JAYAWANT SHIKSHAN PRASARAK MANDAL
WAGHOLI CAMPUS, PUNE- 412207
BHIVARABAI SAWANT INSTITUTE OF TECHNOLOGY AND RESEARCH, WAGHOLI, PUNE

LAB MANUAL

LAB PRACTICE- V
Course Code: 414454

Class: B.E. I.T.

Course Pattern: 2019

Semester VIII A.Y. 2022-23

SUBJECT INCHARGE:- MRS. VIDYA JAGTAP

Course Objectives:

1. **The course aims to provide an understanding of the principles on which the distributed systems are based, their architecture, algorithms and how they meet the demands of Distributed applications.**

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

Course Outcomes:

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

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

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

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

LIST OF EXPERIMENT

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

ASSIGNMENT NO. 1

Problem Statement:

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

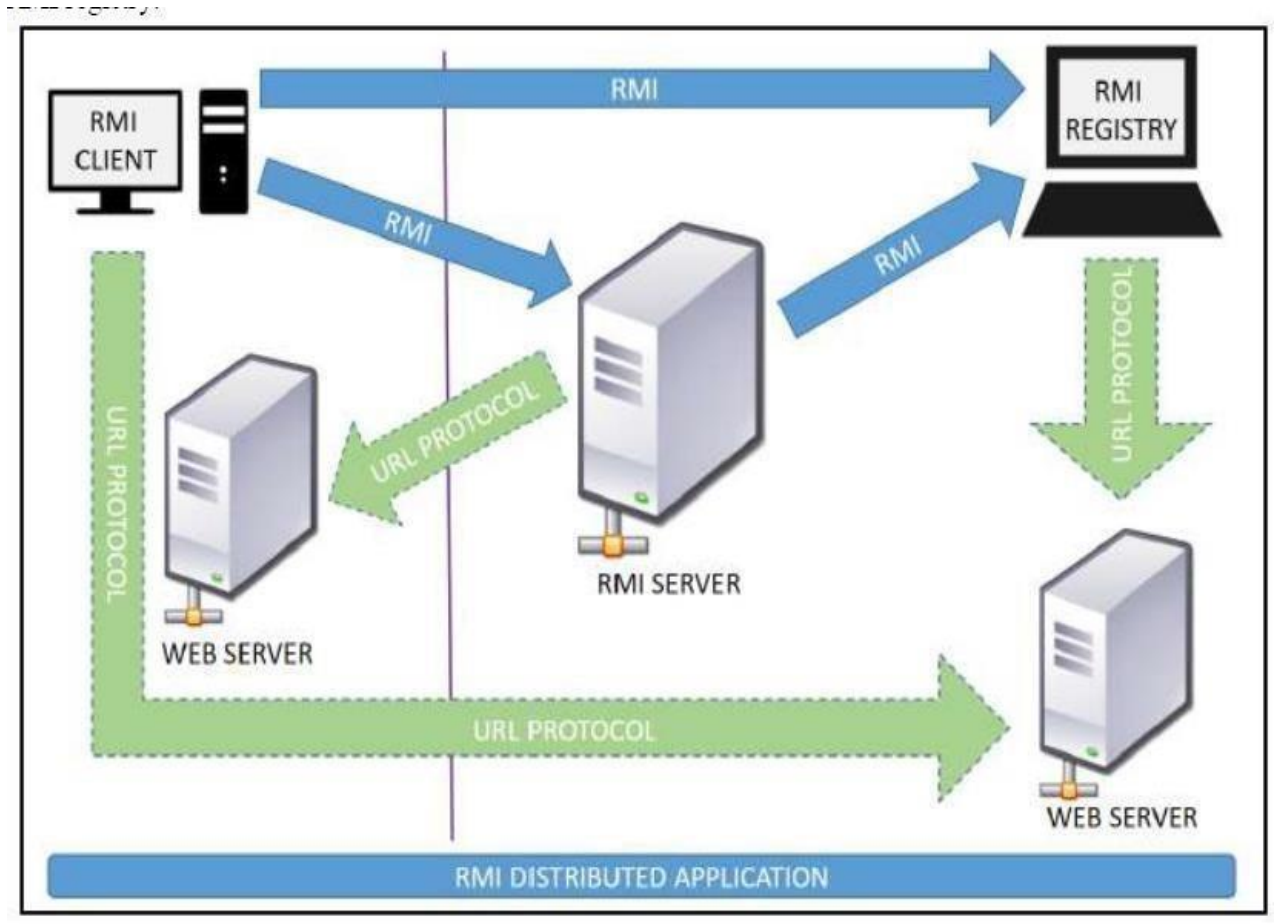
Tools / Environment:

Java Programming Environment, JDK 1.8, RMI-registry

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 ofan 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.

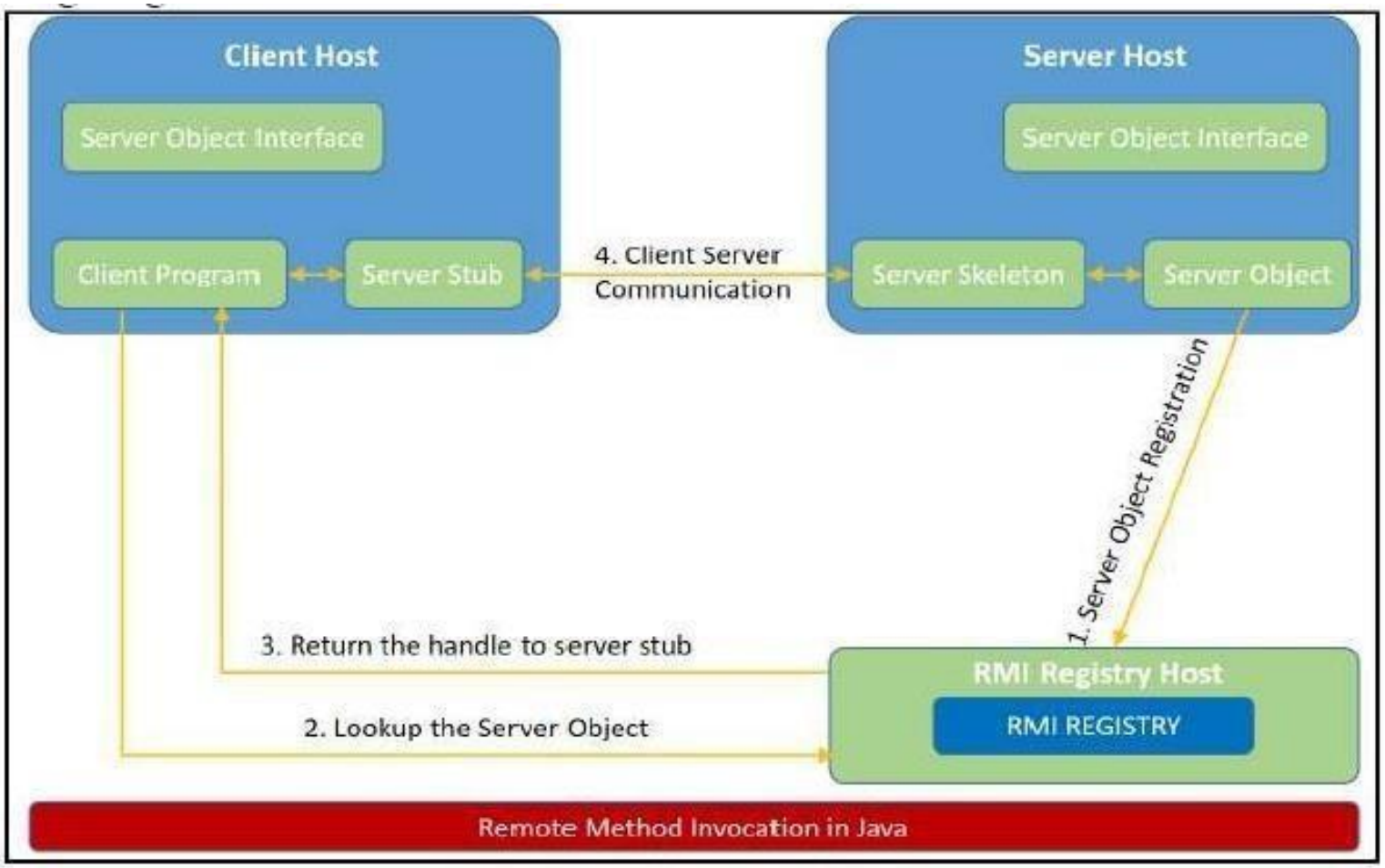
Remote interface: This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.

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 byall modern Java frameworks, including spring. In the same pattern, the definition of a remote interface takes importance in RMI design as well.

2. **Remote object implementation**: Java allows a class to implement more than one interface at a time. This helps remote objects implement one or more remote interfaces. The remote objectclass 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:
Consider building an application to perform diverse mathematical operations.
The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

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

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

The second source file, **AddServerImpl.java**, implements the remote interface. The implementation of the **add()** method is straightforward. All remote objects must extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.

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

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

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

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

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

To generate a stub the command is RMIcompiler is invoked as follows:
rmic AddServerImpl.
This command generates the file **AddServerImpl_Stub.class**.

3. Install Files on the Client and Server Machines
Copy **AddClient.class**, **AddServerImpl_Stub.class**, **AddServerIntf.class** to a directory on the client machine.
Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_Stub.class**, and **AddServer.class** to a directory on the server machine.

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

5. Start the Server
The server code is started from the command line: java AddServer

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

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

Source code:

// **Program for AddClient**

```
import  java.rmi.*;
public class AddClient {
      public static void main(String args[]) {
            try {
                  String addServerURL = "rmi://" + args[0] + "/AddServer";
                  AddServerIntf addServerIntf =
                     (AddServerIntf)Naming.lookup(addServerURL);
                  System.out.println("The first number is: " + args[1]);
                  double d1 = Double.valueOf(args[1]).doubleValue();
                  System.out.println("The second number is: " + args[2]);
                  double d2 = Double.valueOf(args[2]).doubleValue();
                  System.out.println("The sum is: " + addServerIntf.add(d1,
                     d2));
            }
            catch(Exception e) {
                  System.out.println("Exception: " + e);
            }
      }
}
```

//Program for AddServer

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

//Program for AddServerImpl

```
import  java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject implements
AddServerIntf {
      public AddServerImpl() throws RemoteException {
      }
      public double add(double d1, double d2) throws RemoteException {
            return d1 + d2;
      }
}
```

//Program for AddServerIntf

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

Compilation and Executing the solution:

1. Create all java files and compile using **javac** command , it will generate **.class** files.
2. Generate stubs invoking **rmic AddServerImpl** it will generate **AddServerImpl_Stub.class** file.
3. Copy **AddClient.class**, **AddServerImpl_Stub.class**, and **AddServerIntf.class** to a directory on the client machine/folder.
4. Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_ Stub.class**, and **AddServer.class** to a directory on the server machine/folder.
5. Start the RMI Registry on the Server Machine using **rmiregistry**
6. In new terminal start the Server using **java AddServer**
7. In another new terminal **s**tart the Client **java AddClient servername/ip_address 8 9** where servername is first argument and 8 , 9 are second & third arguments respectively.
   e.g **java AddClient 127.0.0.1 8 9** for localhost (when client and server on same machine)
   e.g **java AddClient 172.16.86.80 8 9** (when client and server on different machine, specify IP address of server machine)

OUTPUT:

dell@dell-Vostro-3546:~$ cd Desktop

dell@dell-Vostro-3546:~/Desktop$ cd ass1b

dell@dell-Vostro-3546:~/Desktop/ass1b$ javac *.java

dell@dell-Vostro-3546:~/Desktop/ass1b$ rmic AddServerImpl

Warning: generation and use of skeletons and static stubs for JRMP

is deprecated. Skeletons are unnecessary, and static stubs have been

uperseded by dynamically generated stubs. Users are

encouraged to migrate away from using rmic to generate skeletons and static

stubs. See the documentation for java.rmi.server.UnicastRemoteObject.

dell@dell-Vostro-3546:~/Desktop/ass1b$ rmiregistry

*****SERVER SIDE:**

dell@dell-Vostro-3546:~$ cd Desktop

dell@dell-Vostro-3546:~/Desktop$ cd ass1b

dell@dell-Vostro-3546:~/Desktop/ass1b$ cd Server

dell@dell-Vostro-3546:~/Desktop/ass1b/Server$ javac *.java

dell@dell-Vostro-3546:~/Desktop/ass1b/Server$ java AddServer

in server side

*****CLIENT SIDE:**

dell@dell-Vostro-3546:~$ cd

Desktop dell@dell-Vostro-

3546:~/Desktop$ cd ass1b

dell@dell-Vostro-3546:~/Desktop/ass1b$ cd

Client dell@dell-Vostro-

3546:~/Desktop/ass1b/Client$ javac *.java

dell@dell-Vostro-3546:~/Desktop/ass1b/Client$ java AddClient

localhost 4 5The first number is: 4

The second number is: 5

The sum is: 9.0

dell@dell-Vostro-3546:~/Desktop/ass1b/Client$

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 aJava object that executes on one machine to invoke a method of a Java object that executes on anothermachine. This is an important feature, because it allows us to build distributed applications.

ASSIGNMENT NO. 2

Problem Statement:

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

Tools / Environment:

Java Programming Environment, JDK 1.8

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.

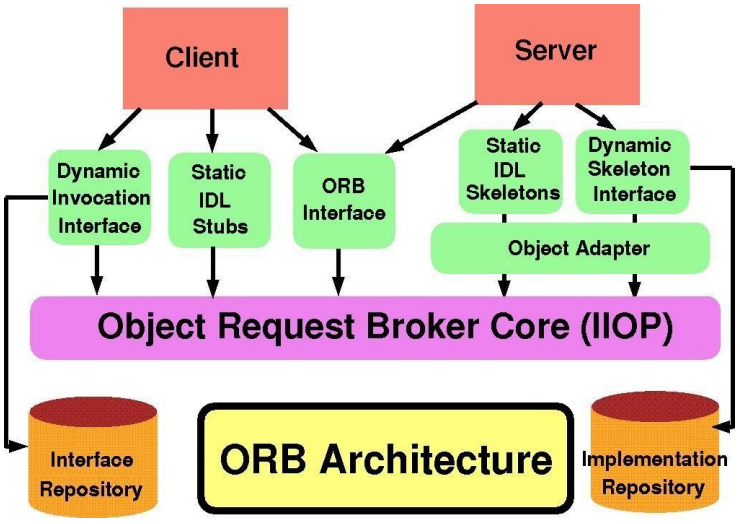
An application developed based on CORBA standards with standard **Internet Inter-ORB Protocol** (**IIOP**), irrespective of the vendor that develops it, should be able to smoothly integrate and operate with another application developed based on CORBA standards through the same or different vendor.
Except 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 IDL script. This is close to writing a program to an interface, a concept we have been discussing that most recent programming languages and frameworks, such as Spring. The interface has to be defined clearly for each object. The systems encapsulate the actual implementation along with their respective data handling and processing, and only the methods are available to the rest of the world through the interface. Hence, the clients are forced to develop their invocation logic for the IDL interface exposed by the application they want to connect to with the method parameters (input and output) advised by the interface operation.

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

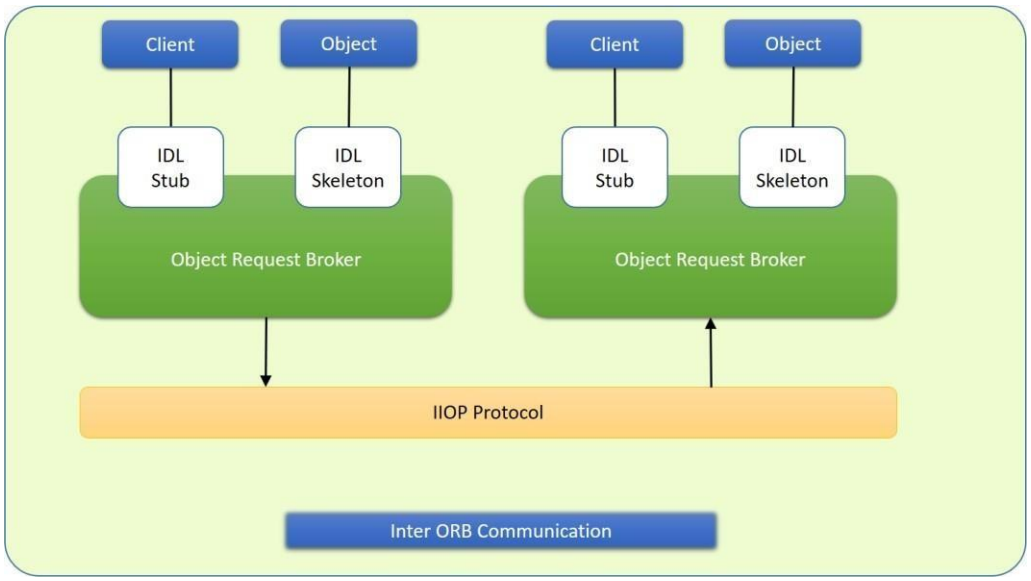


In CORBA, each object instance acquires an object reference for itself with the electronic token identifier. Client invocations are going to use these object references that have the ability to figure out which ORB instance they are supposed to interact with. The stub and skeleton represent the client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

establish this communication through ORB and pass the arguments to the right method and its instance during the invocation.

Inter-ORB communication

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



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

Java Support for CORBA

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

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

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

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

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

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

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

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

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

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

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

Designing the solution:

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

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

1. Creating CORBA Objects using Java IDL:

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

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

Interfaces in IDL are declared much like interfaces in Java.

Modules

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

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

Interfaces

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

```
Interface PrintServer: Server {...
```

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

Data members and methods

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

```
readonly attribute string myString;
```

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

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

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

A complete IDL example

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 serverside implementation of the interface. The base class can accept requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named `Server` is called `_ServerImplBase`.

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

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

3. Writing the Implementation

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

Implementing the solution:

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

1. Defining the Interface (Hello.idl)

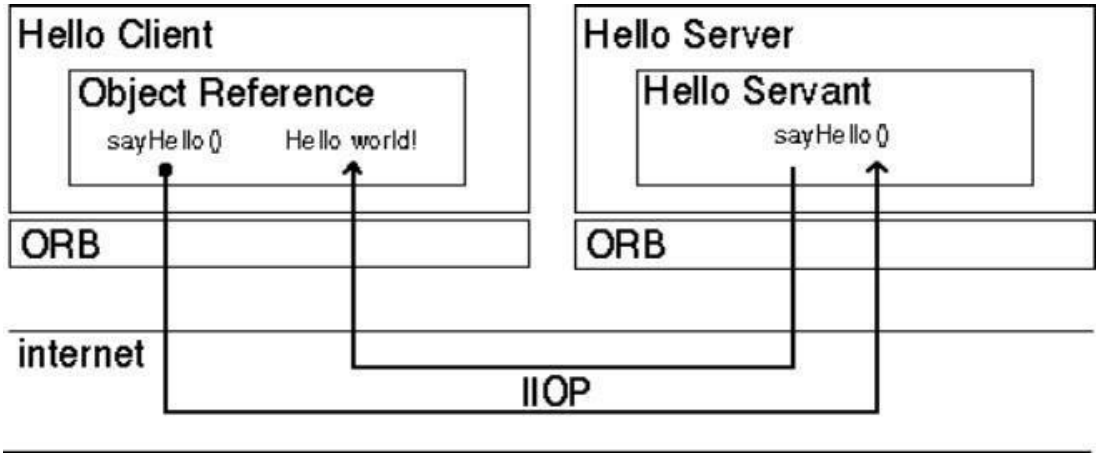
The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL).To complete the application, you simply provide the server **(HelloServer.java)** and client **(HelloClient.java)** implementations.

2. Implementing the Server (HelloServer.java)

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

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

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



3. Implementing the Client Application (HelloClient.java)

The example application client that follows:

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

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. Thisstep assumes that you have included the path to the java/bin directory in your path.

idlj -fall ReverseModule.idl

The idlj compiler generates a number of files.

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

javac *.java ReverseModule/*.java

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

orbd -ORBInitialPort 1050&

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

java ReverseServer -ORBInitialPort 1050& -ORBInitialHost localhost&

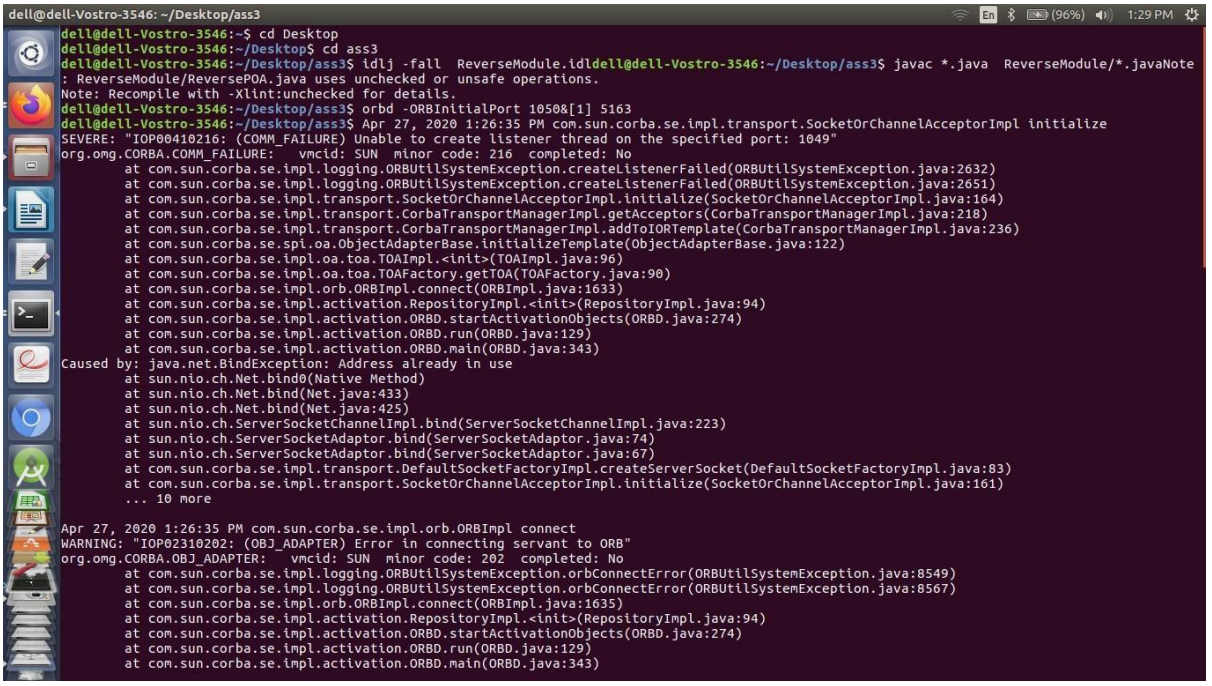
6. Run the client application :

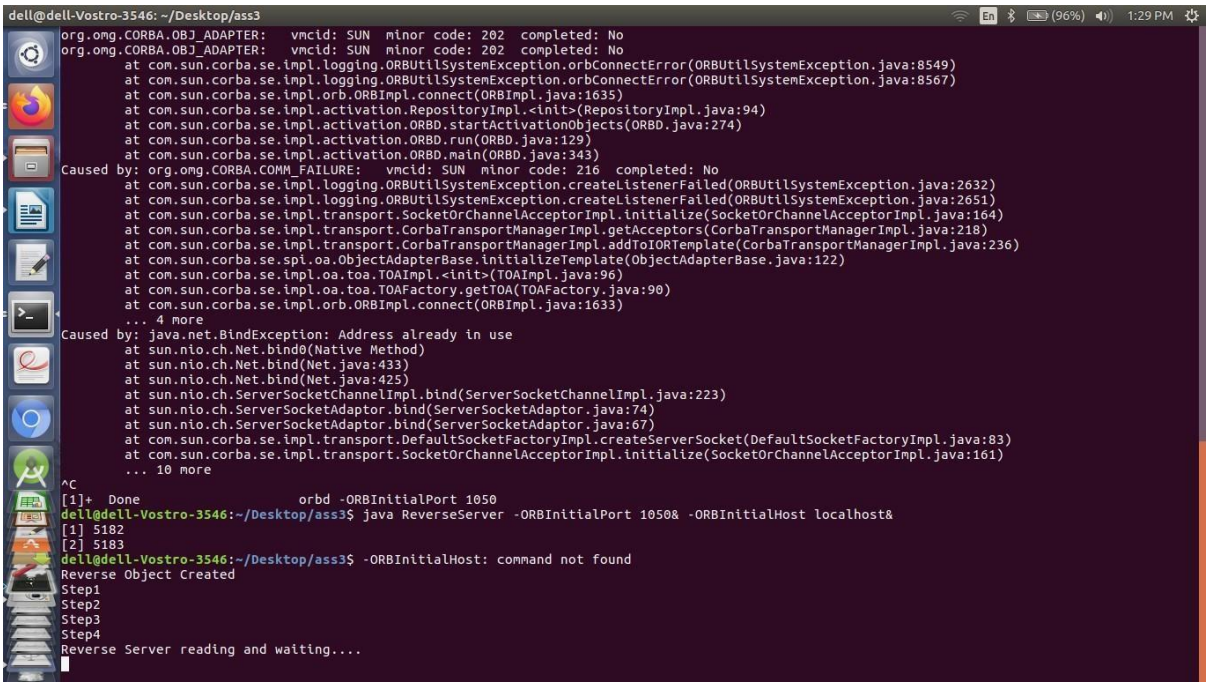
java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost

Output:

```
***SERVER SIDE:
dell@dell-Vostro-3546:~$ cd Desktop
dell@dell-Vostro-3546:~/Desktop$ cd ass3
dell@dell-Vostro-3546:~/Desktop/ass3$        idlj        -fall
ReverseModule.idldell@dell-Vostro-3546:~/Desktop/ass3$   javac
*.java ReverseModule/*.javaNote: ReverseModule/ReversePOA.java
uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
dell@dell-Vostro-3546:~/Desktop/ass3$   orbd   -ORBInitialPort
1050&[1] 5163
dell@dell-Vostro-3546:~/Desktop/ass3$  java  ReverseServer  -
ORBInitialPort 1050& -ORBInitialHost localhost&
[1] 4933
[2] 4934
dell@dell-Vostro-3546:~/Desktop/ass3$ -ORBInitialHost: command
not found
Reverse Object Created
Step1
Step2
```

Step3
Step4
Reverse Server reading and waiting....





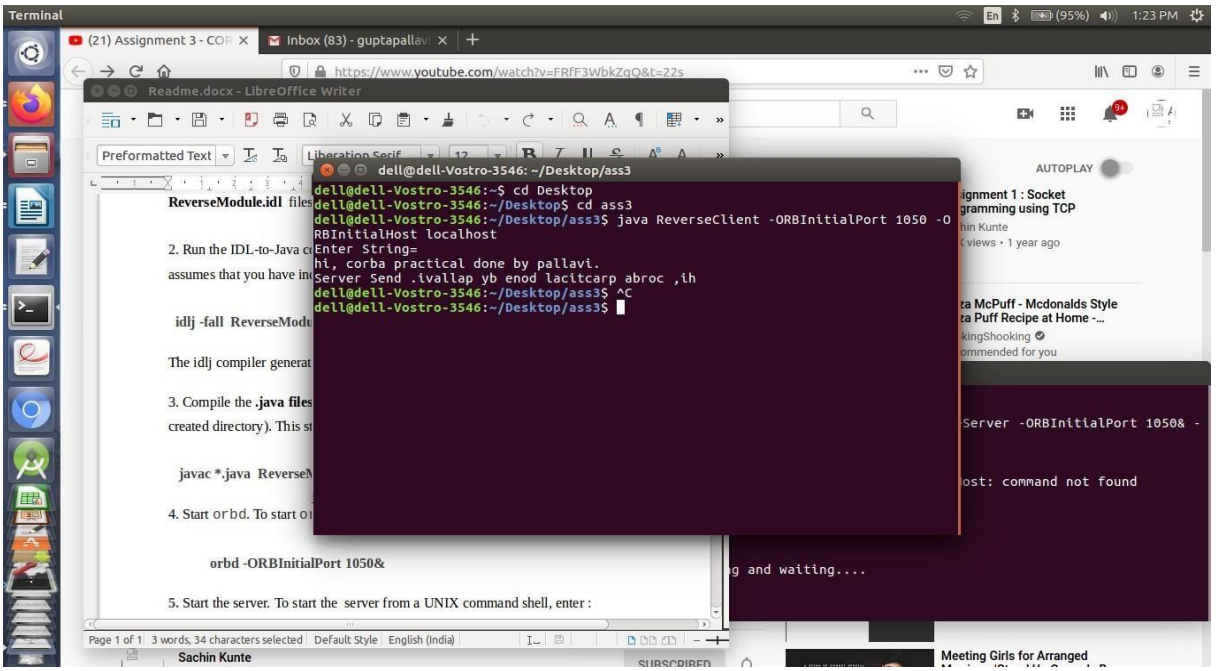
*****CLIENT SIDE:**

dell@dell-Vostro-3546:~$ cd Desktop

dell@dell-Vostro-3546:~/Desktop$ cd ass3

dell@dell-Vostro-3546:~/Desktop/ass3$ java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost

```
Enter String=
hi, corba practical done by pallavi.
Server Send .ivallap yb enod lacitcarp abroc ,ih
dell@dell-Vostro-3546:~/Desktop/ass3$
```



Conclusion:

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

ASSIGNMENT NO. 3

Problem Statement:

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

Tools / Environment:

Java Programming Environment, JDK1.8 or higher, MPI Library (mpi.jar), MPJ Express Software (Version 0.44).

Theory:

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

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

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

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

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

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

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

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

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

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

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

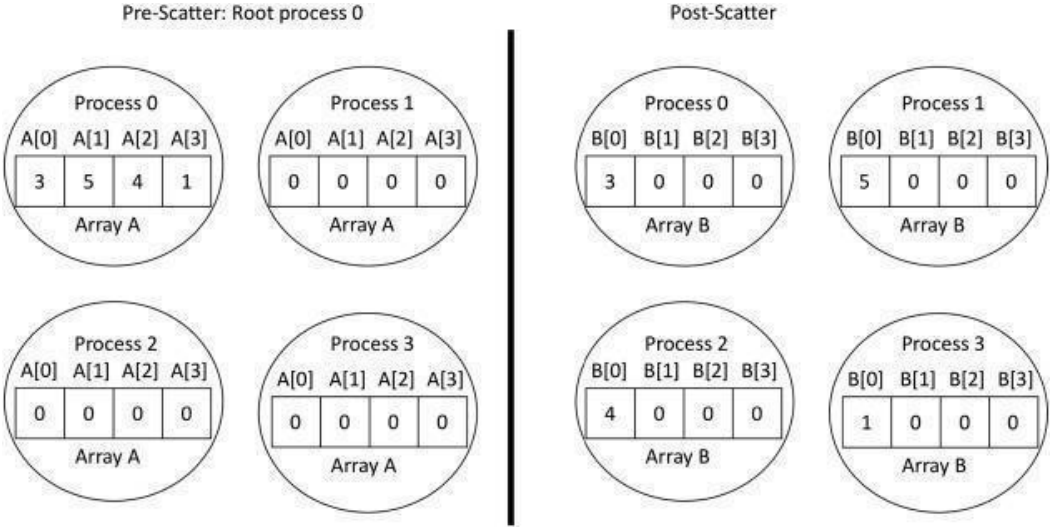


Fig: The Scatter Process

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

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

Multiple Data (SPMD).

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

Designing the solution:

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

Implementing the solution:

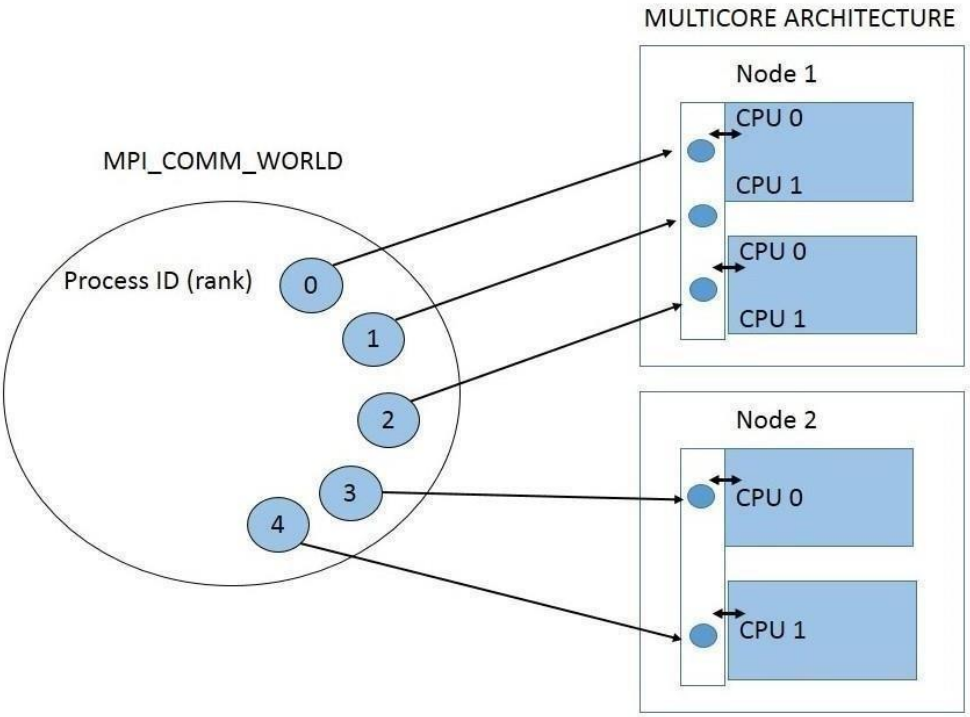


Fig: MPI on Multicore Architecture

Code:

```
import mpi.MPI;

public class ScatterGather {
      public static void main(String args[]){
            //Initialize MPI execution environment
            MPI.Init(args);
            //Get the id of the process
            int rank = MPI.COMM_WORLD.Rank();
            //total number of processes is stored in size
            int size = MPI.COMM_WORLD.Size();
            int root=0;
            //array which will be filled with data by root process
            int sendbuf[]=null;

            sendbuf= new int[size];

            //creates data to be scattered
            if(rank==root){
                  sendbuf[0] = 10;
                  sendbuf[1] = 20;
                  sendbuf[2] = 30;
                  sendbuf[3] = 40;

                  //print current process number
                  System.out.print("Processor "+rank+" has data: ");
                  for(int i = 0; i < size; i++){
                        System.out.print(sendbuf[i]+" ");
                  }
                  System.out.println();
            }
            //collect data in recvbuf
            int recvbuf[] = new  int[1];

            //following are the args of Scatter method
            //send, offset, chunk_count, chunk_data_type, recv,  offset,
            //chunk_count, chunk_data_type, root_process_id
            MPI.COMM_WORLD.Scatter(sendbuf, 0, 1, MPI.INT, recvbuf,  0,
1, MPI.INT, root);
            System.out.println("Processor "+rank+" has data:
"+recvbuf[0]);
            System.out.println("Processor "+rank+" is doubling the
data");
            recvbuf[0]=recvbuf[0]*2;
            //following are the args of Gather method
            //Object sendbuf, int sendoffset, int sendcount, Datatype
            //sendtype, Object recvbuf, int recvoffset, int  recvcount,
            //Datatype recvtype,
            //int root)
            MPI.COMM_WORLD.Gather(recvbuf, 0, 1, MPI.INT, sendbuf, 0,
1, MPI.INT, root);
            //display the gathered result
            if(rank==root){
                  System.out.println("Process 0 has data: ");
                  for(int i=0;i<4;i++){
```

```
                        System.out.print(sendbuf[i]+ " ");
                    }
                }
                //Terminate MPI execution
                environment MPI.Finalize();
        }
}
```

For implementing the MPI program in multi-core environment, we need to **install MPJ express**
library.

Download MPJ Express (mpj.jar, Version 0.44) and unpack it.

Compiling and Executing:

1. **Set MPJ_HOME and PATH environment variables**: export
   MPJ_HOME=/path/to/mpj/
   export PATH=$MPJ_HOME/bin:$PATH
   (These above two lines can be added to ~/.bashrc)
2. **Compile ScatterGather.java:**
   javac -cp $MPJ_HOME/lib/mpj.jar
   ScatterGather.java (mpj.jar is inside lib folder in the
   downloaded MPJ Express)
3. **Execute:**
   $MPJ_HOME/bin/mpjrun.sh -np **4** ScatterGather
   Note: the number 4 above indicates the no. of processes.

Output:

```
bvcoew@bvcoew-Lenovo-Product:~$ export
MPJ_HOME=/home/bvcoew/Desktop/4346/2/mpj-v0_44
bvcoew@bvcoew-Lenovo-Product:~$ cd Desktop/4345/2
bvcoew@bvcoew-Lenovo-Product:~/Desktop/4346/2$ javac -cp
$MPJ_HOME/lib/mpj.jar ScatterGather.java
bvcoew@bvcoew-Lenovo-Product:~/Desktop/4346/2$   $MPJ_HOME/bin/mpjrun.sh  -np
4 ScatterGather
bash: /home/bvcoew/Desktop/4345/2/mpj-v0_44/bin/mpjrun.sh: Permission
denied
bvcoew@bvcoew-Lenovo-Product:~/Desktop/4346/2$ chmod 777 mpj-
v0_44/bin/mpjrun.sh
bvcoew@bvcoew-Lenovo-Product:~/Desktop/4346/2$   $MPJ_HOME/bin/mpjrun.sh  -np
4 ScatterGather
MPJ Express (0.44) is started in the multicore configuration
Processor 0 has data: 10 20 30 40
Processor 0 has data: 10
Processor 2 has data: 30
```

```
Processor 1 has data: 20
Processor 3 has data: 40
Processor 2 is doubling the data
Processor 1 is doubling the data
Processor 3 is doubling the data
Processor 0 is doubling the data
Process 0 has data:
20 40 60 80
```

*****SYSTEM MONITOR*****

Conclusion:

There has been a large amount of interest in parallel programming using Java. mpj is an MPI binding with Java along with the support for multi core architecture so that user can develop the code on his/her own laptop or desktop. This is an effort to develop and run parallel programs according to MPI standard.

ASSIGNMENT NO.6

Problem Statement:

Implement Bully and Ring algorithm for leader election.

Tools / Environment:

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

Related Theory:

Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role. In general, it does not matter which process takes on this special responsibility, but one of them has to do it. If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process P has a unique identifier id(P). In general, election algorithms attempt to locate the process with the highest identifier and designate it as coordinator.

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

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 takeoverthe coordinators job.

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

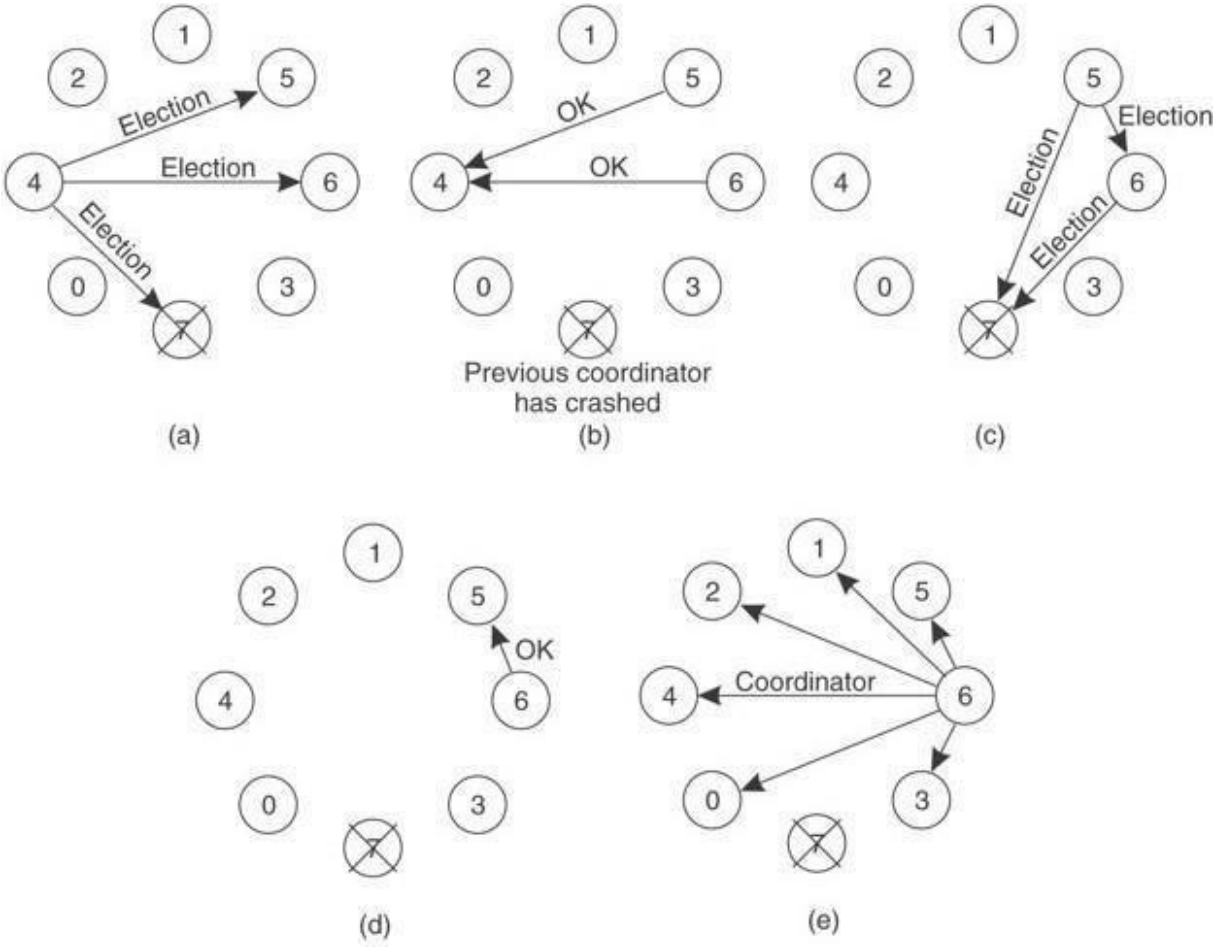


Figure 1: Bully Algorithm

Ring Algorithm:

Initiation:

1. 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 goesto the next member along the ring, or the one after that, until a running process is located).
4. At each step, sender adds its own process number to the list in the message.
5. When the message gets back to the process that started it all i.e Message comes back to initiator, the **process with maximum ID Number** in the queue **wins the Election**.
6. Initiator announces the winner by sending another message (Coordinator message) around the ring.

Implementing the solution:

For Ring Algorithm:

1. Creating Class for Process which includes
    i) State: Active / Inactive

ii) Index: Stores index of process.

iii) ID: Process ID

2. Import Scanner Class for getting input from Console
3. Getting input from User for number of Processes and store them into object of classes.
4. Sort these objects on the basis of process id.
5. Make the last process id as "inactive".
6. Ask for menu

1.Election

2.Quit

7. Ask for initializing election process.
8.These inputs will be used by Ring Algorithm.

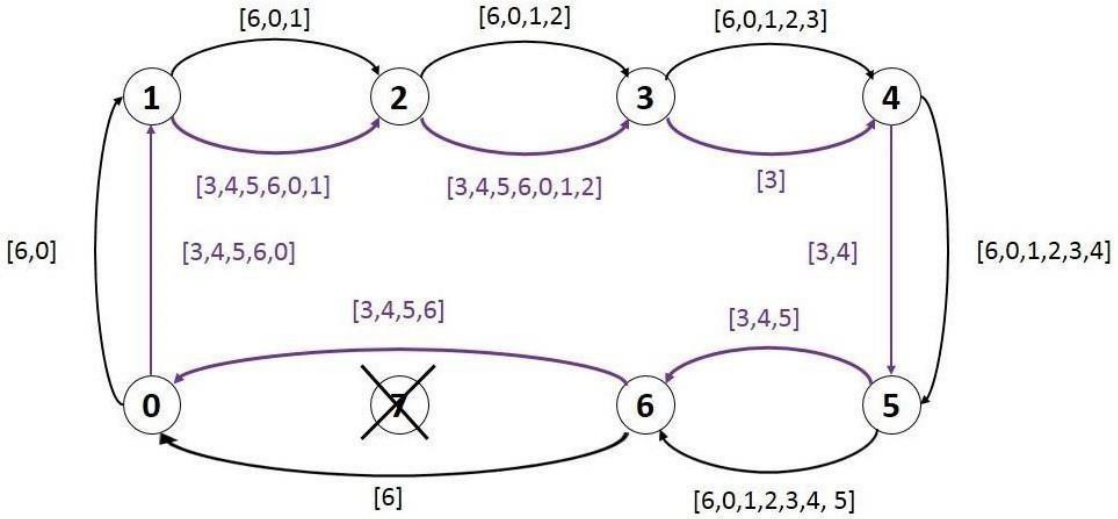


Figure 2: Ring Algorithm

Source code:

Ring.java

```
import java.util.Scanner;

public class Ring {

	public static void main(String[] args) {

		// TODO Auto-generated method stub

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

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

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

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


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


		for (i = 0; i < num; i++) {
			System.out.print("  [" + i + "]" + " " + proc[i].id);
		}

		int init;
		int ch;
		int temp1;
		int temp2;
		int ch1;
		int arr[] = new int[10];

		proc[num - 1].state = "inactive";
```

```
            System.out.println("\n process " + proc[num - 1].id + "select
as co-ordinator");

            while (true) {
                  System.out.println("\n 1.election 2.quit ");
                  ch = in.nextInt();

                  for (i = 0; i < num; i++) {
                        proc[i].f = 0;
                  }

                  switch (ch) {
                  case 1:
                        System.out.println("\n Enter the Process number who
initialsied election : ");
                        init = in.nextInt();
                        init--;
                        temp2 = init;
                        temp1 = init + 1;

                        i = 0;

                        while (temp2 != temp1) {
                              if ("active".equals(proc[temp1].state) &&
proc[temp1].f == 0) {

                                    System.out.println("\nProcess " +
proc[init].id + " send message to " + proc[temp1].id);
                                    proc[temp1].f = 1;
                                    init = temp1;
                                    arr[i] = proc[temp1].id;
                                    i++;
                              }
                              if (temp1 == num) {
                                    temp1 = 0;
                              } else {
                                    temp1++;
                              }
                        }

                        System.out.println("\nProcess " + proc[init].id + "
send message to " + proc[temp1].id);
                        arr[i] = proc[temp1].id;
                        i++;
                        int max = -1;


// finding maximum for co-ordinator selection
                        for (j = 0; j < i; j++) {
                              if (max < arr[j]) {
                                    max = arr[j];
                              }
                        }

// co-ordinator is found then printing on console
                        System.out.println("\n process " + max + "select as
co-ordinator");


                        for (i = 0; i < num; i++) {
```

```
                              if (proc[i].id == max) {
                                    proc[i].state = "inactive";
                              }
                        }
                        break;
                  case 2:
            System.out.println("Program terminated ...");
            return ;
                  default:
                        System.out.println("\n invalid response \n");
                        break;
                  }


            }
      }

}

class Rr {

      public int index;    // to store the index of process
      public int id;       // to store id/name of process
      public int f;
      String state;        // indiactes whether active or inactive state of
node

}
```

Compiling and Executing the solution:

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

Output:

```
Enter the number of process :
4
Enter the id of process :
1
Enter the id of process :
2
Enter the id of process :
3
Enter the id of process :
4
  [0] 1  [1] 2  [2] 3  [3] 4
 process 4select as co-ordinator

 1.election 2.quit
1

 Enter the Process number who initialsied election :
2

Process 2 send message to 3

Process 3 send message to 1

Process 1 send message to 2

 process 3select as co-ordinator
```

```
 1.election 2.quit
2
Program terminated ...
```

Bully.java

```
import java.io.InputStream;
import java.io.PrintStream;
import java.util.Scanner;

public class Bully {
    static boolean[] state = new boolean[5];
    int coordinator;

    public static void up(int up) {
        if (state[up - 1]) {
            System.out.println("process" + up + "is already up");
        } else {
            int i;
            Bully.state[up - 1] = true;
            System.out.println("process " + up + "held election");
            for (i = up; i < 5; ++i) {
                System.out.println("election message sent from process" +
up + "to process" + (i + 1));
            }
            for (i = up + 1; i <= 5; ++i) {
                if (!state[i - 1]) continue;
                System.out.println("alive message send from process" + i
+ "to process" + up);
                break;
            }
        }
    }

    public static void down(int down) {
        if (!state[down - 1]) {
            System.out.println("process " + down + "is already dowm.");
        } else {
            Bully.state[down - 1] = false;
        }
    }

    public static void mess(int mess) {
        if (state[mess - 1]) {
            if (state[4]) {
                System.out.println("0K");
            } else if (!state[4]) {
                int i;
                System.out.println("process" + mess + "election");
                for (i = mess; i < 5; ++i) {
                    System.out.println("election send from process" +
mess + "to process " + (i + 1));
```

```
                }
                for (i = 5; i >= mess; --i) {
                    if (!state[i - 1]) continue;
                    System.out.println("Coordinator message send from
process" + i + "to all");
                    break;
                }
            }
        } else {
            System.out.println("Prccess" + mess + "is down");
        }
    }

    public static void main(String[] args) {
        int choice;
        Scanner sc = new Scanner(System.in);
        for (int i = 0; i < 5; ++i) {
            Bully.state[i] = true;
        }
        System.out.println("5 active process are:");
        System.out.println("Process up = p1 p2 p3 p4 p5");
        System.out.println("Process 5 is coordinator");
        do {
            System.out.println("........ ");
            System.out.println("1 up a process.");
            System.out.println("2.down a process");
            System.out.println("3 send a message");
            System.out.println("4.Exit");
            choice = sc.nextInt();
            switch (choice) {
                case 1: {
                    System.out.println("bring proces up");
                    int up = sc.nextInt();
                    if (up == 5) {
                        System.out.println("process 5 is co-ordinator");
                        Bully.state[4] = true;
                        break;
                    }
                    Bully.up(up);
                    break;
                }
                case 2: {
                    System.out.println("bring down any process.");
                    int down = sc.nextInt();
                    Bully.down(down);
                    break;
                }
                case 3: {
                    System.out.println("which process will send
message");
                    int mess = sc.nextInt();
                    Bully.mess(mess);
                }
            }
        } while (choice != 4);
    }
}
```

Output:

```
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
2
bring down any process.
5
.........
1 up a process.
2.down a process
3 send a message
4.Exit
3
which process will send message
2
process2election
election send from process2 to process 3
election send from process2 to process 4
election send from process2 to process 5
Coordinator message send from process4to all
.........
1 up a process.
2.down a process
3 send a message
4.Exit
4
```

Conclusion:

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