ASSIGNMENT NO. 1

Problem Statement:

To implement multi-threaded client/server Process communication using RMI.

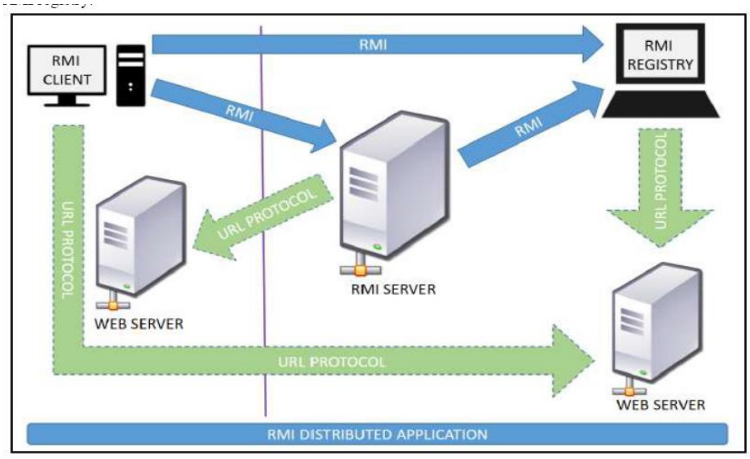
Tools / Environment:

Eclipse, Java 8, rmiregistry

Related Theory:

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

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



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

Key terminologies of RMI:

The following are some of the important terminologies used in a Remote Method Invocation.

Remote object: This is an object in a specific JVM whose methods are exposed so they could be invoked by another program deployed on a different JVM.

Remote interface: This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.

RMI: This is a way of invoking a remote object's methods with the help of a remote interface. It can be carried with a syntax that is similar to the local method invocation.

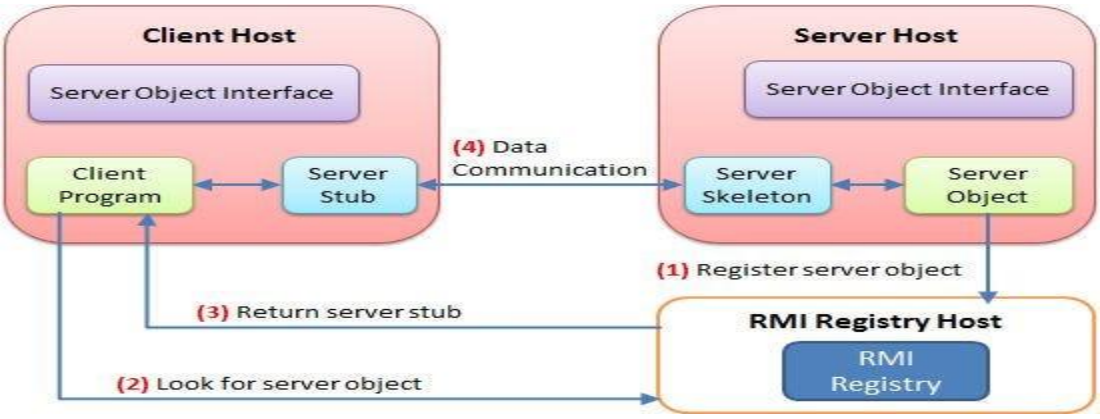
Stub: This is a Java object that acts as an entry point for the client object to route any outgoing requests. It exists on the client JVM and represents the handle to the remote object. If any object invokes a method on the stub object, the stub establishes RMI by following these steps:

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

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

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

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



Designing the solution:

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

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

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

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

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

Implementing the solution:
Steps to implement RMI:

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.

STEP 2: Implementing the remote interface
• To implement the remote interface, the class should extend to `UnicastRemoteObject` class of `java.rmi` package.

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 is rmic classname. 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 :

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.

Step 7: Compile and execute application programs:

Conclusion:

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

Submitted By: _____

Roll No.: _____

Class: B.E.I.T.

Staff Name: Ms. M. A. Rane

Staff Signature:

ASSIGNMENT NO. 2

Problem Statement:

To Develop any distributed application using CORBA to demonstrate object brokering.

(Calculator or String operations).

Tools / Environment:

Java Programming Environment, JDK 1.8

Related Theory:

Common Object Request Broker Architecture (CORBA):

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

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

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

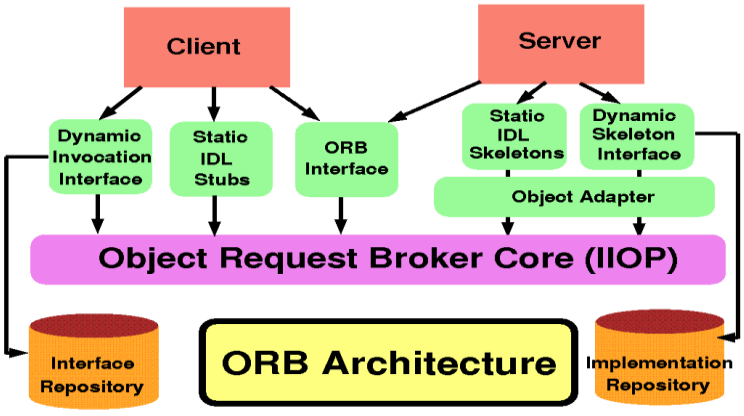
Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR&Benefits" maintain an object model with details of the organization, employees with demographic information, benefits, payroll, and deductions.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

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

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

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

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

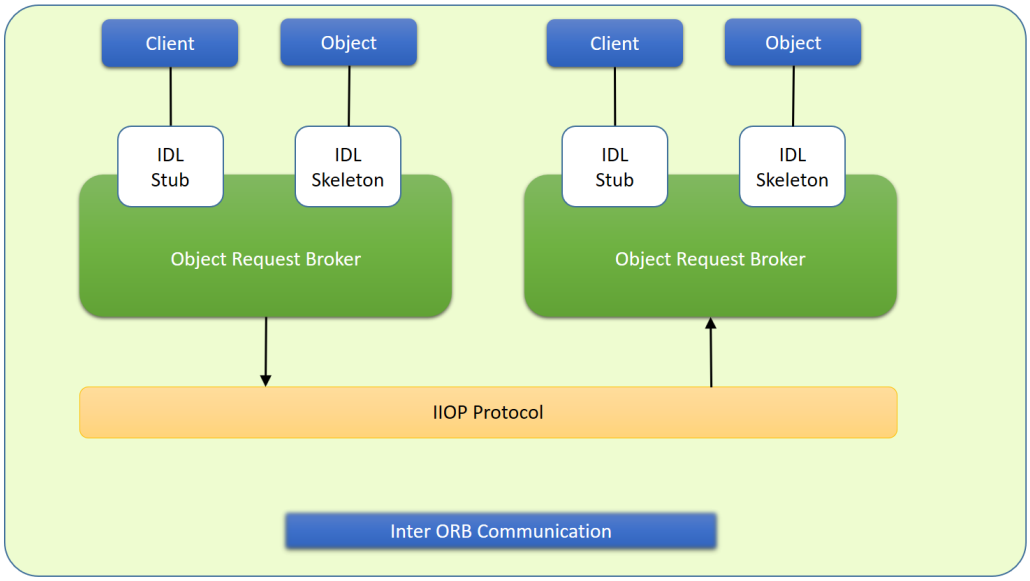


In CORBA, each object instance acquires an object reference for itself with the electronic token identifier. Client invocations are going to use these object references that have the ability to figure

out which ORB instance they are supposed to interact with. The stub and skeleton represent the client and server, respectively, to their counterparts. They help establish this communication through ORB and pass the arguments to the right method and its instance during the invocation.

Inter-ORB communication

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



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

Java Support for CORBA

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

CORBA standards provide the proven, interoperable infrastructure to the Java platform. IIOP (Internet Inter-ORB Protocol) manages the communication between the object components that power the system. The Java platform provides a portable object infrastructure that works on every major operating system. CORBA provides the network transparency, Java provides the

implementation transparency. **An *Object Request Broker (ORB)* is part of the Java Platform. The ORB is a runtime component that can be used for distributed computing using IIOP communication. Java IDL is a Java API for interoperability and integration with CORBA.**

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

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

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

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

The IDL Programming Model:

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

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

To use the IDL programming model, define remote interfaces using OMG Interface Definition Language (IDL), then compile the interfaces using `idlj` compiler. When you run the `idlj`

compiler over your interface definition file, it generates the Java version of the interface, as well as the class code files for the stubs and skeletons that enable applications to hook into the ORB.

Portable Object Adapter (POA) : An *object adapter* is the mechanism that connects a request using an object reference with the proper code to service that request. The Portable Object Adapter, or POA, is a particular type of object adapter that is defined by the CORBA specification. The POA is designed to meet the following goals:
· Allow programmers to construct object implementations that are portable between different ORB products.
· Provide support for objects with persistent identities.

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

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

1. Creating CORBA Objects using Java IDL:
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

Here's a complete IDL example that declares a module within another module, which itself contains several interfaces:

```
module OS {
     module services {
          interface Server {
               readonly attribute string serverName;
               boolean init(in string sName);
          };
          interface Printable {
               boolean print(in string header);
          };
          interface PrintServer : Server {
               boolean printThis(in Printable p);
          };
     };
};
```

The first interface, `Server`, has a single read-only `string` attribute and an `init()` method that accepts a `string` and returns a `boolean`. The `Printable` interface has a single

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

2. Turning IDL into Java

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

· A Java interface with the same name as the IDL interface. This can act as the basis for a Java implementation of the interface (but you have to write it, since IDL doesn't provide any details about method implementations).

· A *helper* class whose name is the name of the IDL interface with "Helper" appended to it (e.g., `ServerHelper`). The primary purpose of this class is to provide a static `narrow()` method that can safely cast CORBA `Object` references to the Java interface type. The helper class also provides other useful static methods, such as `read()` and `write()` methods that allow you to read and write an object of the corresponding type using I/O streams.

· A *holder* class whose name is the name of the IDL interface with "Holder" appended to it (e.g., `ServerHolder`). This class is used when objects with this interface are used as `out` or `inout` arguments in remote CORBA methods. Instead of being passed directly into the remote method, the object is wrapped with its holder before being passed. When a remote method has parameters that are declared as `out` or `inout`, the method has to be able to update the argument it is passed and return the updated value. The only way to guarantee this, even for primitive Java data types, is to force `out` and `inout` arguments to be wrapped in Java holder classes, which are filled with the output value of the argument when the method returns.

The `idltoj` tool generate 2 other classes:

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

· **A server *skeleton* class,** called *_interface-name*`ImplBase`, that is a base class for a server side implementation of the interface. The base class can accept requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named `Server` is called `_ServerImplBase`.

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

3. Writing the Implementation

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

Implementing the solution:

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

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

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

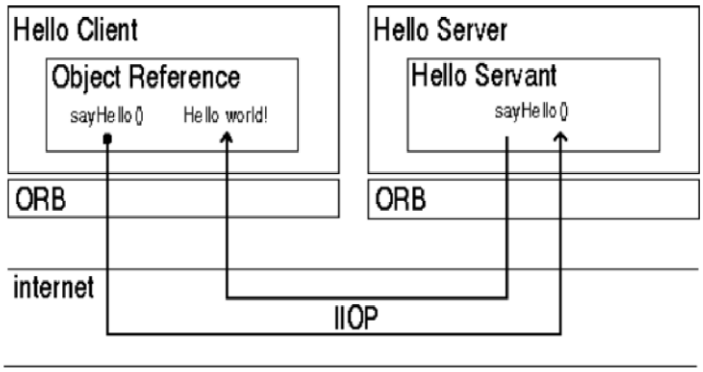
2. Implementing the Server `(HelloServer.java)`

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

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

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

· Registers the new object in the naming context under the name "Hello"
· Waits for invocations of the new object from the client.



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

The example application client that follows:

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

Building and Executing the solution:

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

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

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

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

idlj -fall Hello.idl

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

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

- `HelloPOA.java:`

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

- `HelloStub.java:`

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

- `Hello.java:`

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

Code for ReverseString:

ReverseServer.java

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

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

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

            // creating the calculator object
```

```
            ReverseImpl rvr = new ReverseImpl();

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

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

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

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

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

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

ReverseClient.java

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

class ReverseClient
{

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

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

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

            String name = "Reverse";
            ReverseImpl = ReverseHelper.narrow(ncRef.resolve_str(name));
```

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

            String tempStr= ReverseImpl.reverse_string(str);

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

ReverseImpl.java

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

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

ReverseModule.idl

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

Compiling and Executing:

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

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

idlj -fall ReverseModule.idl

The idlj compiler generates a number of files.

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

javac *.java ReverseModule/*.java

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

orbd -ORBInitialPort 1050&

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

java ReverseServer -ORBInitialPort 1050& -ORBInitialHost localhost&

6. Run the client application :

java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost

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).

Output:

Server:

```
~/LP5/CORBA$  idlj -fall  ReverseModule.idl
~/LP5/CORBA$ javac *.java  ReverseModule/*.java
Note: ReverseModule/ReversePOA.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for details.
~/LP5/CORBA$ orbd -ORBInitialPort 1050&
[1] 2769
~/LP5/CORBA$ java ReverseServer -ORBInitialPort 1050& -ORBInitialHost
localhost&
[2] 2797
[3] 2798
~/LP5/CORBA$ Reverse Object Created
Step1
Step2
Step3
-ORBInitialHost: command not found
Step4
Reverse Server reading and waiting....

[4]+  Exit 127                -ORBInitialHost localhost
~/LP5/CORBA$
```

Client:

```
~/LP5/CORBA$  java ReverseClient -ORBInitialPort 1050 -ORBInitialHost
localhost
Enter String=This is CORBA Program!
Server Send !margorP ABROC si sihT
~/LP5/CORBA$
```

Conclusion:

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

Submitted By: _____

Roll No.: _____

Class: B.E.I.T.

Staff Name: Ms. M. A. Rane/Mr. K. V. Patil

Staff Signature:

Date:

ASSIGNMENT NO. 3

Problem Statement:

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

Tools / Environment:
C++ Programming Environment, MPI Library (openmpi-4.1.4.)

Theory:

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

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

Message-Passing Interface Basics:

Group is the set of processes that communicate with one another.
Communicator is the central object for communication in MPI. There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**.

Every MPI program must contain the preprocessor directive:

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

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

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

`MPI_Finalize` cleans up all the extraneous mess that was first put into place by `MPI_Init`. The principal weakness of this limited form of processing is that the processes on different nodes run entirely independent of each other. It cannot enable capability or coordinated computing. **To get the different processes to interact, the concept of communicators is needed.** MPI programs are made up of concurrent processes executing at the same time that in almost all cases are also communicating with each other. To do this, an object called the "communicator" is provided by MPI. Thus the user may specify any number of communicators within an MPI program, each with its own set of processes. "**`MPI_COMM_WORLD`**" communicator contains all the concurrent processes making up an MPI program.

The size of a communicator is the number of processes that makes up the particular communicator. The following function call provides the value of the number of processes of the specified communicator:

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

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

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

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

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

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

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

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

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

Communication Collectives: Communication collective operations can dramatically expand interprocess communication from point-to-point to n-way or all-way data exchanges.

The scatter operation: The scatter collective communication pattern, like broadcast, shares data of one process (the root) with all the other processes of a communicator. But in this case it partitions a set of data of the root process into subsets and sends one subset to each of the processes. Each receiving process gets a different subset, and there are as many subsets as there are processes. In this example the send array is A and the receive array is B. B is initialized to 0. The root process (process 0 here) partitions the data into subsets of length 1 and sends each subset to a separate process.

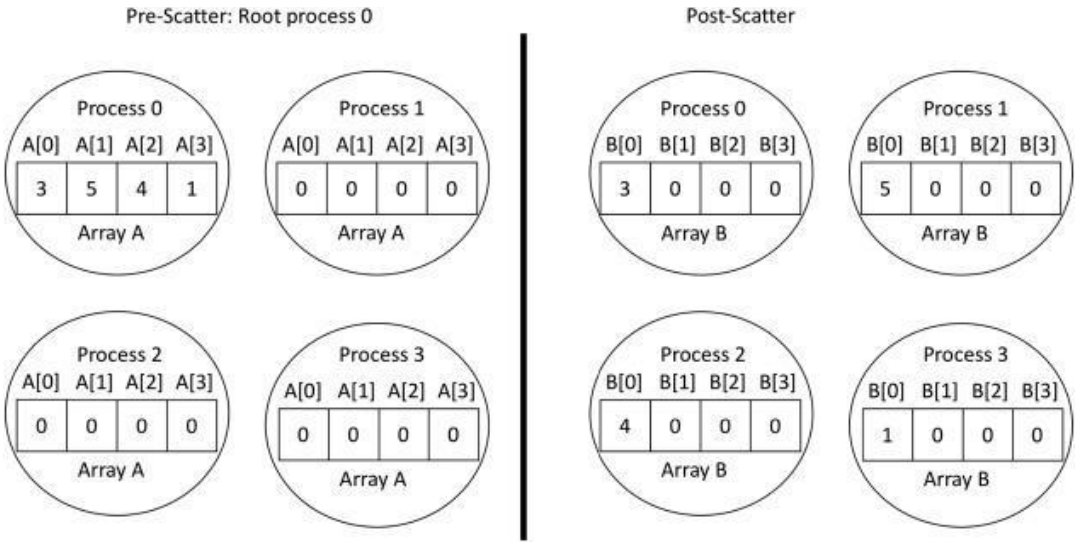


Fig: The Scatter Process

Designing the solution:

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

Implementing the solution:

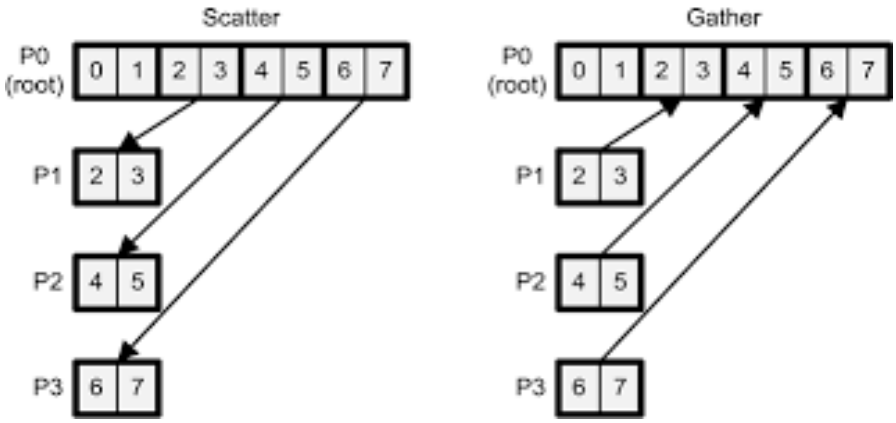


Fig: MPI on Scatter and Gather Operations

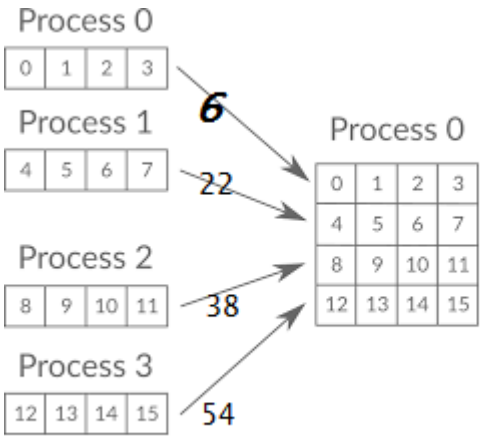


Fig. MPI on gathering results(sums) of distributed numbers

Installation of OPENMPI

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 the openmpi folder is created in ‘opt‘ folder of the 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
```

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;
}
```

Output:

```
$mpicc prgm.c
$mpirun -np 4 ./a.out
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
```

2nd Program to be Written

Problem Statement for Batch C:

To develop a distributed system, to find the average number of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate average element calculated at different processors.

Conclusion:

We have distributed N elements to n number of processors and collected their sum implementing parallel processing using MPI functions.

Submitted By: _____

Roll No.: _____

Class: B.E.I.T.

Staff Name: Ms. M. A. Rane

Staff Signature:

Date:

ASSIGNMENT NO. 5

Problem Statement:

To implement a token ring based mutual exclusion algorithm.

Tools / Environment:

Java Programming Environment, JDK 1.8 or C++ Programming Environment

Related Theory:

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 as shown in Fig. The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. Each process knows who is next in line after itself.

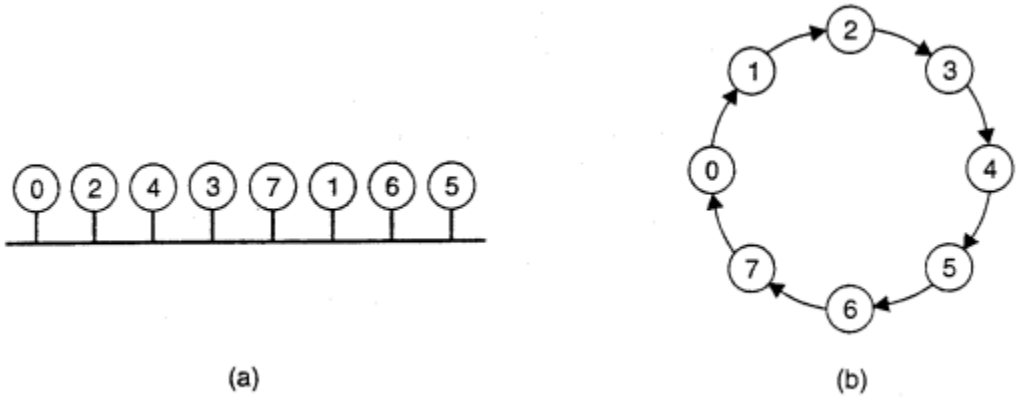


Fig. (a) An unordered group of processes on a network. (b) A logical ring constructed

When the ring is initialized, process 0 is given a token. The token circulates around the ring. It is passed from process k to process k +1 (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it needs to access the shared resource. If so, the process goes ahead, does all the work it needs to, and releases the resources. After it has finished, it passes the token along the ring. It is not permitted to immediately enter the resource again using the same token.

If a process is handed the token by its neighbor and is not interested in the resource, it just passes the token along. As a consequence, when no processes need the resource, the token just circulates at high speed around the ring. The correctness of this algorithm is easy to see. Only one process has the token at any instant, so only one process can actually get to the resource. Since the token circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to have access to the resource, at worst it will have to wait for every other process to use the resource.

If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it. The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintain the current ring configuration.

Advantages:

- The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a CS
- Since the token circulates among processes in a well-defined order, starvation cannot occur.

Disadvantages:

- Once a process decides it wants to enter a CS, at worst it will have to wait for every other process to enter and leave one critical region.
- If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is not constant. The fact that the token has not been spotted for an hour does not mean that it has been lost; some process may still be using it.
- The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can pass the token to the next member down the line

A brief comparison of the four mutual exclusion algorithms is given below.

Algorithm	Messages per Entry/Exit	Delay before Entry	Problems
Centralised	3	2	Coordinator Crash
Decentralised	3mk, k=1,2,....	2 m	Starvation, Low efficiency
Distributed	2 (n-1)	2 (n-1)	Crash of any process
Token Ring	1 to ∞	0 to n-1	Lost token, Process Crash

Table: A comparison of three mutual exclusion algorithms.

With the token ring algorithm, the number of messages is variable. If every process constantly wants to enter a critical region. then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

Finally, all algorithms except the decentralized one suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system.

Conclusion:

Token Ring algorithm achieves mutual exclusion in a distributed system by creating a bus network of processes. We successfully implemented Token-Ring Mutual Exclusion.

Submitted By: _____

Roll No.: _____

Class: B.E.I.T.

Staff Name: Ms. M. A. Rane

Staff Signature:

Date

ASSIGNMENT NO. 7

Problem Statement:

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

Tools / Environment:

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

Related Theory:

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

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

Types of Web Services:
There are two types of web services:

1. **SOAP**: SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it's XML based, it's platform and language independent. So, our server can be based on JAVA and the 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 popular recently because it has a 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.

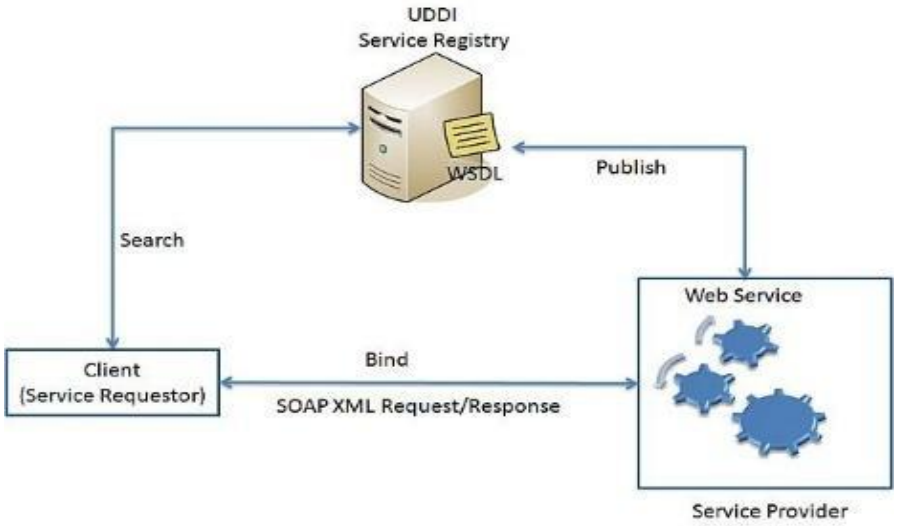
5. Then, the web service takes this response content as a SOAP message and reverts to the SOAP HTTP request invoked by the client program with this response.

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.

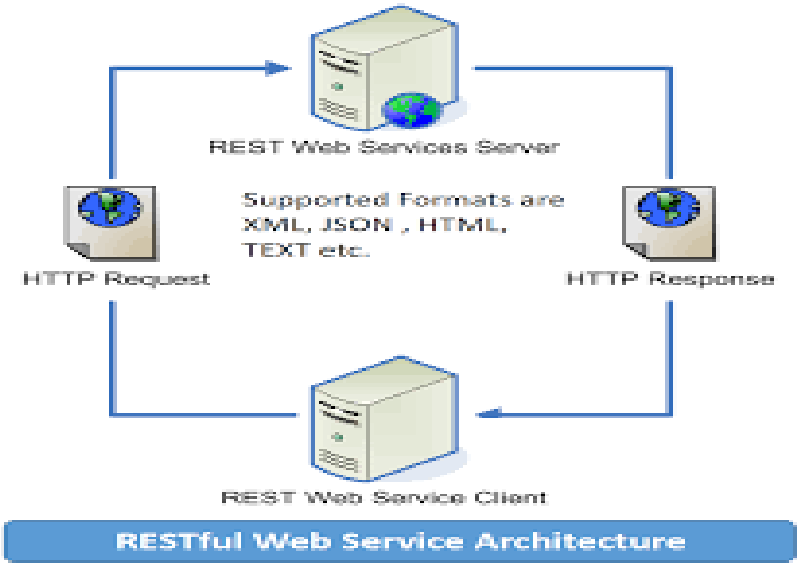
Web Services Description Language (**WSDL**)*:* WSDL is an XML document containing information about web services, such as the method name, method parameters, and how to invoke the service. WSDL is part of the UDDI registry. It acts as an interface between applications that want to interact based on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



SOAP Web Services Architecture

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
Bandwith usage	Uses more bandwith over the internet	Uses less bandwith
Client-server coupling	Tighter client-server coupling	Looser client server coupling
Security	Built in mechanism for security	No built in security
Data formats	Supports only XML	Supports multiple formats
Exposing business logic	Service interfaces	URIs
Failure handling	Retry logic built-in	Expects clients to retry
Caching data	Cannot be cached	Can be cached
Java API	JAX-WS	JAX-RS

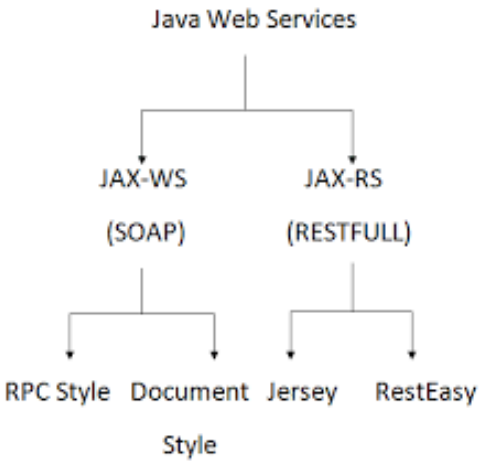
Designing the solution:

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

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

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

Both of these APIs are part of standard JDK installation, so no need to add any jars to work with them.



Implementing the solution:

Step 1: Choosing a container:

You can either deploy your web service in a web container.

- Choose **File** > **New Project** (Ctrl-Shift-N on Linux and Windows).
- Select **Web Application** from the Java **Web** category.
- Name the project *CalculatorWSApplication*.
- Select a location for the project. Click **Next**.
- Select the server **[Glassfish / Tomcat]** and **Java EE** version and click **Finish.**

Step 2: Creating a Web Service from a Java Class:

- Right-click the *CalculatorWSApplication* node and **choose New** > **Web Service**.
- Name the web service *CalculatorWS* and type *org.me.calculator* in Package.
- Keep "**Create Web Service from Scratch**" check box selected.
- If you are creating a Java EE project on GlassFish, select "**Implement Web Service as a Stateless Session Bean"**.
- Click **Finish**. The Projects window displays the structure of the new web service and the source code is shown in the editor area.

Step 3: Adding an Operation to the Web Service:

- Find the web service's node in the Projects window. Right-click that node. A context menu opens.
- Click **Add Operation** in either the visual designer or the context menu. The Add Operation dialog opens.
- In the upper part of the **Add Operation dialog box**, type **add** in **Name** and type **int** in the **Return Type** drop-down list.
- In the lower part of the Add Operation dialog box, click **Add** and create a parameter of type **int** named **i**. Click **Add** again and create a parameter of type **int** called **j**.
- Click **OK** at the bottom of the Add Operation dialog box. You return to the editor.
- Remove the **default hello operation**, either by deleting the hello() method in the **source code** or by selecting the hello operation in the **visual designer** and clicking **Remove Operation**.
- Click **Source** menu and view the generated code.
- In the editor, extend the skeleton **add operation.** Add the following:

```
int k = i + j;
return k; (instead of return 0)
```

Step 4: Deploying and Testing the Web Service:

Once you deploy a web service to a server, you can use the IDE to open the server's test client. The GlassFish server provides test clients whereas in Tomcat Web Server, there is no test client.

- Right-click the **project** and choose **Deploy**. The IDE starts the application server, builds the application, and deploys the application to the server.

- In the IDE's **Projects** tab, expand the **Web Services** node of the *CalculatorWSApplication* project. Right-click the *CalculatorWS* node, and choose **Test Web Service.**
- The IDE opens the tester page in the browser, if you deployed a web application to the GlassFish server.

Step 5: Consuming the Web Service:

Once the web service is deployed, you need to **create a client** to make use of the web service's **add** method. Here, you can create three types of clients: a Java class in a Java SE application, a servlet, and a JSP page in a web application.

Client 1: Java Class in Java SE Application

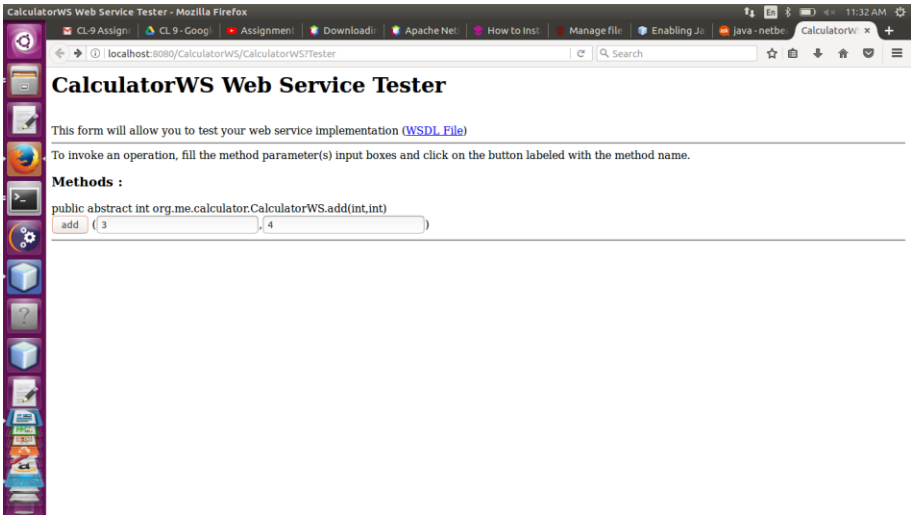
- Choose **File** > **New Project**.
- Select **Java Application** from the **Java** category.
- Name the project CalculatorWS_Client_Application.
- Keep "**Create Main Class selected**" and accept all other default settings. Click **Finish**.
- Right-click the CalculatorWS_Client_Application node and choose **New** > **Web Service Client**. The New Web Service Client wizard opens.
- Select "Project as the **…** source. Click on **Browse** button. Browse to the CalculatorWS web service in the **CalculatorWSApplication** project.
- When you have selected the web service, click **OK**.
- Do not select **a package** name. Leave this field empty. Keep the other settings at default and click **Finish**.
- The Projects window displays the new **web service client**, with a node for the **add** method that is created
- Double-click your **main class** so that it opens in the Source Editor. Drag the **add** node below the main() method.
- In the main() method body, **write the** code that initializes values for i and j, calls add(), and prints the result.

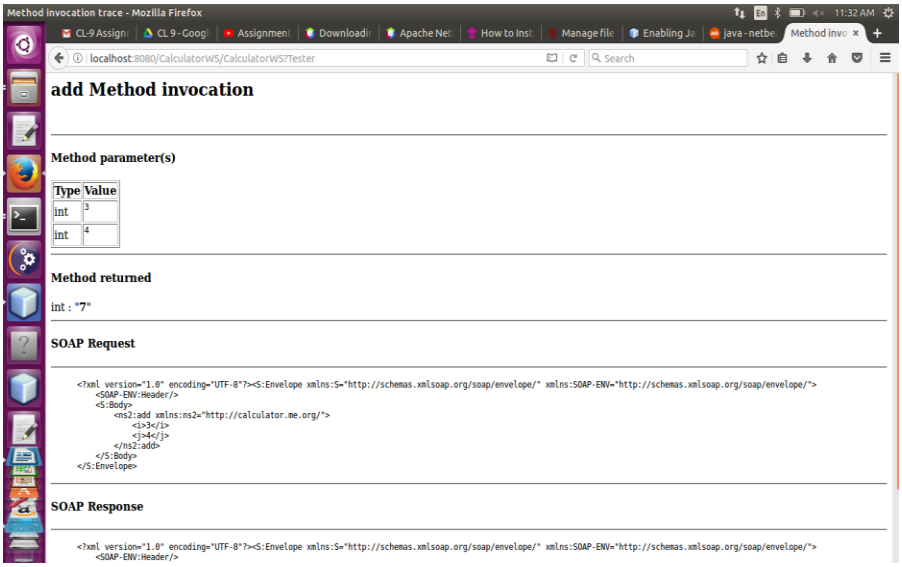
```
try
{
     int i=3;
     int j=4;
     int result = add(i,j);
     System.out.println("Result =" + result);
} catch (Exception ex) {
          System.out.println("Exception =" + ex);
```

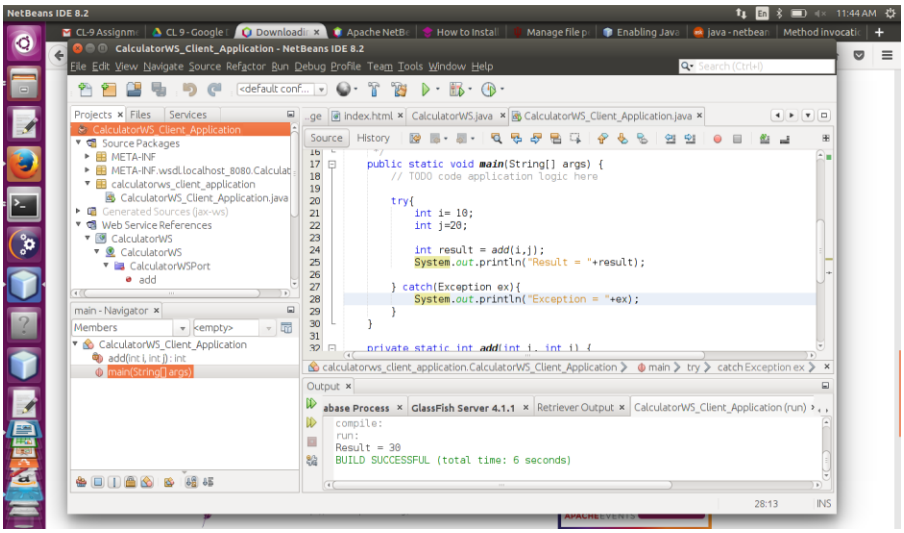
Compiling and Executing the solution:

Right Click on the Project node and Choose Run.

Output:


CalculatorWS Web Service Tester - Mozilla Firefox
localhost:8080/CalculatorWS/CalculatorWS?Tester
CalculatorWS Web Service Tester
This form will allow you to test your web service implementation (WSDL File)
To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.
Methods :
public abstract int org.me.calculator.CalculatorWS.add(int,int)
add (3 , 4)


Method invocation trace - Mozilla Firefox
localhost:8080/CalculatorWS/CalculatorWS?Tester
add Method invocation
Method parameter(s)
Type Value
int 3
int 4
Method returned
int : "7"
SOAP Request
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<S:Body>
<ns2:add xmlns:ns2="http://calculator.me.org/">
3
<j>4</j>
</ns2:add>
</S:Body>
</S:Envelope>
SOAP Response
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>



Conclusion:

This assignment described the Web services approach to the Service Oriented Architecture concept. Also, described the Java APIs for programming Web services and demonstrated examples of their use by providing detailed step-by-step examples of how to program Web services in Java.

Submitted By:

Roll No.:

Class: B.E.I.T.

Staff Name: Ms. M. A. Rane

Staff Signature:

Date: