ASSIGNMENT NO.1

PROBLEM STATEMENT:

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

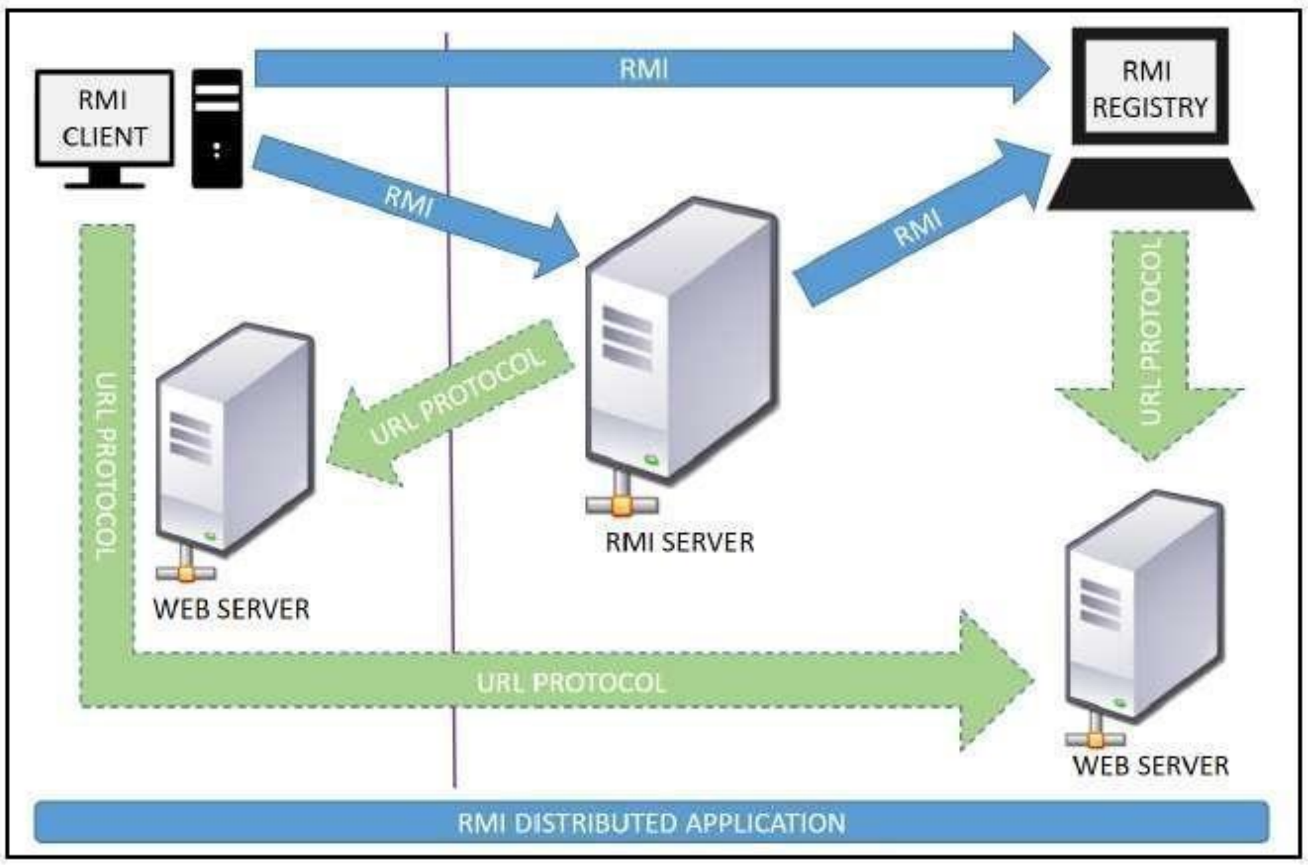
Tools / Environment:

Java Programming Environment, jdk 1.8, rmiregistry

Theory:

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

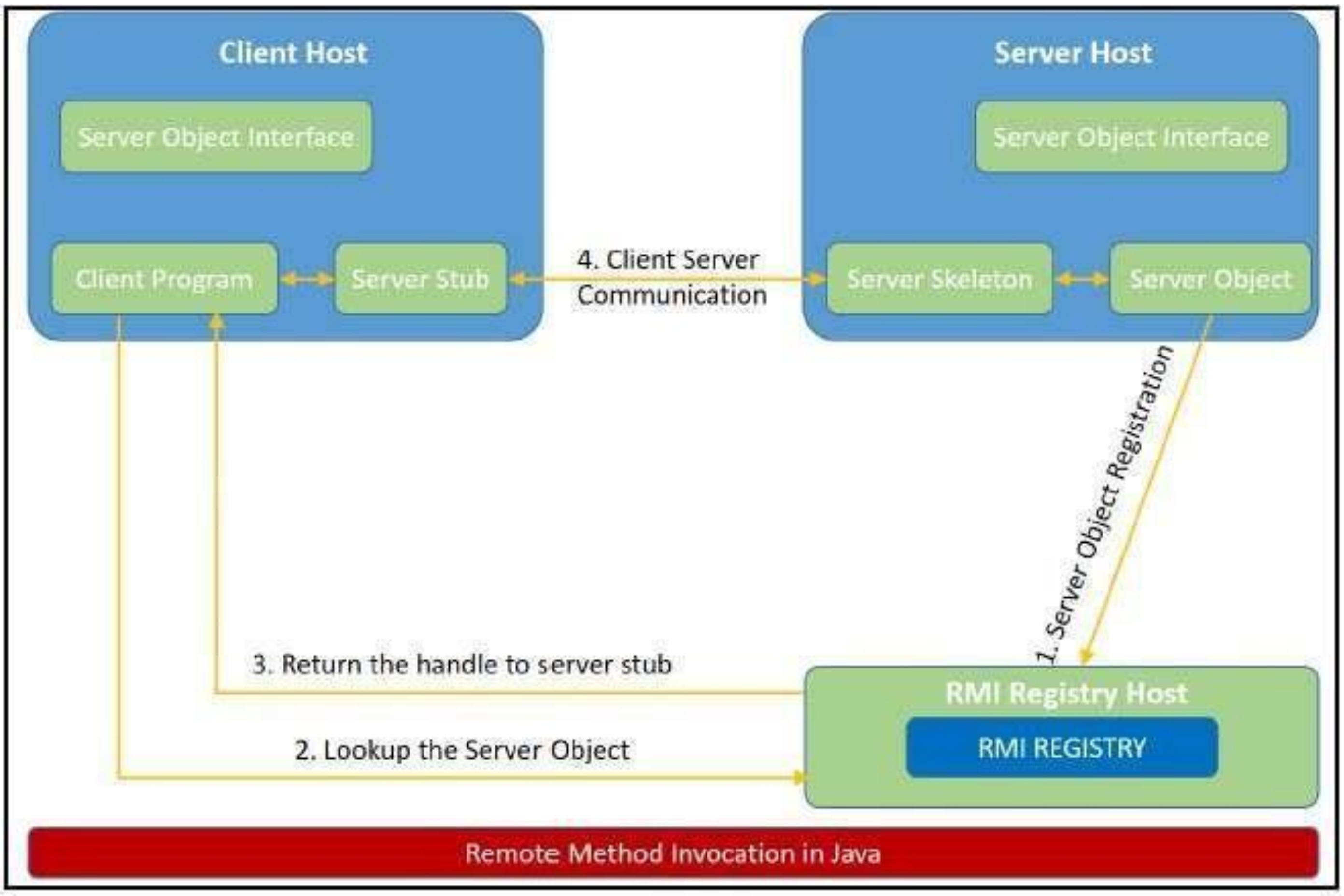
- **Key terminologies of RMI:**

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

1. **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.
2. **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 behaviours.

3. **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.
4. **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:
   i. It initiates a connection to the remote machine JVM.
   ii. It marshals (write and transmit) the parameters passed to it via the remote JVM.
   iii. 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.
5. **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:
   i. It reads the parameter sent to the remote method.
   ii. It invokes the actual remote object method.
   iii. It marshals (writes and transmits) the result back to the caller (stub).

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


Client Host
Server Object Interface
Client Program
Server Stub
4. Client Server Communication
Server Host
Server Object Interface
Server Skeleton
Server Object
1. Server Object Registration
3. Return the handle to server stub
2. Lookup the Server Object
RMI Registry Host
RMI REGISTRY
Remote Method Invocation in Java

Designing the solution:

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

1. Design and implement a component that should not only be involved in the distributed application, but also the local components.

2. Ensure that the components that participate in the RMI calls are accessible across networks.
3. Establish a network connection between applications that need to interact using the RMI.

- **Remote interface definition:** The purpose of defining a remote interface is to declare the methods that should be available for invocation by a remote client.

Programming the interface instead of programming the component implementation is an essential design principle adopted by all modern Java frameworks, including Spring. In the same pattern, the definition of a remote interface takes importance in RMI design as well.

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

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

Implementation:

Consider building an application to perform diverse mathematical operations.

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

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

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

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

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

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

The application begins by forming a string that follows the URL syntax. This URL uses the **rmi** protocol. The string includes the IP address or name of the server and the string “AddServer”. The program then invokes the **lookup( )** method of the **Naming** class. This method accepts one argument, the **rmi** URL, and returns a reference to an object of type **AddServerIntf**. All remote method invocations can then be directed to this object.

The program continues by displaying its arguments and then invokes the remote **add()**method. The sum is returned from this method and is then printed.

Use **javac** to compile the four source files that are created.

2. Generate a stub

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

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

To generate a stub the command is RMIcompiler is invoked as follows:

rmic AddServerImpl.

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

3. Install Files on the Client and Server Machines

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

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

4. Start the RMI Registry on the Server Machine

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

start rmiregistry

5. Start the Server

The server code is started from the command line, as shown here:

java AddServer

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

6. Start the Client

The **AddClient** software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

java AddClient 192.168.13.14 7 8

SOURCE CODE:

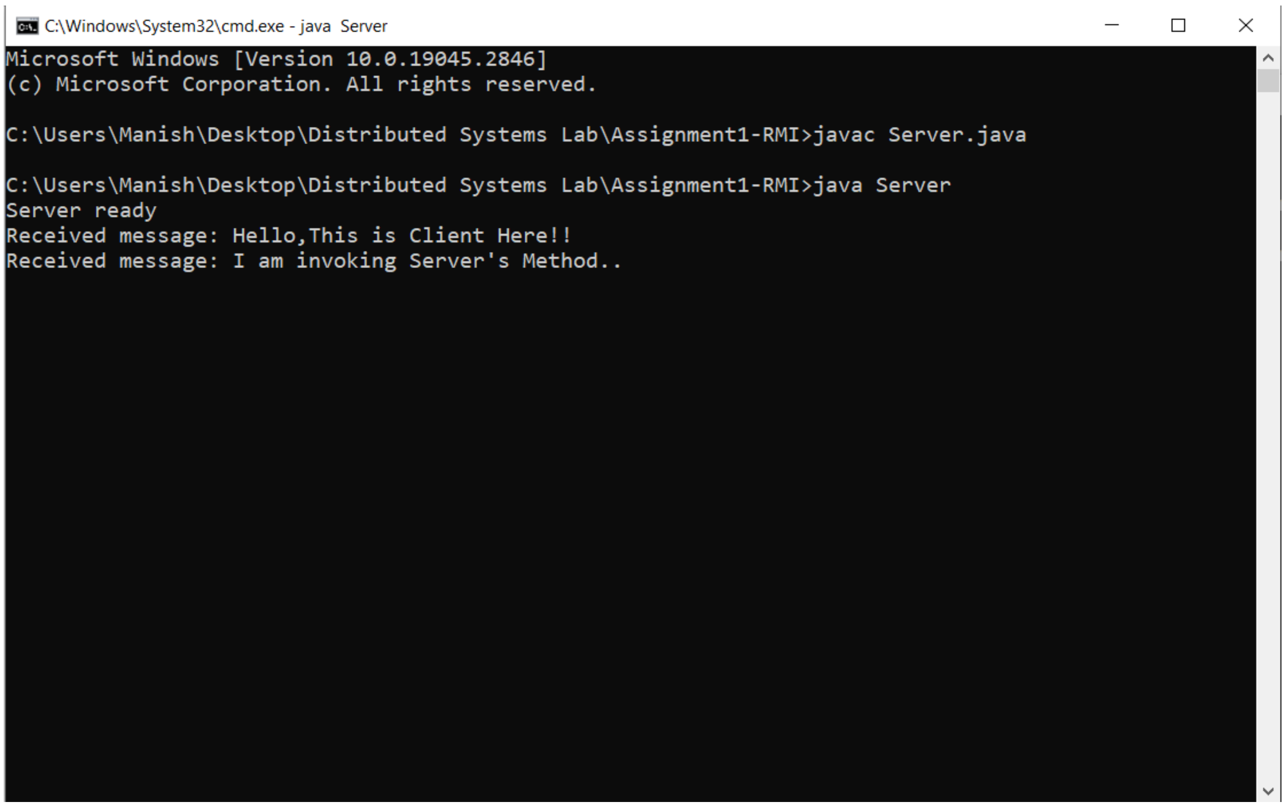
- ***Client.java***

```
import java.rmi.Naming;

import java.rmi.RemoteException;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;
import java.util.Scanner;


public class Client implements Runnable {
private Service service;
public Client(Service service) {
this.service = service;
}
public void run() {
Scanner scanner = new Scanner(System.in);
while (true) {
System.out.print("Enter message: ");
String message = scanner.nextLine();
try {
service.receiveMessage(message);
} catch (RemoteException e) {
System.out.println("Client exception: " + e.toString());
e.printStackTrace();
}
}
}
public static void main(String[] args) {
try {
Registry registry = LocateRegistry.getRegistry(1099);
Service service = (Service) Naming.lookup("rmi://localhost/Service");
Client client = new Client(service);
Thread thread = new Thread(client);
thread.start();
} catch (Exception e) {
System.out.println("Client exception: " + e.toString());
e.printStackTrace();
}
}
}
```

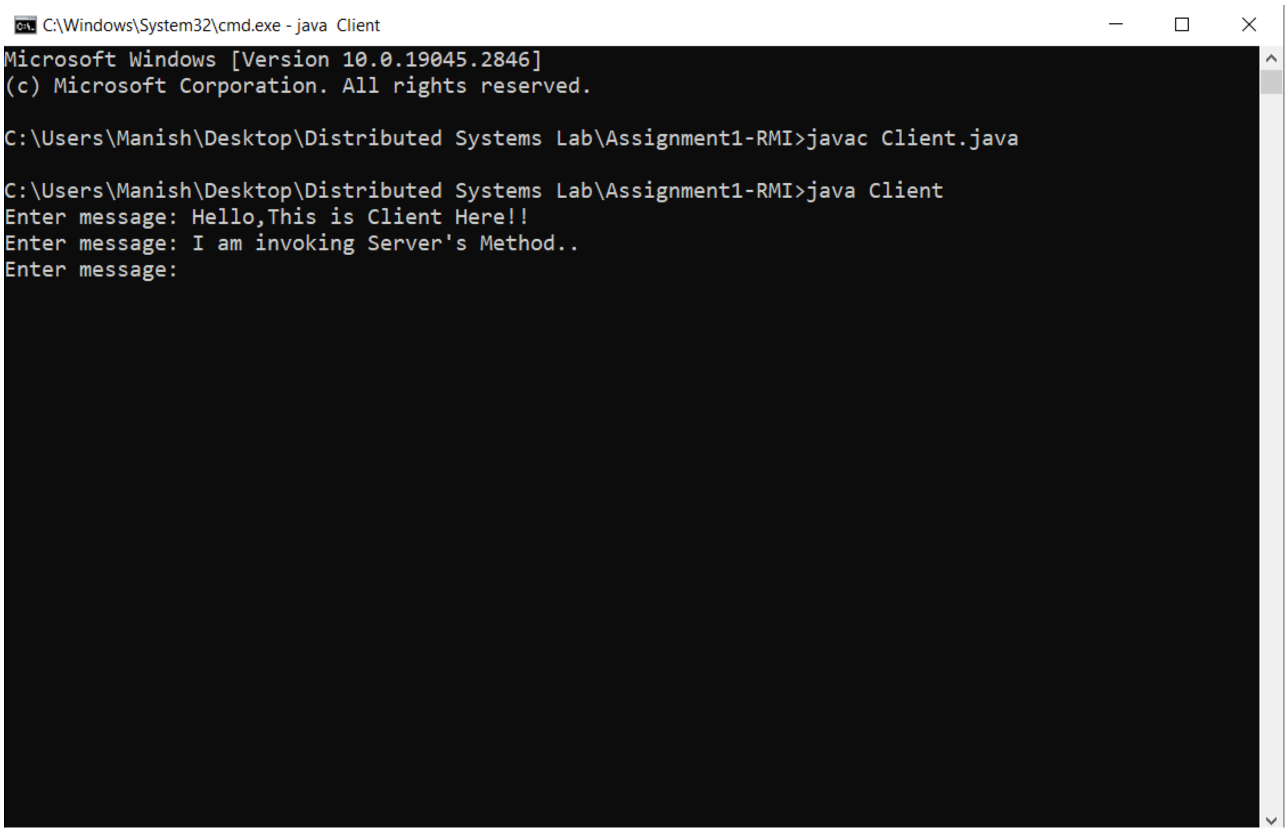
- ***Server.java***

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;

public class Server implements Service {
private ArrayList<String> messages = new ArrayList<>();
public Server() throws RemoteException {
UnicastRemoteObject.exportObject(this, 0);
}
public void receiveMessage(String message) throws RemoteException {
System.out.println("Received message: " + message);
messages.add(message);
}
public static void main(String[] args) {
try {
Server server = new Server();
Registry registry = LocateRegistry.createRegistry(1099);
Naming.rebind("rmi://localhost/Service", server);
System.out.println("Server ready");
} catch (Exception e) {
System.out.println("Server exception: " + e.toString());
e.printStackTrace();
}
}
}

interface Service extends java.rmi.Remote {
void receiveMessage(String message) throws RemoteException;
}
```

OUTPUT:



```
C:\Windows\System32\cmd.exe - java  Server
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Manish\Desktop\Distributed Systems Lab\Assignment1-RMI>javac Server.java

C:\Users\Manish\Desktop\Distributed Systems Lab\Assignment1-RMI>java Server
Server ready
Received message: Hello,This is Client Here!!
Received message: I am invoking Server's Method..
```



```
C:\Windows\System32\cmd.exe - java  Client
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Manish\Desktop\Distributed Systems Lab\Assignment1-RMI>javac Client.java

C:\Users\Manish\Desktop\Distributed Systems Lab\Assignment1-RMI>java Client
Enter message: Hello,This is Client Here!!
Enter message: I am invoking Server's Method..
Enter message:
```

CONCLUSION:

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

ASSIGNMENT NO.2

PROBLEM STATEMENT:

Develop any distributed application using CORBA to demonstrate object brokering.

(Calculator or String operations).

Tools / Environment:

Java Programming Environment, JDK 1.8

Theory:

Common Object Request Broker Architecture (CORBA):

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

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

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

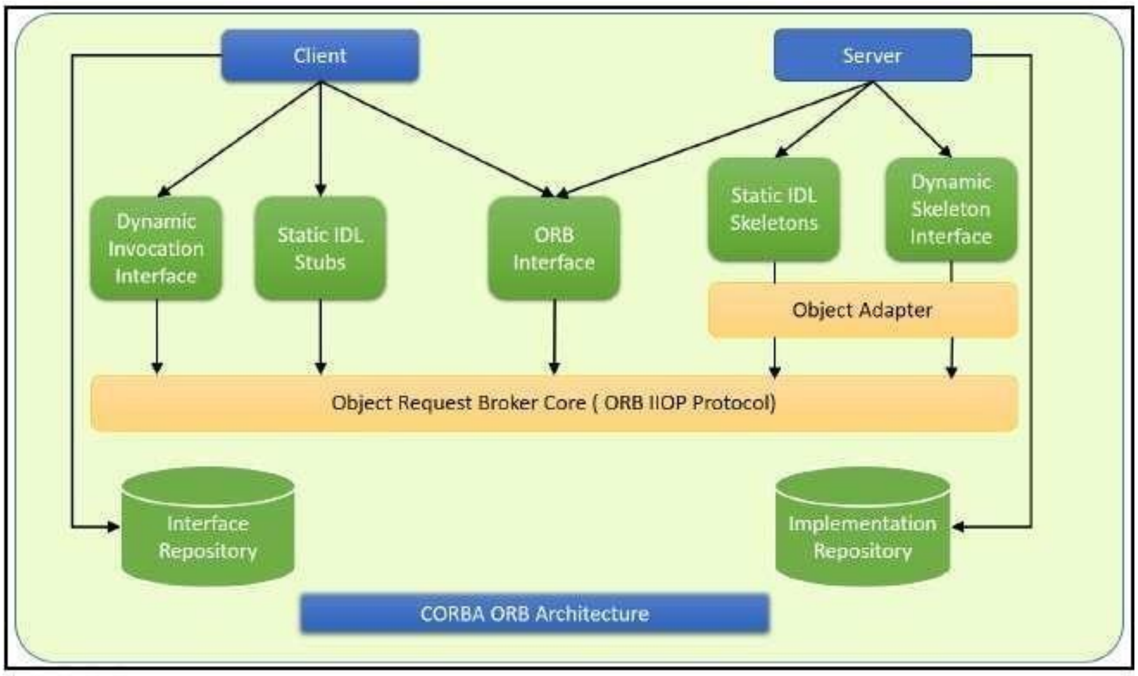
Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR&Benefits" maintain an object model with details of the organization, employees with demographic information, benefits, payroll, and deductions.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

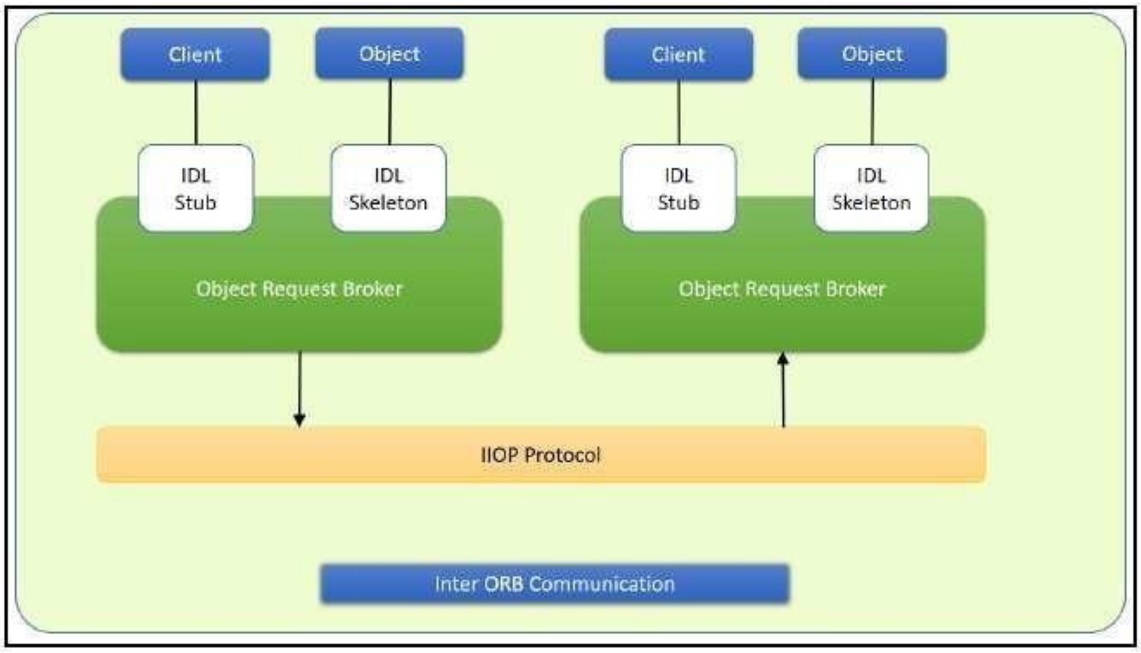
The client and server use stubs and skeletons as proxies, respectively. The IDL interface follows a strict definition, and even though the client and server are implemented in different technologies, they should integrate smoothly with the interface definition strictly implemented.


Client
Server
Dynamic Invocation Interface
Static IDL Stubs
ORB Interface
Static IDL Skeletons
Dynamic Skeleton Interface
Object Adapter
Object Request Broker Core (ORB IIOP Protocol)
Interface Repository
Implementation Repository
CORBA ORB Architecture

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 idljcompiler 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.omgprefix 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 idljcompiler. When you run the idljcompiler 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:

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

• 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 idljcompiler 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 idljcompiler, which uses a POA server-side model.

1. Creating CORBA Objects using Java IDL:

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

Modules

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

```
// IDL

module jen

{

module corba {

interface NeatExample ...

};

};
```

Interfaces

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

```
interface PrintServer : Server {

...
```

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

Data members and methods

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

readonly attribute string myString;

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

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

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

A complete IDL example

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

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

The first interface, Server, has a single read-only stringattribute and an init() method **20**

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 *Serveris* called *_ServerImplBase*.

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

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

3. Writing the Implementation

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

IMPLEMENTING THE SOLUTION:

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

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

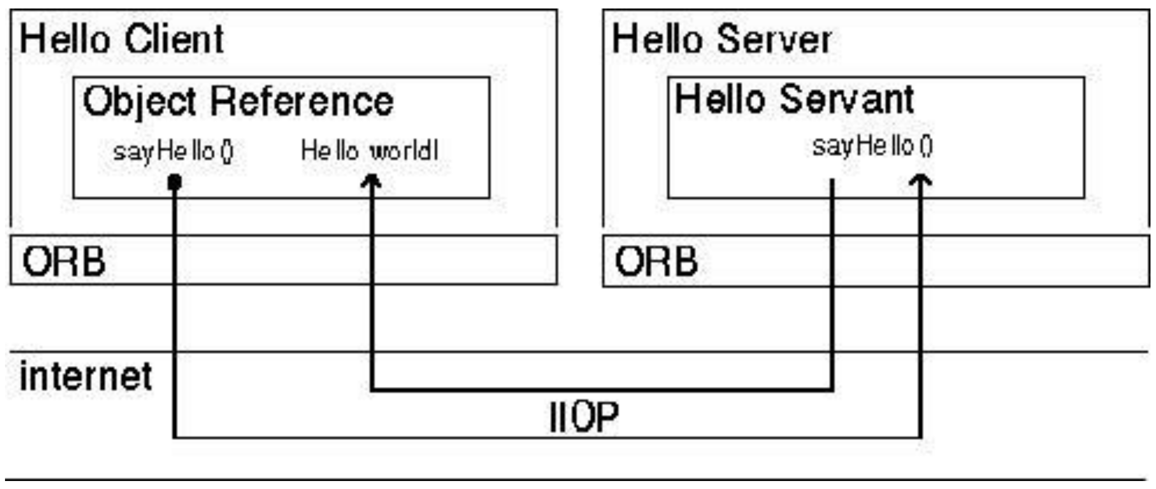
The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL).To complete the application, you simply provide the server **(HelloServer.java)** and client **(HelloClient.java)** implementations.

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

The example server consists of two classes, the servant and the server. The servant, *HelloImpl*, is the implementation of the *HelloIDL* interface; each *Hello instance* is implemented by a *HelloImpl instance*. The servant is a subclass of *HelloPOA*, which is generated by the *idljcompiler* 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 Helloobject) 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.

SOURCE CODE:

- ***ReverseClient.java***

```
import ReverseModule.*;

import org.omg.CosNaming.*;

import org.omg.CosNaming.NamingContextPackage.*;

import org.omg.CORBA.*;

import java.io.*;


class ReverseClient

{

  public static void main(String args[]){

    Reverse ReverseImpl=null;

    try{

        // initialize the ORB

        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

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

        NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

        String name = "Reverse";
```

```
            //Helper class provides narrow method that cast corba object reference (ref) into the java interface
            // System.out.println("Step2");
            // Look ups "Reverse" in the naming context
            ReverseImpl = ReverseHelper.narrow(ncRef.resolve_str(name));
            System.out.println("Enter String=");
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            String str= br.readLine();
            String tempStr= ReverseImpl.reverse_string(str);
            System.out.println(tempStr);
        }catch(Exception e){
            e.printStackTrace();
          }
    }
}
```

- ***ReverseImpl.java***

```
import ReverseModule.ReversePOA;
import java.lang.String;
class ReverseImpl extends ReversePOA
{
    ReverseImpl(){
        super();
        System.out.println("Reverse Object Created");
    }
    public String reverse_string(String name){
        StringBuffer str=new StringBuffer(name);
        str.reverse();
        return (("Server Send "+str));
    }
}
```

- ***ReverseModule.idl***

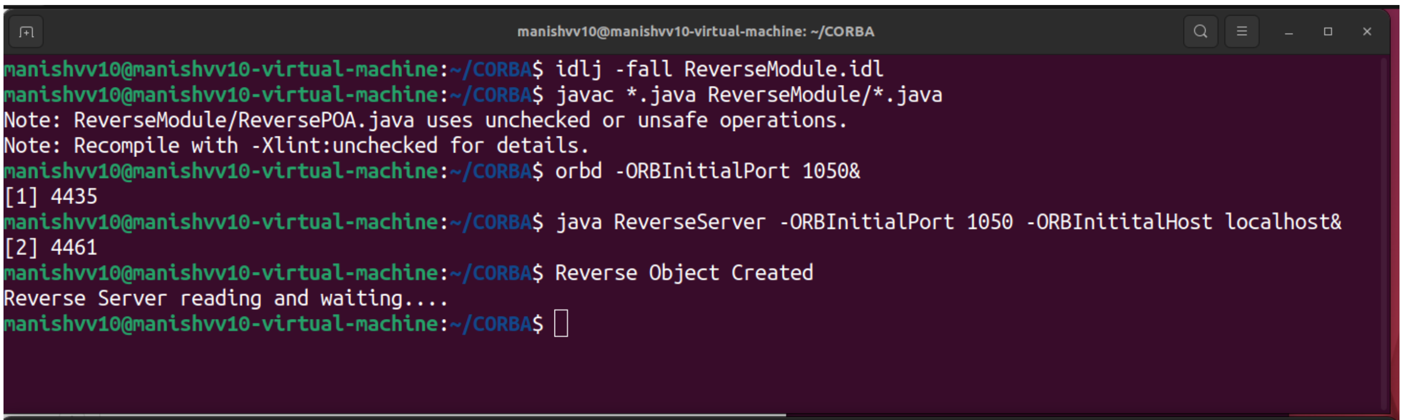
```
module ReverseModule //module ReverseModule is the name of the module
{
    interface Reverse{
        string reverse_string(in string str);
    };
};
```

- ***ReverseServer.java***

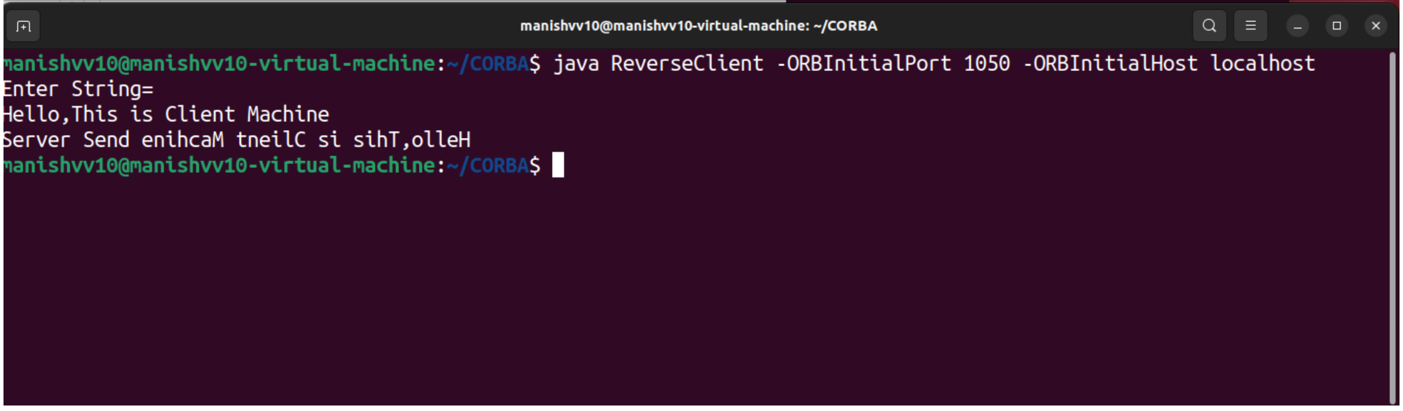
```
import ReverseModule.Reverse;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
class ReverseServer
{
    public static void main(String[] args)
    {
        try{
            // initialize the ORB
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            // initialize the portable object adaptor (BOA/POA) connects client request using object reference
            //uses orb method as resolve_initial_references
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootPOA.the_POAManager().activate();
            // creating an object of ReverseImpl class
            ReverseImpl rvr = new ReverseImpl();
            //server consist of 2 classes ,servent and server. The servent is the subclass of ReversePOA which is generated
by the idlj compiler
            // The servent ReverseImpl is the implementation of the ReverseModule idl interface
            // get the object reference from the servant class
            //use root POA class and its method servant_to_reference
            org.omg.CORBA.Object ref = rootPOA.servant_to_reference(rvr);
            // System.out.println("Step1");
            Reverse h_ref = ReverseModule.ReverseHelper.narrow(ref);// Helper class provides narrow method that cast
corba object reference (ref) into the java interface
            // System.out.println("Step2");
            // orb layer uses resolve_initial_references method to take initial reference as NameService
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            //Register new object in the naming context under the Reverse
            // System.out.println("Step3");
```

```
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      //System.out.println("Step4");

      String name = "Reverse";

      NameComponent path[] = ncRef.to_name(name);

      ncRef.rebind(path,h_ref);

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

      System.out.println("Reverse Server reading and waiting....");

      orb.run();

    }

    catch(Exception e){

      e.printStackTrace();

    }

  }

}
```

OUTPUT:



```
manishvv10@manishvv10-virtual-machine:~/CORBA$ idlj -fall ReverseModule.idl
manishvv10@manishvv10-virtual-machine:~/CORBA$ javac *.java ReverseModule/*.java
Note: ReverseModule/ReversePOA.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
manishvv10@manishvv10-virtual-machine:~/CORBA$ orbd -ORBInitialPort 1050&
[1] 4435
manishvv10@manishvv10-virtual-machine:~/CORBA$ java ReverseServer -ORBInitialPort 1050 -ORBInititalHost localhost&
[2] 4461
manishvv10@manishvv10-virtual-machine:~/CORBA$ Reverse Object Created
Reverse Server reading and waiting....
manishvv10@manishvv10-virtual-machine:~/CORBA$ 
```



```
manishvv10@manishvv10-virtual-machine:~/CORBA$ java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost
Enter String=
Hello,This is Client Machine
Server Send enihcaM tneilC si sihT,olleH
manishvv10@manishvv10-virtual-machine:~/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.

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:

C++ compiler (gcc), Ubuntu OS, MPI Library

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 message passing.

A basic prerequisite for message passing is a good communication API. Java comes with various ready-made packages for communication, notably an interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. The parallel computing world is mainly concerned with

`symmetric' communication, occurring in groups of interacting peers. This symmetric model of communication is captured in the successful Message Passing Interface standard (MPI).

Message-Passing Interface Basics:

Every MPI program must contain the preprocessor directive:

#include <mpi.h>

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

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

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

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

int MPI_Comm_size(MPI_Comm comm, int _size)

The function *"MPI_Comm_size"* required to return the number of processes; int size.

MPI_Comm_size(MPI_COMM_WORLD,&size);

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

int MPI_Comm_rank (MPI_Comm comm, int _rank).

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

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

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

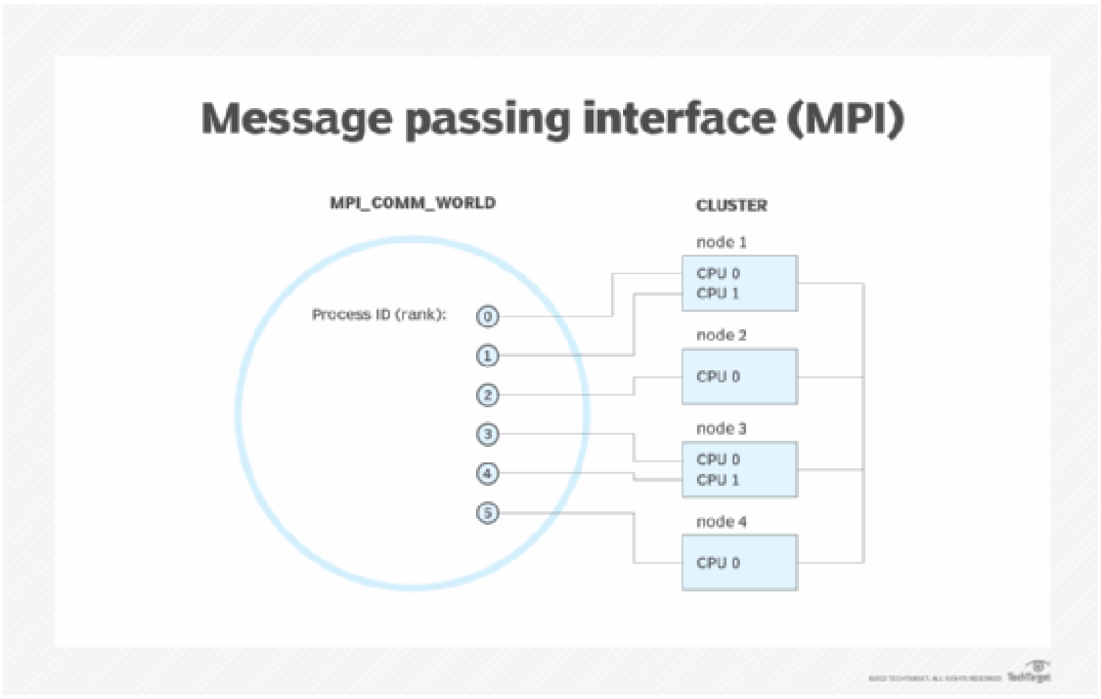
The receive command (MPI_Recv) describes both the data to be transferred and the connection to be established. The MPI_Recv construct is structured as follows:

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

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

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

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



MPJ Express is an open source Java message passing library that allows application developers to write and execute parallel applications for multicore processors and compute clusters

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

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

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

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

DESIGNING THE SOLUTION:

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

IMPLEMENTING THE SOLUTION:

1. For implementing the MPI program in multi-core environment, we need to install mpich library on ubuntu
    a. In the terminal execute the command ***sudo apt install mpich***
2. Write c program implementing the code and save it with .c extension
3. Compile and run the program using following code:

mpicc program_name.c -o object_file

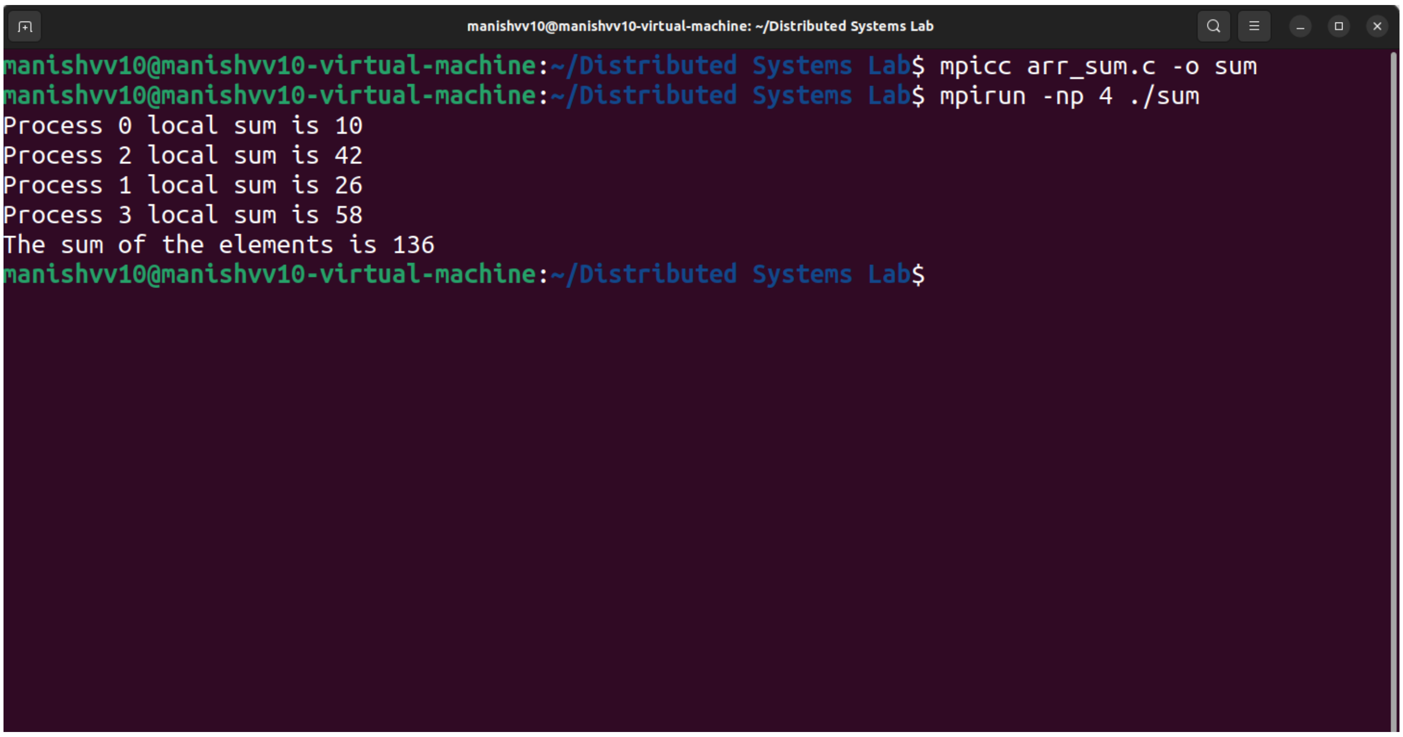
4. Running the program using MPI by executing the following command -

mpirun -np [number of processes] ./object_file

SOURCE CODE:

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

#define ARRAY_SIZE 16

int main(int argc, char** argv) {
  int rank, size;
  int sum = 0;
  int array[ARRAY_SIZE];

  // Initialize MPI
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  // Populate the array on the root process
  if (rank == 0) {
    for (int i = 0; i < ARRAY_SIZE; i++) {
      array[i] = i + 1;
    }
  }

  // Scatter the array to all processes
  int subarray_size = ARRAY_SIZE / size;
  int subarray[subarray_size];
  MPI_Scatter(array, subarray_size, MPI_INT, subarray, subarray_size, MPI_INT, 0, MPI_COMM_WORLD);

  // Sum the local elements
  int local_sum = 0;
  for (int i = 0; i < subarray_size; i++) {
    local_sum += subarray[i];
  }

  // Display the local sum of each process
  printf("Process %d local sum is %d\n", rank, local_sum);

  // Reduce the local sums to get the final sum on the root process
  MPI_Reduce(&local_sum, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  // Print the result on the root process
  if (rank == 0) {
    printf("The sum of the elements is %d\n", sum);
  }

  // Finalize MPI
  MPI_Finalize();
  return 0;
}
```

OUTPUT:



manishvv10@manishvv10-virtual-machine: ~/Distributed Systems Lab

```
manishvv10@manishvv10-virtual-machine:~/Distributed Systems Lab$ mpicc arr_sum.c -o sum
manishvv10@manishvv10-virtual-machine:~/Distributed Systems Lab$ mpirun -np 4 ./sum
Process 0 local sum is 10
Process 2 local sum is 42
Process 1 local sum is 26
Process 3 local sum is 58
The sum of the elements is 136
manishvv10@manishvv10-virtual-machine:~/Distributed Systems Lab$
```

CONCLUSION:

There has been a large amount of interest in parallel programming. mpich is an MPI binding with C language along with the support for multicore architecture so that user can develop the code on its own laptop or desktop. This is an effort to develop and run parallel programs according to MPI standard.

ASSIGNMENT NO. 4

PROBLEM STATEMENT:

Implement Berkeley algorithm for clock synchronization.

TOOLS / ENVIRONMENT:

Anaconda (Jupiter Notebook/Spyder/Visual studio), Python, VS Code

THEORY:

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

1. The machine does not have an accurate time source.
2. The network or machine does not have a UTC server.

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

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

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

Berkeley's Algorithm:

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

The algorithm is −

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

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

Nodes in the distributed system with their clock timings −

N1 -> 14:00 (master node)

N2 -> 13: 46

N3 -> 14: 15

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

Step 2- leader requests for time from all nodes.

N1 -> time: 14:00

N2 -> time: 13:46

N3 -> time: 14:20

Step 3- The leader averages the times and sends the correction time back to the nodes.

N1 -> Corrected Time 14:02 (+2)

N2 -> Corrected Time 14:02 (+16)

N3 -> Corrected Time 14:02 (-18)

This shows how the synchronization of nodes of a distributed system is done using Berkeley’s algorithm.

SOURCE CODE:

- ***Server.py***

```
# Python3 program imitating a clock server
from functools import reduce
from dateutil import parser
import threading
import datetime
import socket
import time
# datastructure used to store client address and clock data
client_data = {}
''' nested thread function used to receive
   clock time from a connected client '''
def startReceivingClockTime(connector, address):
   while True:
      # receive clock time
      clock_time_string = connector.recv(1024).decode()
      clock_time = parser.parse(clock_time_string)
      clock_time_diff = datetime.datetime.now() - \
         clock_time

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

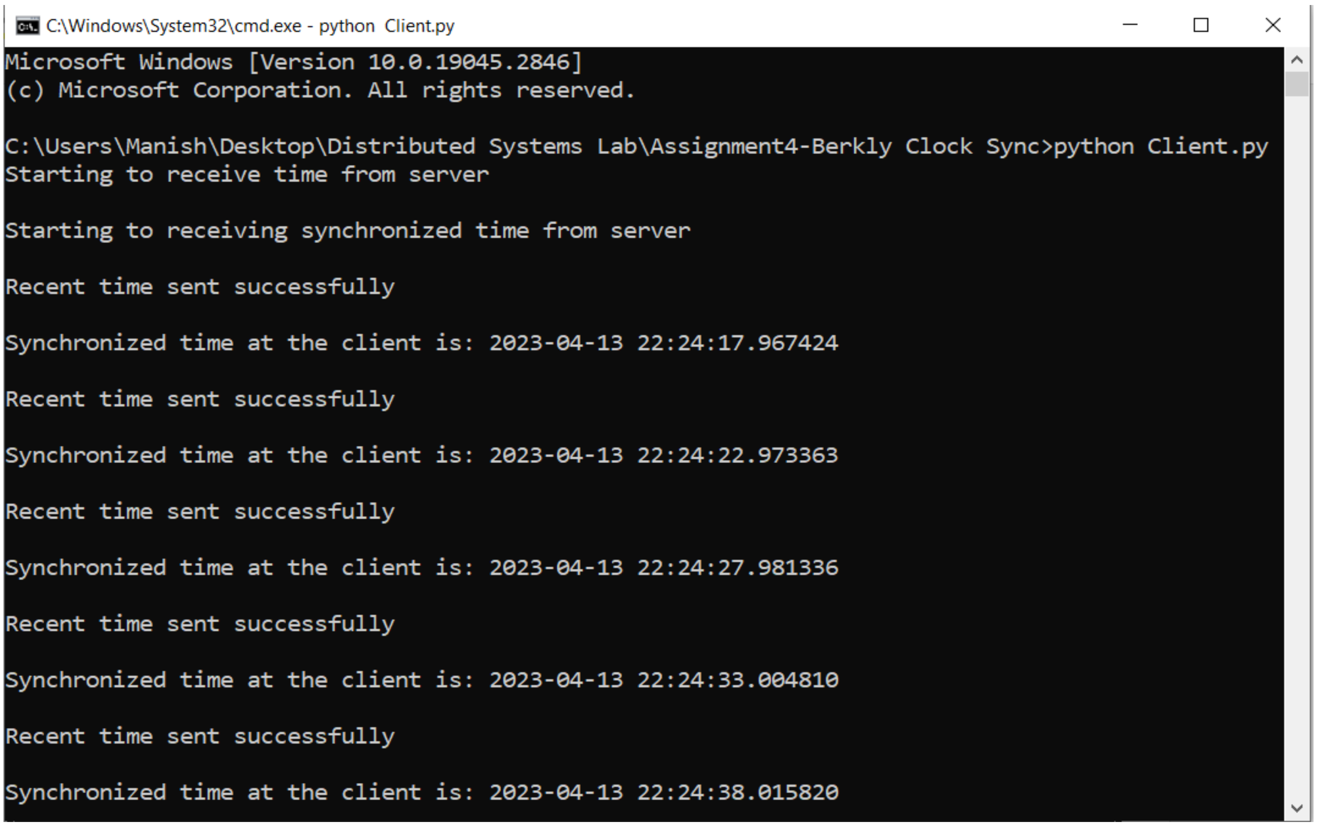
```
        print("Client Data updated with: " + str(address),
              end="\n\n")
        time.sleep(5)
''' master thread function used to open portal for
    accepting clients over given port '''
def startConnecting(master_server):
    # fetch clock time at slaves / clients
    while True:
        # accepting a client / slave clock client
        master_slave_connector, addr = master_server.accept()
        slave_address = str(addr[0]) + ":" + str(addr[1])
        print(slave_address + " got connected successfully")
        current_thread = threading.Thread(
            target=startReceivingClockTime,
            args=(master_slave_connector,
                  slave_address, ))
        current_thread.start()
# subroutine function used to fetch average clock difference
def getAverageClockDiff():
    current_client_data = client_data.copy()
    time_difference_list = list(client['time_difference']
                            for client_addr, client
                            in client_data.items())
    sum_of_clock_difference = sum(time_difference_list,
                              datetime.timedelta(0, 0))

    average_clock_difference = sum_of_clock_difference \
        / len(client_data)

    return average_clock_difference
''' master sync thread function used to generate
    cycles of clock synchronization in the network '''
def synchronizeAllClocks():
    while True:
        print("New synchronization cycle started.")
        print("Number of clients to be synchronized: " +
              str(len(client_data)))
        if len(client_data) > 0:
          average_clock_difference = getAverageClockDiff()
          for client_addr, client in client_data.items():
            try:
              synchronized_time = \
                datetime.datetime.now() + \
                average_clock_difference
              client['connector'].send(str(
                synchronized_time).encode())
            except Exception as e:
              print("Something went wrong while " +
                    "sending synchronized time " +
                    "through " + str(client_addr))
        else:
          print("No client data." +
                " Synchronization not applicable.")
        print("\n\n")
        time.sleep(5)
# function used to initiate the Clock Server / Master Node
def initiateClockServer(port=8080):
    master_server = socket.socket()
    master_server.setsockopt(socket.SOL_SOCKET,
                             socket.SO_REUSEADDR, 1)
    print("Socket at master node created successfully\n")
    master_server.bind(('', port))
    # Start listening to requests
    master_server.listen(10)
    print("Clock server started...\n")
    # start making connections
    print("Starting to make connections...\n")
    master_thread = threading.Thread(
        target=startConnecting,
        args=(master_server, ))
    master_thread.start()
    # start synchronization
    print("Starting synchronization parallelly...\n")
    sync_thread = threading.Thread(
        target=synchronizeAllClocks,
        args=())
    sync_thread.start()
# Driver function
if __name__ == '__main__':
    # Trigger the Clock Server
    initiateClockServer(port=8080)
```

- ***Client.py***

```
# Python3 program imitating a client process
from timeit import default_timer as timer
from dateutil import parser
import threading
import datetime
import socket
import time
# client thread function used to send time at client side
def startSendingTime(slave_client):
    while True:
        # provide server with clock time at the client
        slave_client.send(str(
            datetime.datetime.now()).encode())
        print("Recent time sent successfully",
              end="\n\n")
        time.sleep(5)
# client thread function used to receive synchronized time
def startReceivingTime(slave_client):
    while True:
        # receive data from the server
        Synchronized_time = parser.parse(
            slave_client.recv(1024).decode())
        print("Synchronized time at the client is: " +
             str(Synchronized_time),
             end="\n\n")
# function used to Synchronize client process time
def initiateSlaveClient(port=8080):
    slave_client = socket.socket()
    # connect to the clock server on local computer
    slave_client.connect(('127.0.0.1', port))
    # start sending time to server
    print("Starting to receive time from server\n")
    send_time_thread = threading.Thread(
        target=startSendingTime,
        args=(slave_client, ))
    send_time_thread.start()
    # start receiving synchronized from server
    print("Starting to receiving " +
          "synchronized time from server\n")
    receive_time_thread = threading.Thread(
        target=startReceivingTime,
        args=(slave_client, ))
    receive_time_thread.start()


# Driver function
if __name__ == '__main__':
    # initialize the Slave / Client
    initiateSlaveClient(port=8080)
```

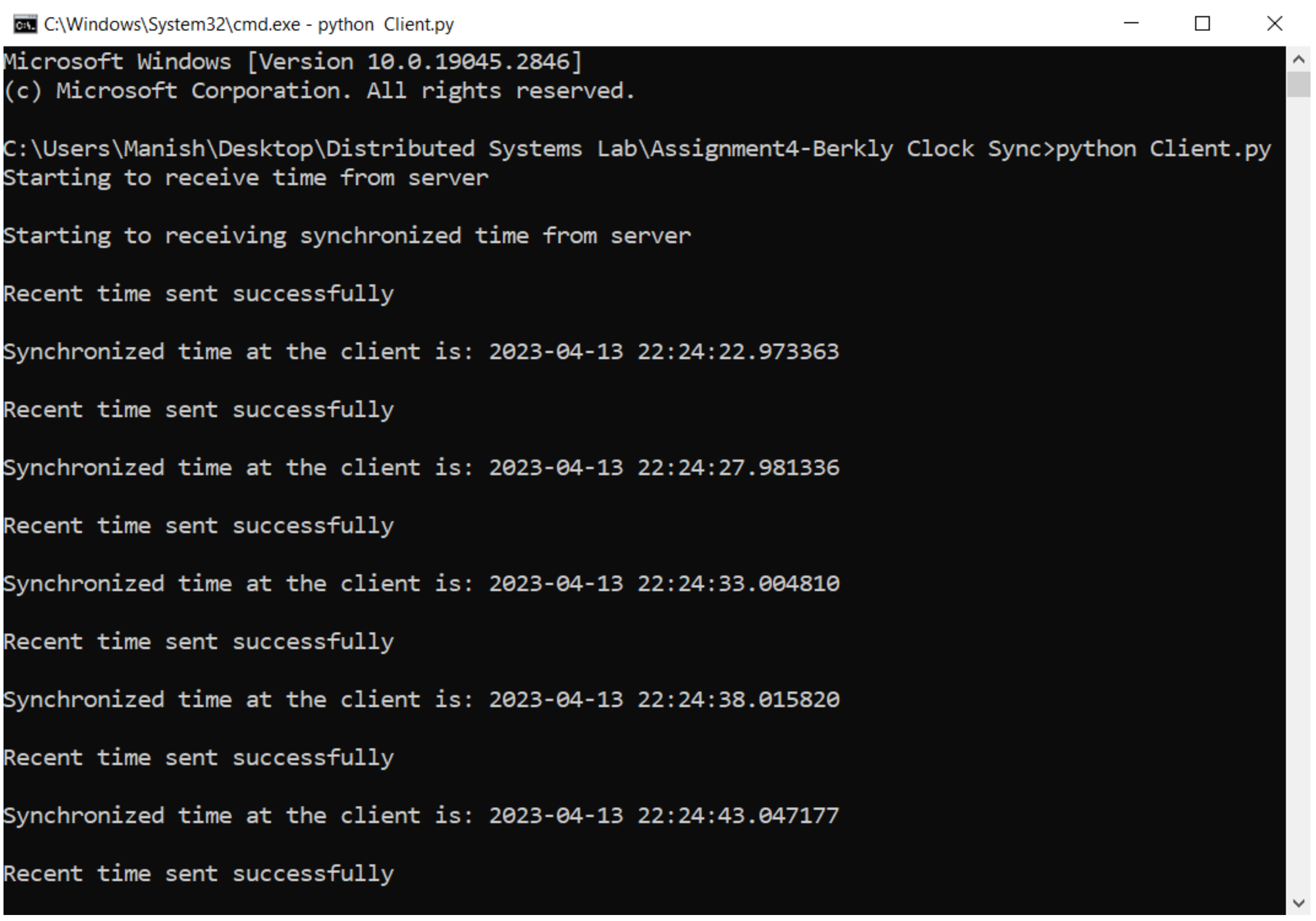
OUTPUT:

- **Client 1:**



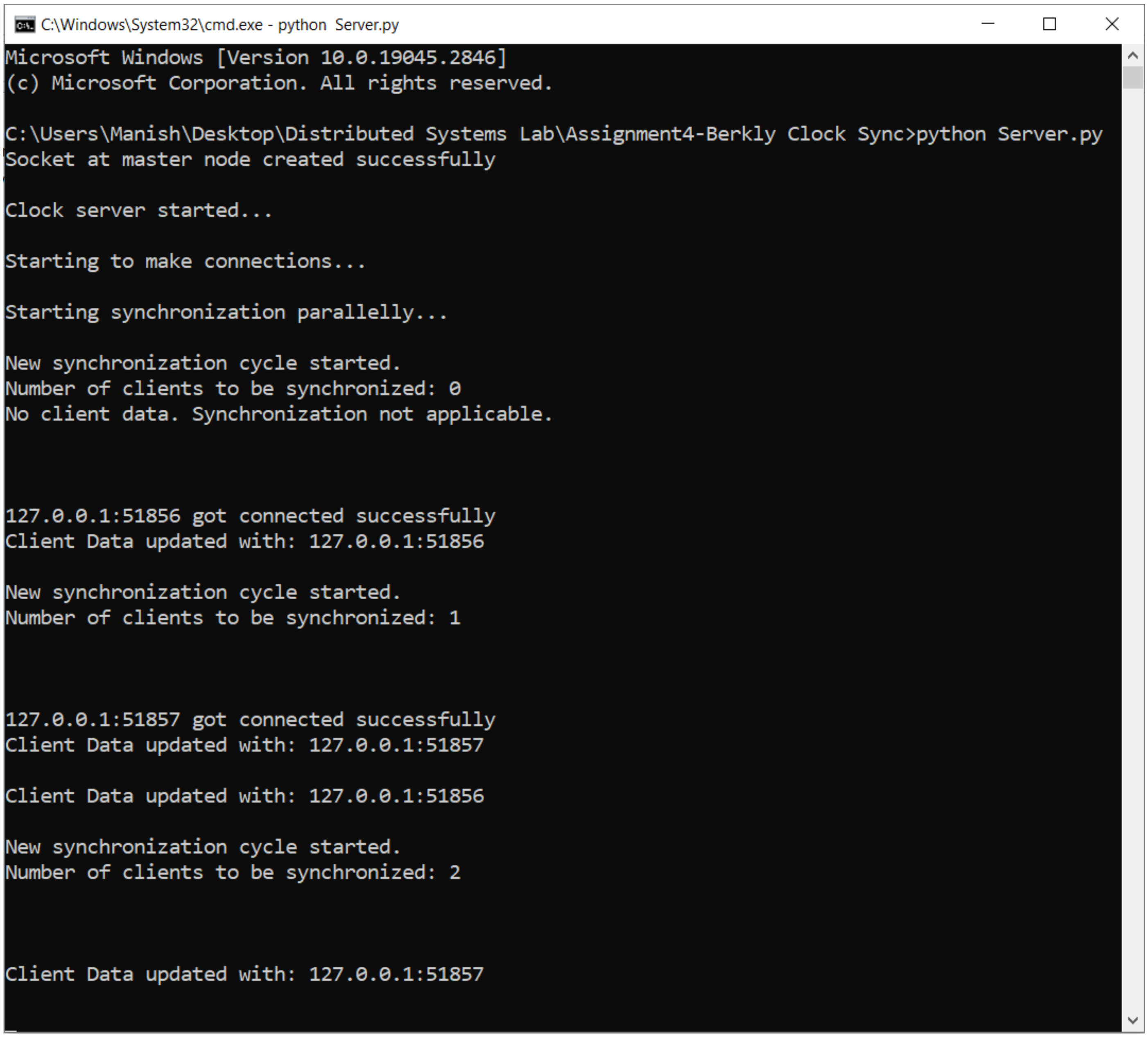
```
C:\Windows\System32\cmd.exe - python  Client.py
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Manish\Desktop\Distributed Systems Lab\Assignment4-Berkly Clock Sync>python Client.py
Starting to receive time from server

Starting to receiving synchronized time from server

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:17.967424

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:22.973363

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:27.981336

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:33.004810

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:38.015820
```

- **Client 2**



```
C:\Windows\System32\cmd.exe - python  Client.py
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Manish\Desktop\Distributed Systems Lab\Assignment4-Berkly Clock Sync>python Client.py
Starting to receive time from server

Starting to receiving synchronized time from server

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:22.973363

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:27.981336

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:33.004810

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:38.015820

Recent time sent successfully

Synchronized time at the client is: 2023-04-13 22:24:43.047177

Recent time sent successfully
```

➢ **Server**



C:\Windows\System32\cmd.exe - python Server.py

```
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Manish\Desktop\Distributed Systems Lab\Assignment4-Berkly Clock Sync>python Server.py
Socket at master node created successfully

Clock server started...

Starting to make connections...

Starting synchronization parallelly...

New synchronization cycle started.
Number of clients to be synchronized: 0
No client data. Synchronization not applicable.



127.0.0.1:51856 got connected successfully
Client Data updated with: 127.0.0.1:51856

New synchronization cycle started.
Number of clients to be synchronized: 1



127.0.0.1:51857 got connected successfully
Client Data updated with: 127.0.0.1:51857

Client Data updated with: 127.0.0.1:51856

New synchronization cycle started.
Number of clients to be synchronized: 2



Client Data updated with: 127.0.0.1:51857

_
```

CONCLUSION:

Thus, we have successfully implemented the Berkeley clock synchronization algorithm.

ASSIGNMENT NO. 5

PROBLEM STATEMENT:

Implement token ring based mutual exclusion algorithm

TOOLS / ENVIRONMENT:

Anaconda/Jupiter Notebook/Spyder/Visual studio, Python, VS Code

THEORY:

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

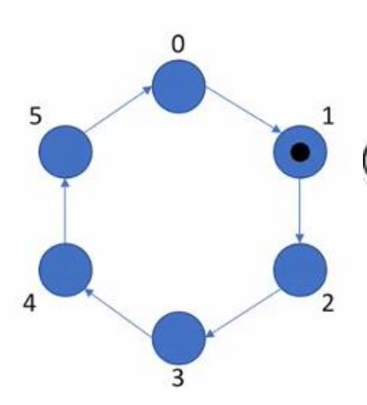
Thing we might want to guarantee for DME specifications are:

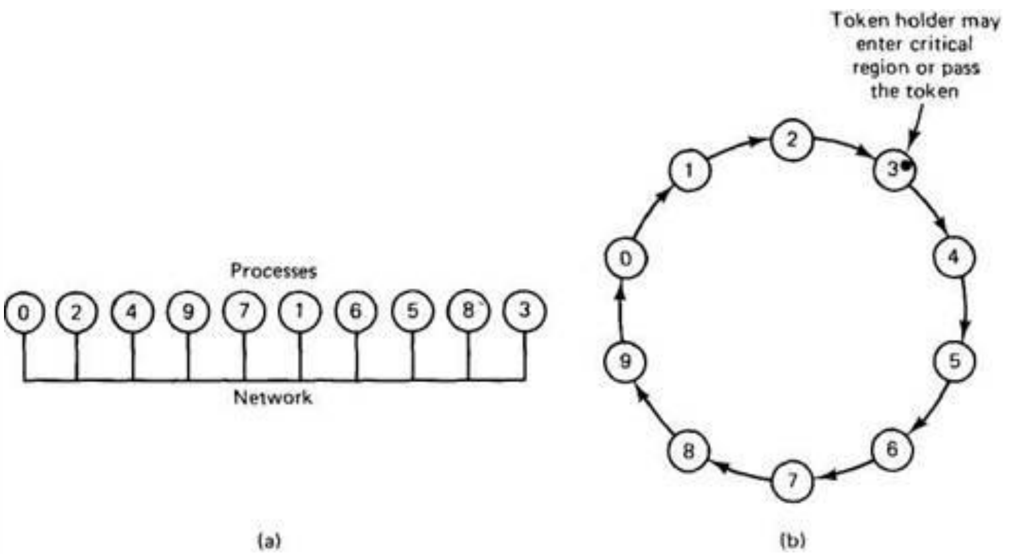
1. **Mutual exclusion,** at most one client is in a critical section (always)
2. **Non-starvation,** A requesting client enters critical section eventually (usually)
3. **Non-overtaking,** A client cannot enter critical section more than once while another client waits (usually)

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

The algorithm works as follows:

- When the ring is initialized, process 0 is given a token. The token circulates around the ring. When a process acquires the token from its neighbour, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token to the next process in the ring. It is not allowed to enter the critical region again using the same token. If a process is handed the token by its neighbour and is not interested in entering a critical region, it just passes the token along to the next process.





In this algorithm it is assumed that all the processes in the system are organized in a logical ring.

The ring positions may be allocated in numerical order of network addresses and is unidirectional in the sense that all messages are passed only in clockwise or anti-clockwise direction.

When a process sends a request message to current coordinator and does not receive a reply within a fixed timeout, it assumes the coordinator has crashed. It then initializes the ring and process Pi is given a token.

The token circulates around the ring. It is passed from process k to k+1 in point-to-point messages. When a process acquires the token from its neighbour it checks to see if it is attempting to enter a critical region. If so, the process enters the region does all the execution and leaves the region. After it has exited it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

If a process is handed the token by its neighbour and is not interested in entering a critical region it just passes along. When no processes want to enter any critical regions, the token just circulates at high speed around the ring.

Only one process has the token at any instant so only one process can actually be in a critical region. Since the token circulates among the process in a well-defined order, starvation cannot occur.

Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.

The disadvantage is that if the token is lost it must be regenerated. But the detection of lost token is difficult. If the token is not received for a long time, it might not be lost but is in use.

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.

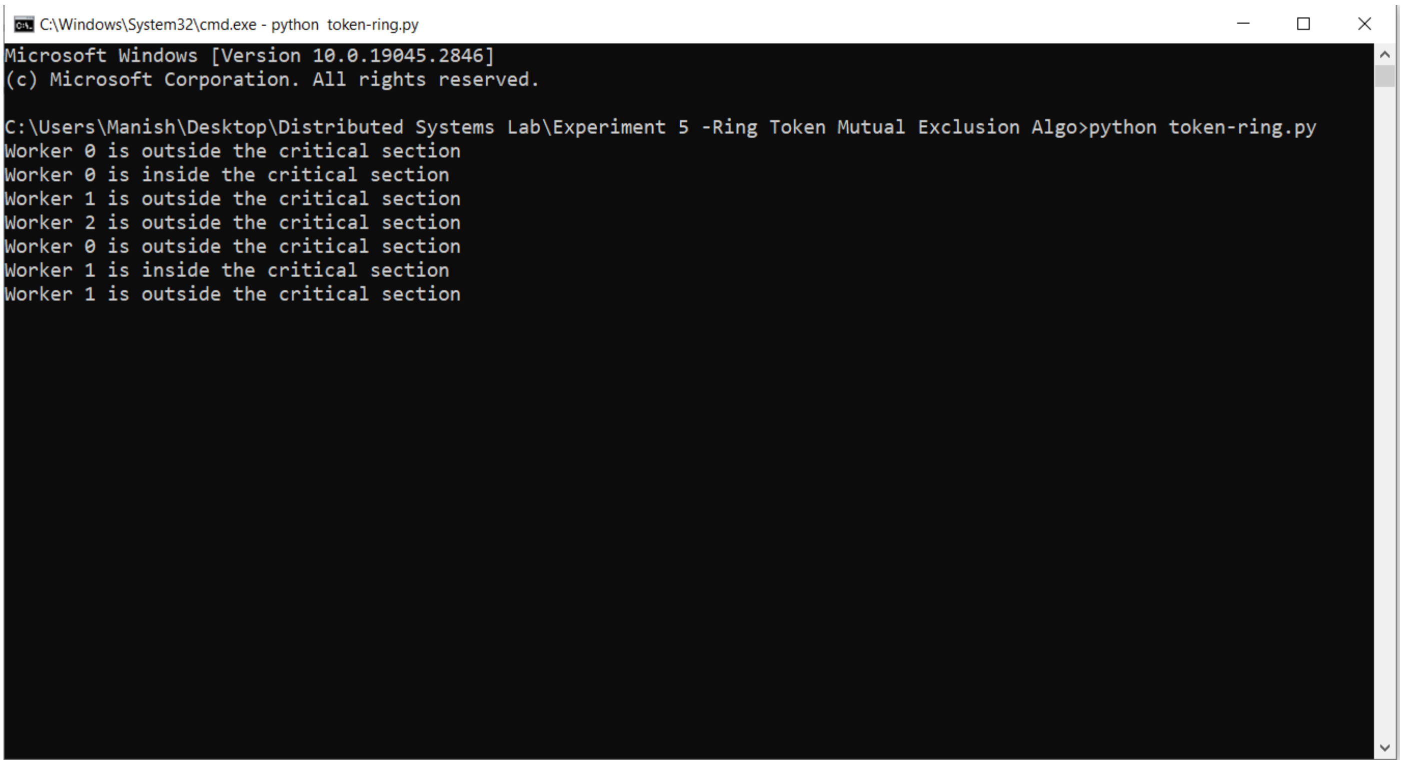
Disadvantages:

- 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 a 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 neighbour 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

SOURCE CODE:

```
import threading

import time


class TokenRingMutex:

    def __init__(self, n):

        self.tokens = [threading.Event() for _ in range(n)]

        self.tokens[0].set()

        self.n = n

        self.queue = []
    def request_critical_section(self):
        self.queue.append(threading.current_thread().ident)
        while True:
            token_idx = self.queue.index(threading.current_thread().ident)
            self.tokens[token_idx % self.n].wait()
            if token_idx == 0:
                return

    def release_critical_section(self):
        token_idx = self.queue.index(threading.current_thread().ident)
        self.tokens[(token_idx + 1) % self.n].set()
        self.queue.remove(threading.current_thread().ident)

def worker(mutex, id):
    while True:
        print(f"Worker {id} is outside the critical section")
        mutex.request_critical_section()
        print(f"Worker {id} is inside the critical section")
        time.sleep(1)
        mutex.release_critical_section()

if __name__ == "__main__":
    mutex = TokenRingMutex(3)
    workers = []
    for i in range(3):
        worker_thread = threading.Thread(target=worker, args=(mutex, i))
        workers.append(worker_thread)
        worker_thread.start()

    for worker_thread in workers:
        worker_thread.join()
```

OUTPUT:


C:\Windows\System32\cmd.exe - python token-ring.py

```
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Manish\Desktop\Distributed Systems Lab\Experiment 5 -Ring Token Mutual Exclusion Algo>python token-ring.py
Worker 0 is outside the critical section
Worker 0 is inside the critical section
Worker 1 is outside the critical section
Worker 2 is outside the critical section
Worker 0 is outside the critical section
Worker 1 is inside the critical section
Worker 1 is outside the critical section
```

CONCLUSION:

Thus, we have successfully implemented the token ring based mutual exclusion algorithm.

ASSIGNMENT NO. 6

PROBLEM STATEMENT:

Implement Bully and Ring algorithm for leader election

TOOLS / ENVIRONMENT:

Anaconda (Jupiter Notebook/Spyder), Python, VS Code

THEORY:

Election Algorithm:

1. Many distributed algorithms require a process to act as a coordinator.

2. The coordinator can be any process that organizes actions of other processes.

3. A coordinator may fail.

4. How is a new coordinator chosen or elected?

Assumptions:

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

There are two types of Distributed Algorithms:

1. Bully Algorithm
2. Ring Algorithm

1. 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 apart of one, and that one is the new coordinator.

3) The new coordinator announces its victory by sending all processes **a CO-ORDINATOR**

message telling them that it is the new coordinator.

C. If a process that was previously down comes back:

1) It holds an election.

2) If it happens to be the highest process currently running, it will win the election and take over the coordinators job.

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

2. Ring Algorithm:

A. Initiation:

1. When a process notices that coordinator is not functioning:
2. Another process (initiator) initiates the election by sending "ELECTION" message (containing its own process number)

B. Leader Election:

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

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

DESIGNING THE SOLUTION:

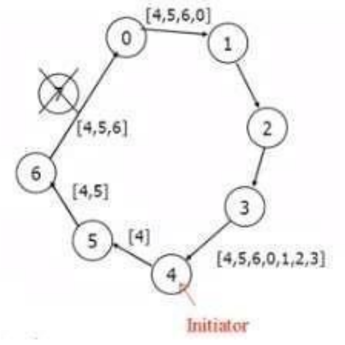
A. For Ring Algorithm

Initiation:

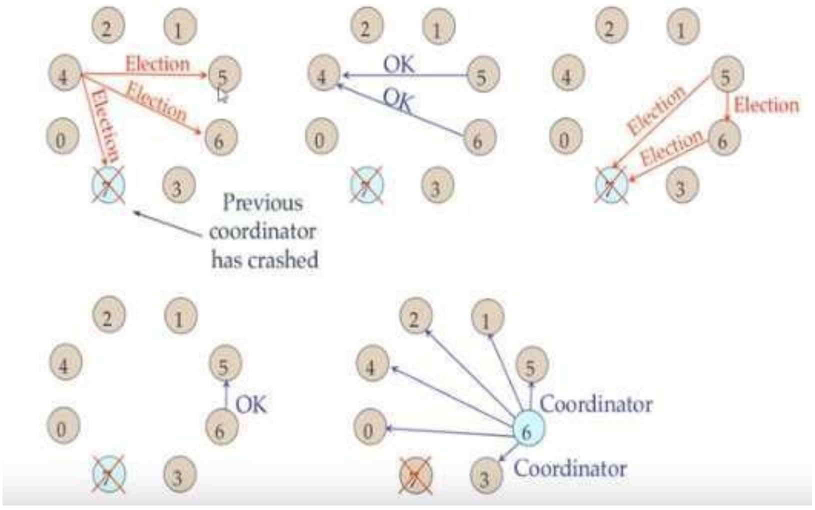
1) Consider the Process 4 understands that Process 7 is not responding.
2) Process 4 initiates the Election by sending "ELECTION" message to its successor (or next alive process) with its ID.

Leader Election:

3. Messages comes back to initiator. Here the initiator is 4.
4. Initiator announces the winner by sending another message around the ring. Here the process with highest process ID is 6. The initiator will announce that Process 6 is Coordinator.



B. For Bully Algorithm:



IMPLEMENTING THE SOLUTION:

For Ring Algorithm:

1. Creating Class for Process which includes
   - i) State: Active / Inactive
   - ii) Index: Stores index of process.
   - iii) ID: Process ID
2. Import Scanner Class for getting input from Console
3. Getting input from User for number of Processes and store them into object of classes.
4. Sort these objects on the basis of process id.
5. Make the last process id as "inactive".
6. Ask for menu
   - i) Election
   - ii) Exit
7. Ask for initializing election process.
8. These inputs will be used by Ring Algorithm.

SOURCE CODE:

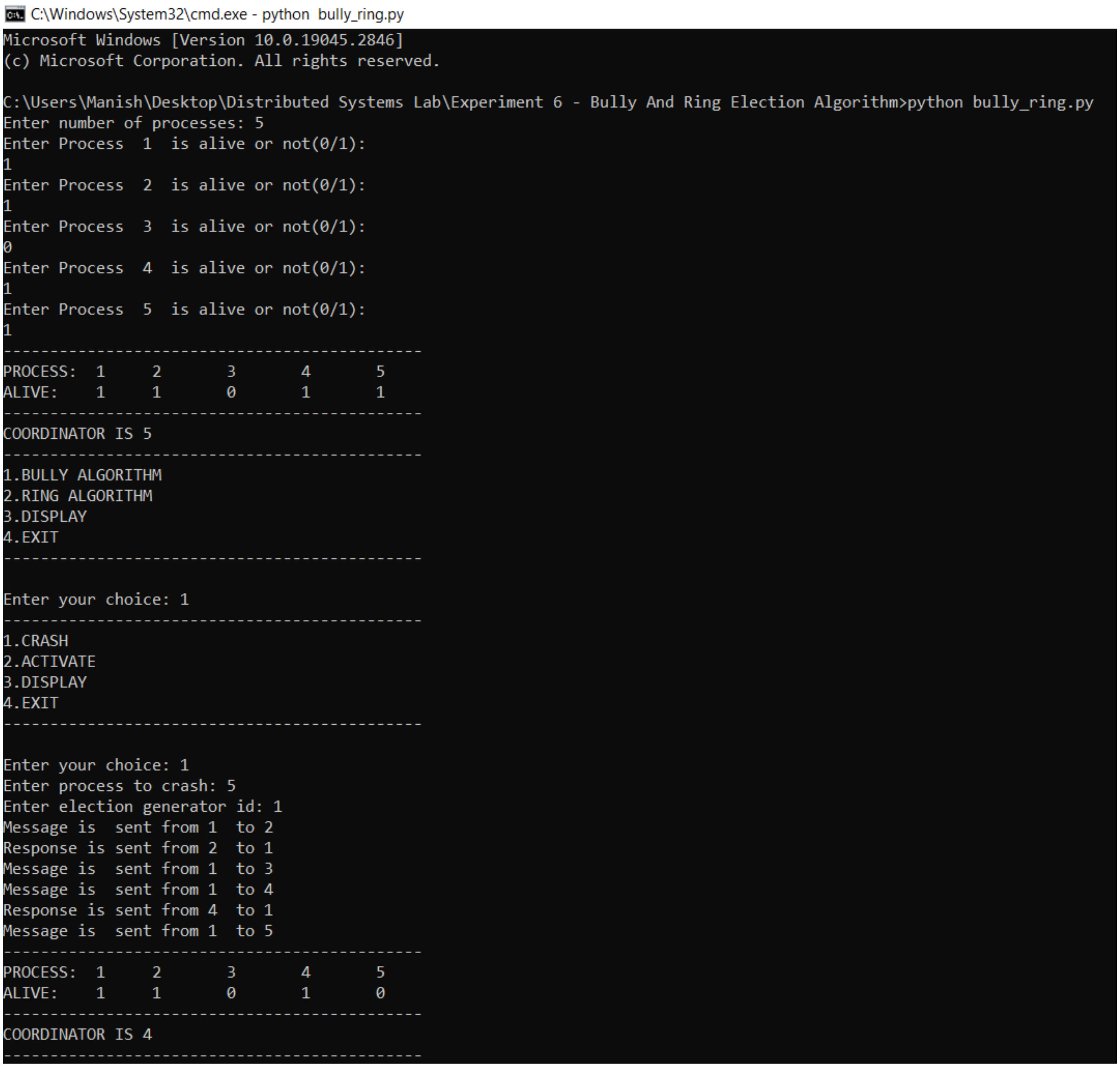
```
# we define MAX as the maximum number of processes our program can simulate

# we declare pStatus to store the process status; 0 for dead and 1 for alive

# we declare n as the number of processes

# we declare coordinator to store the winner of election


MAX = 20

pStatus = [0 for _ in range(MAX)]

n = 0

coordinator = 0


def bully():

   " bully election implementation"

   global coordinator

   condition = True

   while condition:

      print('---------------------------------------------')

      print("1.CRASH\n2.ACTIVATE\n3.DISPLAY\n4.EXIT")
print('---------------------------------------------\n')
print("Enter your choice: ", end='')
schoice = int(input())


if schoice == 1:
   # we manually crash the process to see if our implementation
   # can elect another leader
   print("Enter process to crash: ", end='')
   crash = int(input())
   # if the process is alive then set its status to dead
   if (pStatus[crash] != 0):
      pStatus[crash] = 0
   else:
      print('Process', crash, ' is already dead!\n')
      break
   condition = True
   while condition:
      # enter another process to initiate the election
      print("Enter election generator id: ", end='')
      gid = int(input())
      if (gid == coordinator or pStatus[gid] == 0):
         print("Enter a valid generator id!")
      condition = (gid == coordinator or pStatus[gid] == 0)
   flag = 0
   # if the coordinator has crashed then we need to find another leader
   if (crash == coordinator):
    # the election generator process will send the message to all higher process
      i = gid + 1
      while i <= n:
         print("Message is  sent from", gid, " to", i, end='\n')
         # if the higher process is alive then it will respond
         if (pStatus[i] != 0):
            subcoordinator = i
            print("Response is sent from", i, " to", gid, end='\n')
```

```
                flag = 1
            i += 1
        # the highest responding process is selected as the leader
        if (flag == 1):
            coordinator = subcoordinator
        # else if no higher process are alive then the election generator process
        # is selected as leader
        else:
            coordinator = gid
    display()


elif schoice == 2:
    # enter process to revive
    print("Enter Process ID to be activated: ", end='')
    activate = int(input())
    # if the entered process was dead then it is revived
    if (pStatus[activate] == 0):
        pStatus[activate] = 1
    else:
        print("Process", activate, " is already alive!", end='\n')
        break
    # if the highest process is activated then it is the leader
    if (activate == n):
        coordinator = n
        break
    flag = 0
    # else, the activated process sends message to all higher process
    i = activate + 1
    while i <= n:
        print("Message is  sent from", activate, "to", i, end='\n')
        # if higher process is active then it responds
        if (pStatus[i] != 0):
            subcoordinator = i
            print("Response is sent from", i,
                "to", activate, end='\n')
             flag = 1
          i += 1
       # the highest responding process is made the leader
       if flag == 1:
          coordinator = subcoordinator
       # if no higher process respond then the activated process is leader
       else:
          coordinator = activate
       display()
    elif schoice == 3:
       display()
    elif schoice == 4:
       pass
    condition = (schoice != 4)
def ring():
 " ring election implementation"
 global coordinator, n
 condition = True
 while condition:
    print('---------------------------------------------')
    print("1.CRASH\n2.ACTIVATE\n3.DISPLAY\n4.EXIT")
    print('---------------------------------------------\n')
    print("Enter your choice: ", end='')
    tchoice = int(input())
    if tchoice == 1:
       print("\nEnter process to crash : ", end='')
       crash = int(input())

       if pStatus[crash]:
          pStatus[crash] = 0
       else:
          print("Process", crash, "is already dead!", end='\n')
       condition = True
```

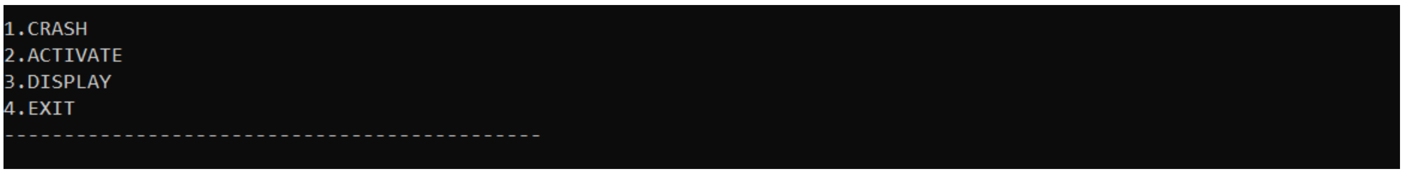
```
        while condition:
            print("Enter election generator id: ", end='')
            gid = int(input())
            if gid == coordinator:
                print("Please, enter a valid generator id!", end='\n')
            condition = (gid == coordinator)

        if crash == coordinator:
            subcoordinator = 1
            i = 0
            while i < (n+1):
                pid = (i + gid) % (n+1)
                if pid != 0:     # since our process starts from 1 (to n)
                    if pStatus[pid] and subcoordinator < pid:
                        subcoordinator = pid
                    print("Election message passed from", pid, ": #Msg", subcoordinator, end='\n')
                i += 1

            coordinator = subcoordinator
        display()

    elif tchoice == 2:
        print("Enter Process ID to be activated: ", end='')
        activate = int(input())
        if not pStatus[activate]:
            pStatus[activate] = 1
        else:
            print("Process", activate, "is already alive!", end='\n')
            break

        subcoordinator = activate
        i = 0
        while i < (n+1):
            pid = (i + activate) % (n+1)
```

```
                if pid != 0:    # since our process starts from 1 (to n)
                    if pStatus[pid] and subcoordinator < pid:
                        subcoordinator = pid
                    print("Election message passed from", pid,
                          ": #Msg", subcoordinator, end='\n')
                i += 1
            coordinator = subcoordinator
            display()
        elif tchoice == 3:
            display()
        condition = tchoice != 4


def choice():
    """ choice of options """
    while True:
        print('---------------------------------------------')
        print("1.BULLY ALGORITHM\n2.RING ALGORITHM\n3.DISPLAY\n4.EXIT")
        print('---------------------------------------------\n')
        fchoice = int(input("Enter your choice: "))


        if fchoice == 1:
            bully()
        elif fchoice == 2:
            ring()
        elif fchoice == 3:
            display()
        elif fchoice == 4:
            exit(0)
        else:
            print("Please, enter valid choice!")


def display():
    """ displays the processes, their status and the coordinator """
    global coordinator
    print('--------------------------------------------')
    print("PROCESS:", end='  ')
    for i in range(1, n+1):
        print(i, end='\t')
    print('\nALIVE:', end='    ')
    for i in range(1, n+1):
        print(pStatus[i], end='\t')
    print('\n--------------------------------------------')
    print('COORDINATOR IS', coordinator, end='\n')
    # print('--------------------------------------------')


if __name__ == '__main__':

    # take_input()

    n = int(input("Enter number of processes: "))
    for i in range(1, n+1):
        print("Enter Process ", i, " is alive or not(0/1): ")
        x = int(input())
        pStatus[i] = x
        if pStatus[i]:
            coordinator = i

    display()
    choice()
```

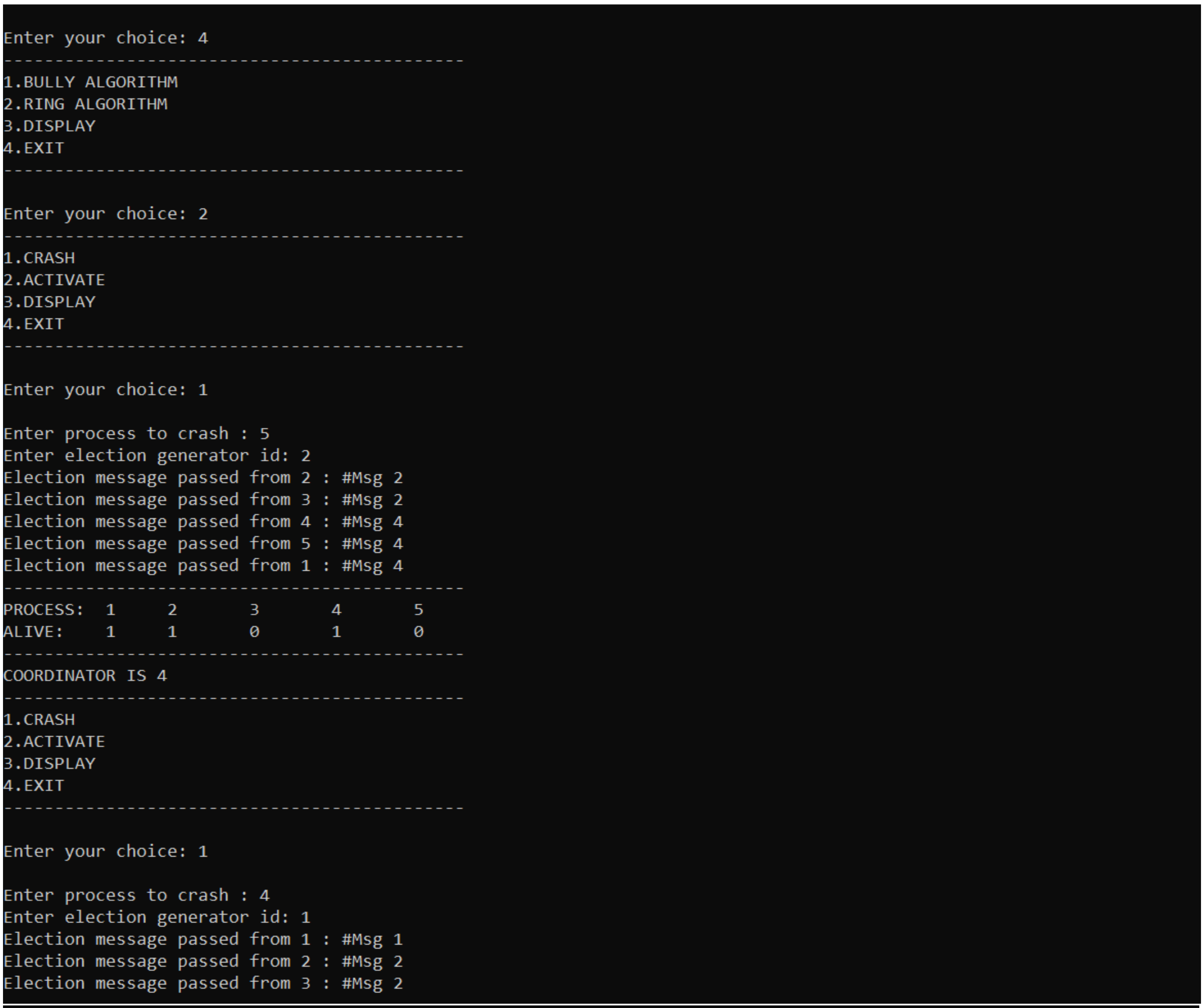
OUTPUT:



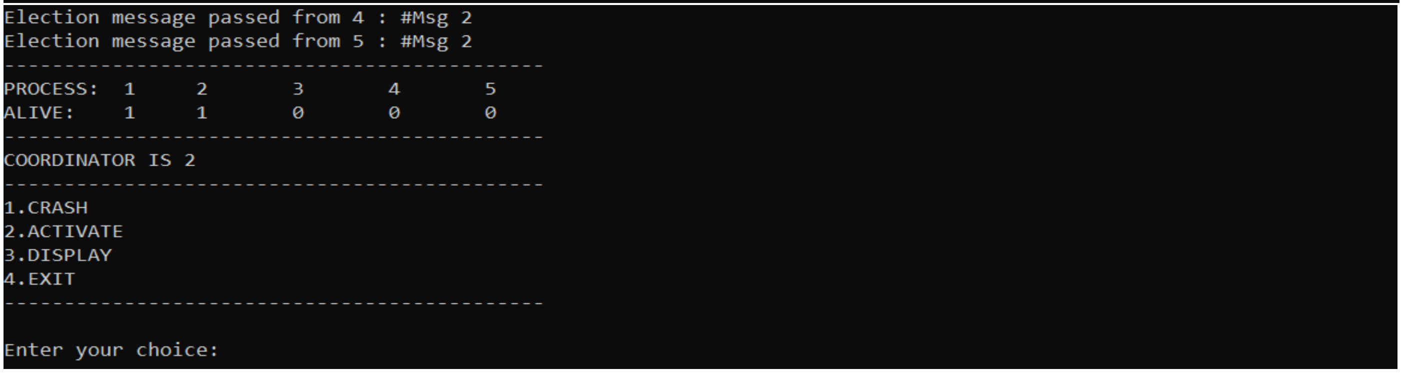
```
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Manish\Desktop\Distributed Systems Lab\Experiment 6 - Bully And Ring Election Algorithm>python bully_ring.py
Enter number of processes: 5
Enter Process  1  is alive or not(0/1):
1
Enter Process  2  is alive or not(0/1):
1
Enter Process  3  is alive or not(0/1):
0
Enter Process  4  is alive or not(0/1):
1
Enter Process  5  is alive or not(0/1):
1
---------------------------------------------
PROCESS:  1     2       3       4       5
ALIVE:    1     1       0       1       1
---------------------------------------------
COORDINATOR IS 5
---------------------------------------------
1.BULLY ALGORITHM
2.RING ALGORITHM
3.DISPLAY
4.EXIT
---------------------------------------------

Enter your choice: 1
---------------------------------------------
1.CRASH
2.ACTIVATE
3.DISPLAY
4.EXIT
---------------------------------------------

Enter your choice: 1
Enter process to crash: 5
Enter election generator id: 1
Message is  sent from 1  to 2
Response is sent from 2  to 1
Message is  sent from 1  to 3
Message is  sent from 1  to 4
Response is sent from 4  to 1
Message is  sent from 1  to 5
---------------------------------------------
PROCESS:  1     2       3       4       5
ALIVE:    1     1       0       1       0
---------------------------------------------
COORDINATOR IS 4
---------------------------------------------
```



```
1.CRASH
2.ACTIVATE
3.DISPLAY
4.EXIT
---------------------------------------------
```



```
Enter your choice: 4
---------------------------------------------
1.BULLY ALGORITHM
2.RING ALGORITHM
3.DISPLAY
4.EXIT
---------------------------------------------

Enter your choice: 2
---------------------------------------------
1.CRASH
2.ACTIVATE
3.DISPLAY
4.EXIT
---------------------------------------------

Enter your choice: 1

Enter process to crash : 5
Enter election generator id: 2
Election message passed from 2 : #Msg 2
Election message passed from 3 : #Msg 2
Election message passed from 4 : #Msg 4
Election message passed from 5 : #Msg 4
Election message passed from 1 : #Msg 4
---------------------------------------------
PROCESS:  1     2       3       4       5
ALIVE:    1     1       0       1       0
---------------------------------------------
COORDINATOR IS 4
---------------------------------------------
1.CRASH
2.ACTIVATE
3.DISPLAY
4.EXIT
---------------------------------------------

Enter your choice: 1

Enter process to crash : 4
Enter election generator id: 1
Election message passed from 1 : #Msg 1
Election message passed from 2 : #Msg 2
Election message passed from 3 : #Msg 2
```



```
Election message passed from 4 : #Msg 2
Election message passed from 5 : #Msg 2
---------------------------------------------
PROCESS:  1     2       3       4       5
ALIVE:    1     1       0       0       0
---------------------------------------------
COORDINATOR IS 2
---------------------------------------------
1.CRASH
2.ACTIVATE
3.DISPLAY
4.EXIT
---------------------------------------------

Enter your choice:
```

CONCLUSION:

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

ASSIGNMENT NO. 7

PROBLEM STATEMENT:

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

TOOLS / ENVIRONMENT:

Node.js, React.js, VS Code

THEORY:

Web Service:

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

A service can be treated as a web service if:

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

Types of Web Services:

There are two types of web services:

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

Web service architectures:

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

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

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

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

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

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

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

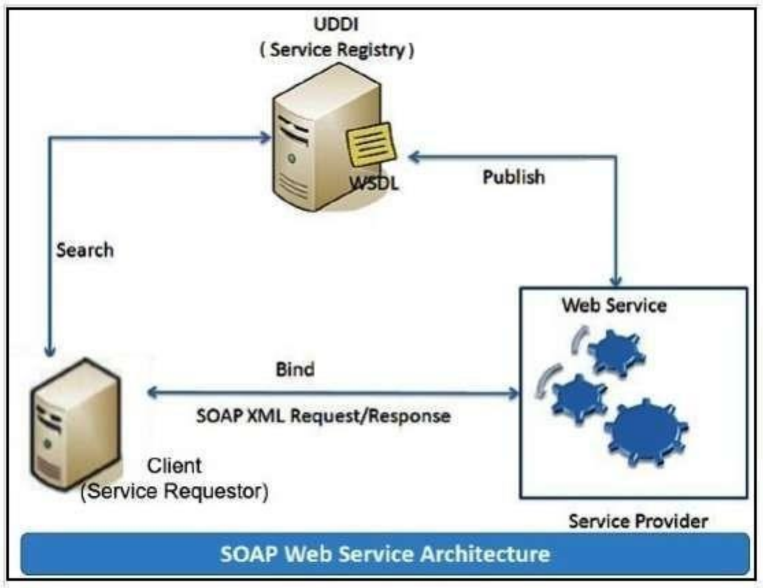
SOAP web services:

Simple Object Access Protocol (SOAP): is an XML-based protocol for accessing web services. It is a W3C recommendation for communication between two applications, and it is a platform- and language-independent technology in integrated distributed applications.

While XML and HTTP together make the basic platform for web services, the following are the key components of standard SOAP web services:

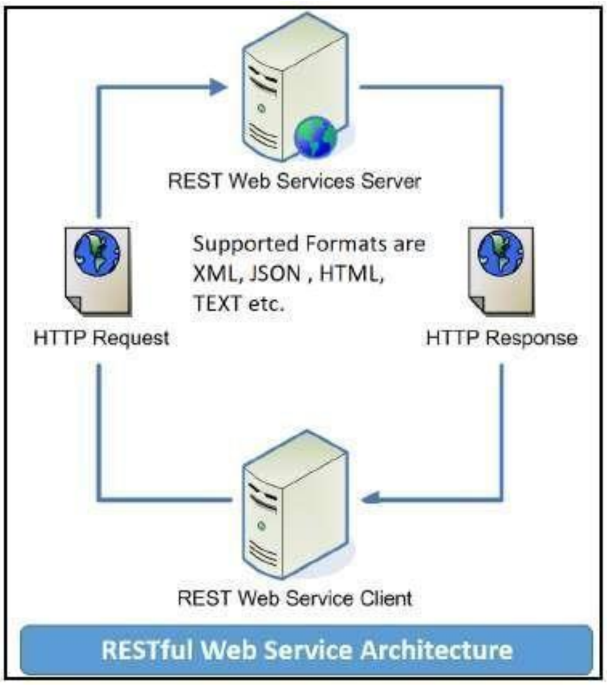
Universal Description, Discovery, and Integration (UDDI): UDDI is an XML based framework for describing, discovering, and integrating web services. It acts as a directory of web service interfaces described in the WSDL language.

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

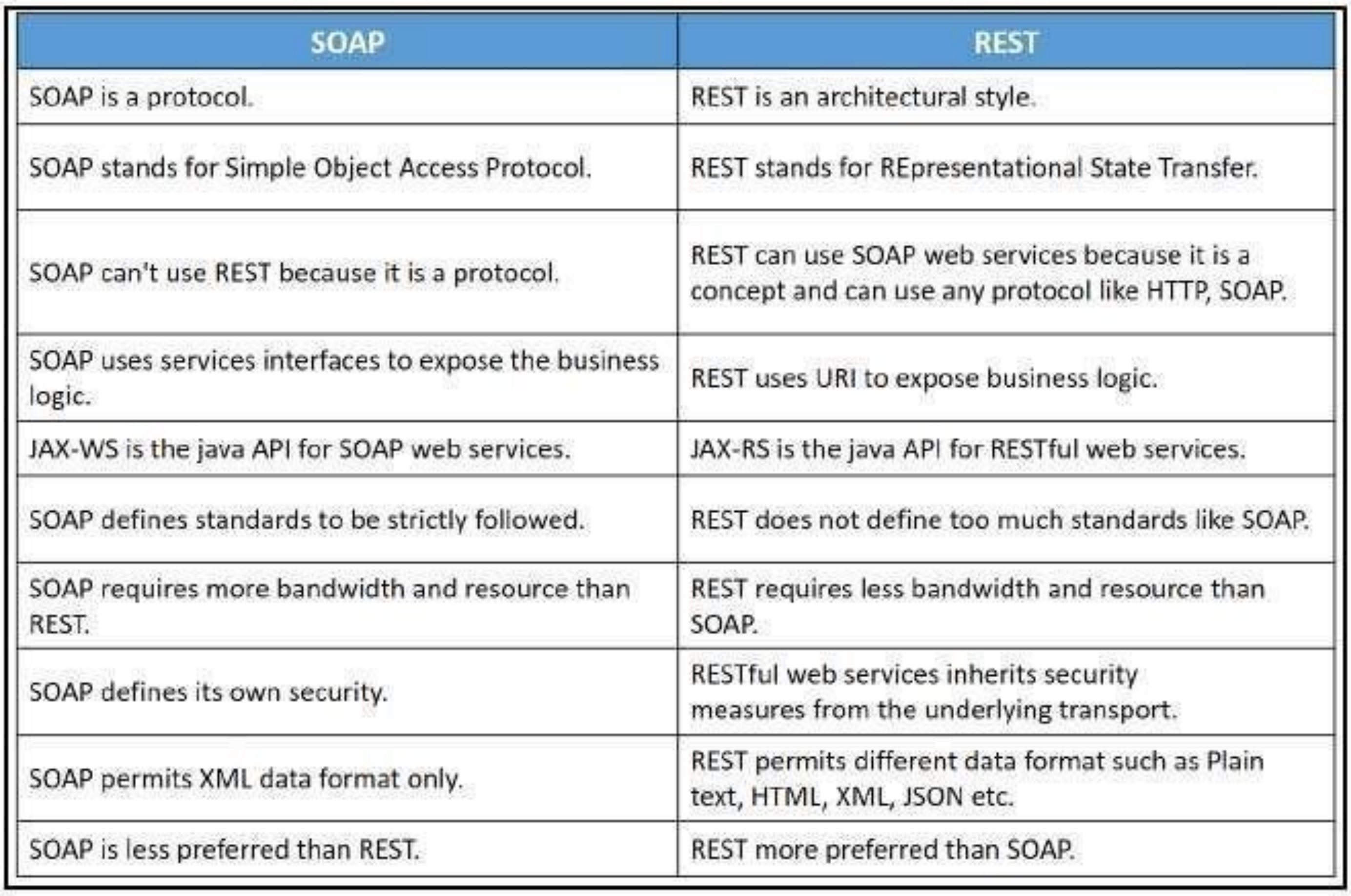


RESTful web services

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


REST Web Services Server
Supported Formats are XML, JSON , HTML, TEXT etc.
HTTP Request
HTTP Response
REST Web Service Client
RESTful Web Service Architecture

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



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

IMPLEMENTING THE SOLUTION:

- We will create a Web Service using Express.js which will be sending a json response to our distributed application written in React.js
- The React App will then take this json response from the above Web Server written in Express.js and then display the response to the user.

SOURCE CODE:

- ***Server.js***

```
const express = require("express");
const app = express()
const response = [
  {
    name: "Manish",
    email: "manish@gmail.com"
  },
  {
    name: "Zabhi",
    email: "zabhi@gmail.com"
  },
  {
    name: "adhwaith",
    email: "adhwaith@gmail.com"
  },
];
app.get("/users", (req, res) => {
  res.json(response);
})

app.listen(5000, () => {
  console.log("listening on port 5000");
})
```

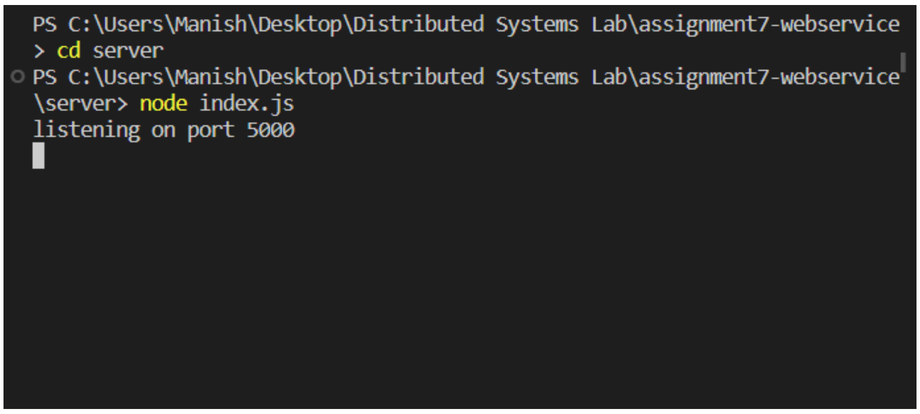
- ***App.js* (Written in React)**

```
import logo from './logo.svg';
import Card from "./components/Card/card"
import './App.css';
import { useEffect, useState } from 'react';
import Axios from 'axios';

function App() {
 const [data, setData] = useState([]);
 useEffect(() => {
  Axios.get('/users').then(res => { setData(res.data) }).catch(e => { console.log(e); })
 })
 return (
  <div className="App">
   <header className="App-header">
    <Card name="Manish" email="manishverma@gmail.com"></Card>
    {data.map((e) => <Card name={e.name} email={e.email}></Card>)}

   </header>
  </div>
 );
}

export default App;
```

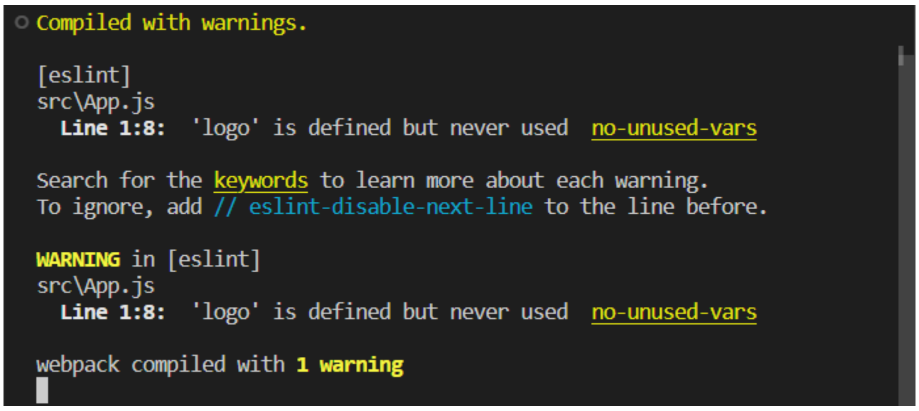
- ***Card.js* (Component to display data)**

```
import React from 'react'
import "./card.css"
const card = (props) => {
  return (
    <div className='Container'>
      <h3> Name :{props.name}</h3>
      <h5>Email :{props.email}</h5>
    </div>
  )
}

export default card
```

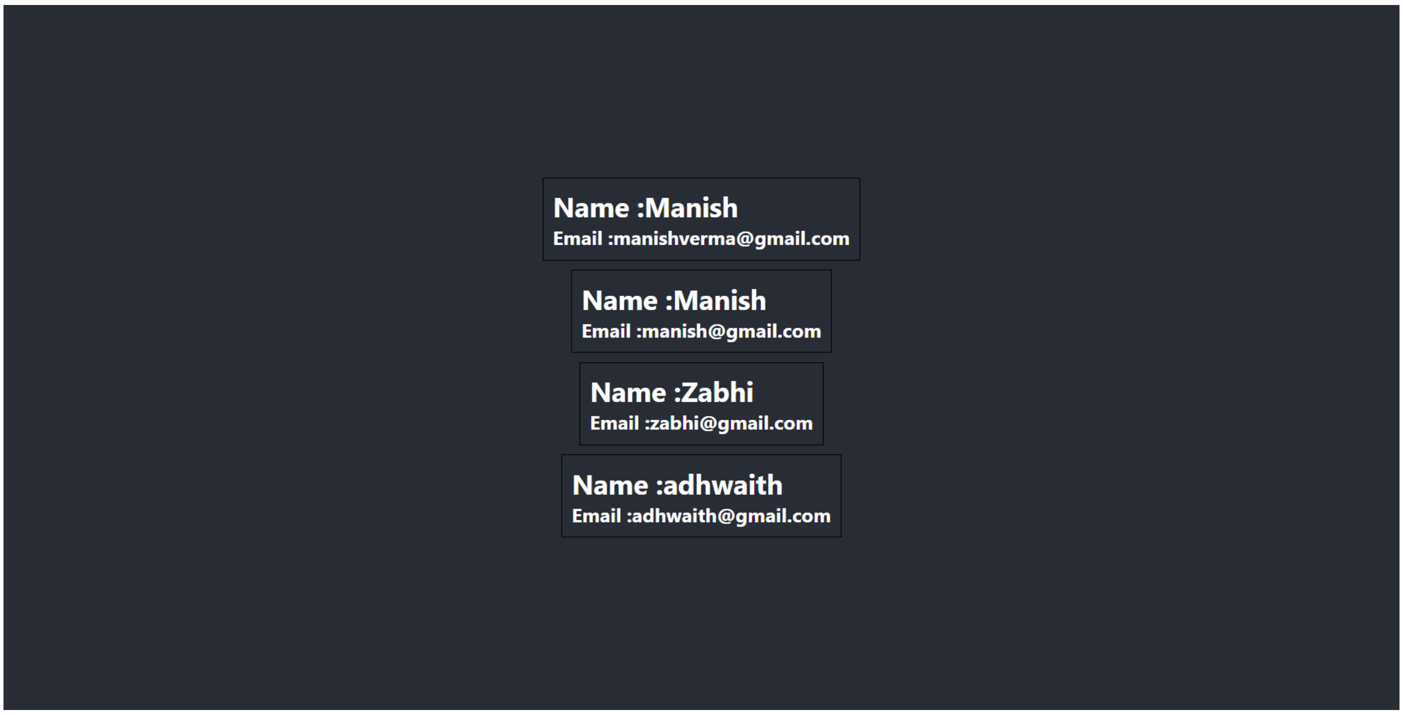
OUTPUT:



```
PS C:\Users\Manish\Desktop\Distributed Systems Lab\assignment7-webservice
> cd server
PS C:\Users\Manish\Desktop\Distributed Systems Lab\assignment7-webservice
\server> node index.js
listening on port 5000
```



```
Compiled with warnings.

[eslint]
src\App.js
  Line 1:8:  'logo' is defined but never used  no-unused-vars

Search for the keywords to learn more about each warning.
To ignore, add // eslint-disable-next-line to the line before.

WARNING in [eslint]
src\App.js
  Line 1:8:  'logo' is defined but never used  no-unused-vars

webpack compiled with 1 warning
```



Name :Manish
Email :manishverma@gmail.com

Name :Manish
Email :manish@gmail.com

Name :Zabhi
Email :zabhi@gmail.com

Name :adhwaith
Email :adhwaith@gmail.com

CONCLUSION:

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