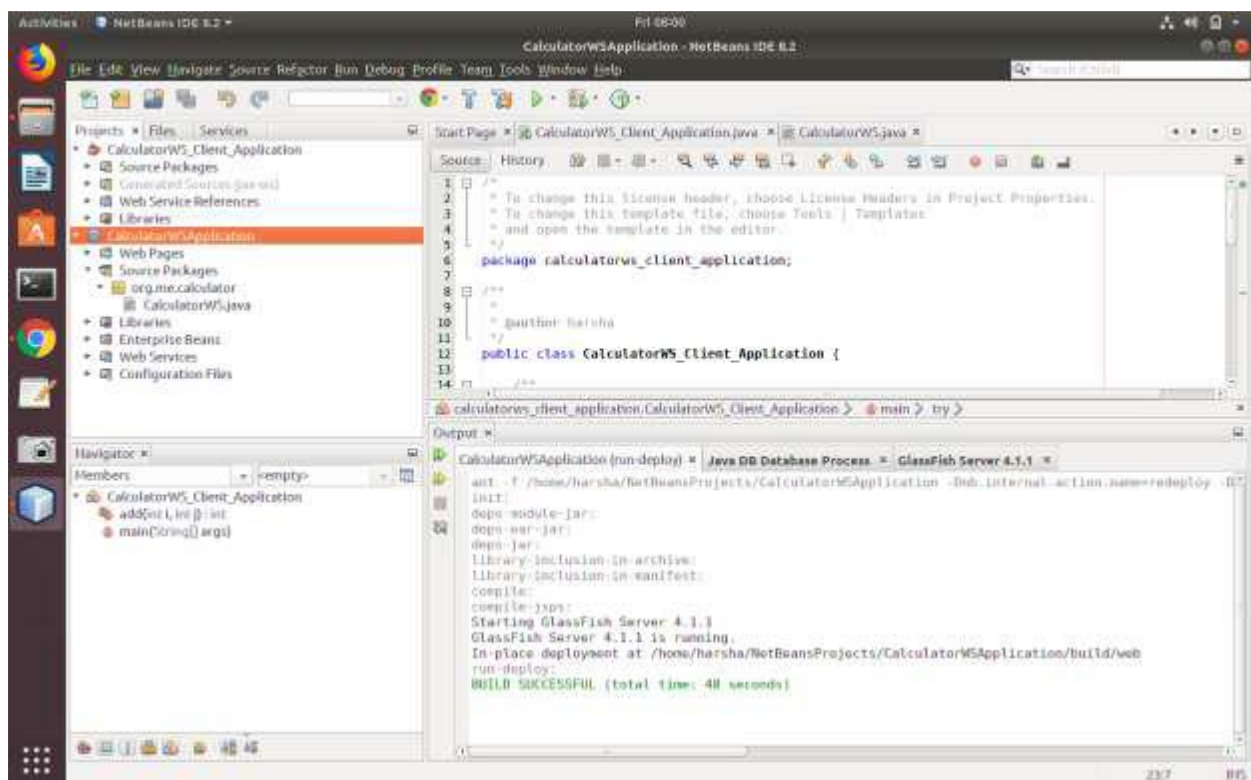


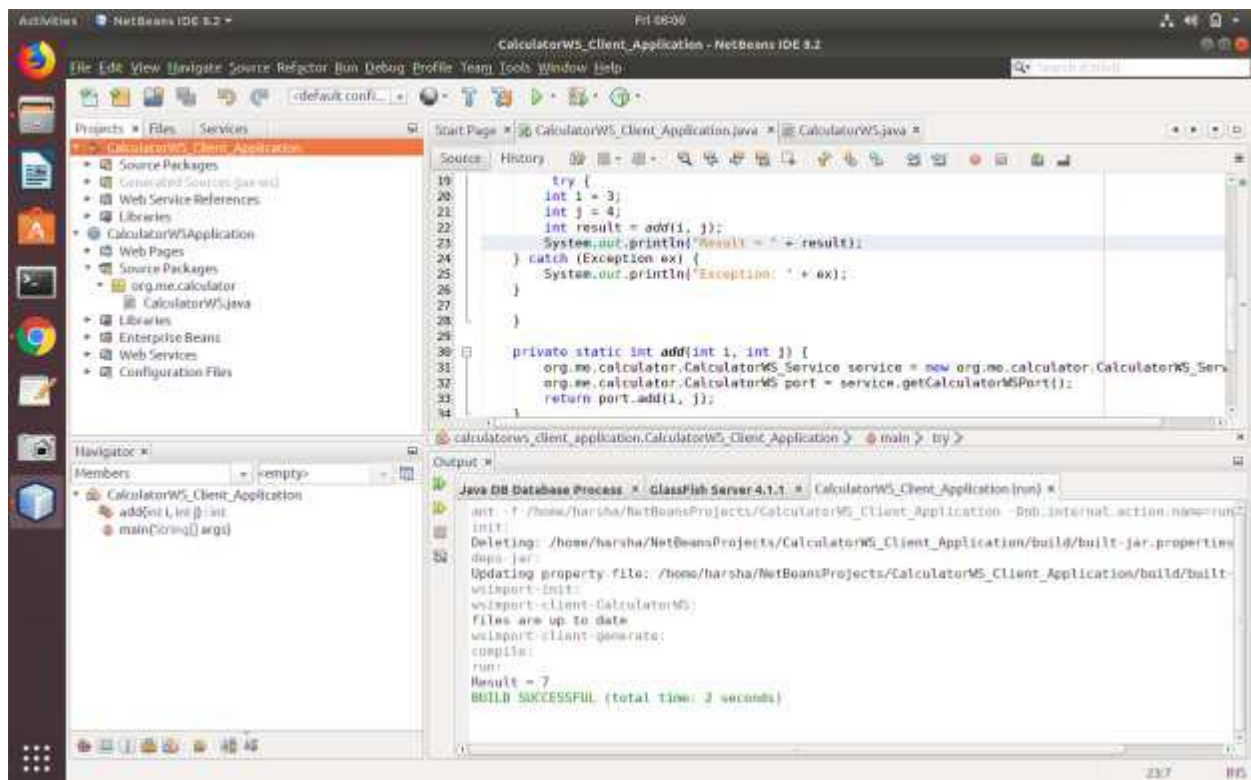
```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package org.me.calculator;
7
8  import javax.jws.WebService;
9  import javax.jws.WebMethod;
10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

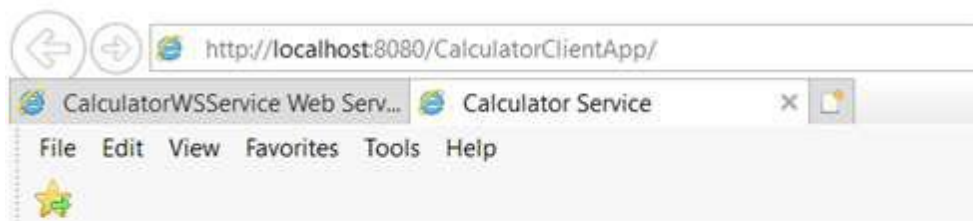
The screenshot shows the NetBeans IDE with the file `CalculatorWS_Client_Application.java` open. The code is as follows:

```
1  /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6  package calculatorws_client_application;
7
8  /**
9   *
10  * @author harsha
11  */
12  public class CalculatorWS_Client_Application {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          // TODO code application logic here
19          try {
20              int i = 3;
21              int j = 4;
22              int result = add(i, j);
23              System.out.println("Result = " + result);
24          } catch (Exception ex) {
25              System.out.println("Exception: " + ex);
26          }
27      }
28
29      private static int add(int i, int j) {
30          org.me.calculator.CalculatorWS_Service service = new org.me.calculator.CalculatorWS_Service();
31          org.me.calculator.CalculatorWS_port = service.getCalculatorWSPort();
32          return port.add(i, j);
33      }
34  }
```



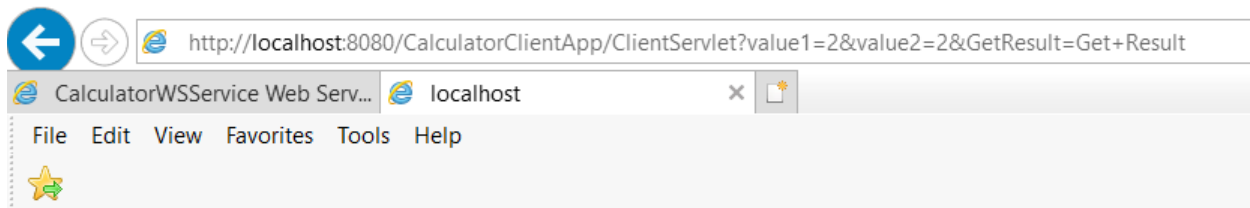


Proposed Output:



Calculator Service

+ =



Servlet ClientServlet at /CalculatorClientApp

Result: $2 + 2 = 4$

Inference:

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.

Oral questions:

1. What is Web Service?
2. What are different types of Web Services?
3. What is SOAP?
4. What is REST?
5. Explain Web Services Architecture?
6. Explain the concept of Service Provider.
7. Explain the concept of Service Requestor.
8. Explain the concept of Service Registry.
9. Differentiate SOAP and REST.