

Sinhgad Technical Education Society's

NBN SINHGAD TECHNICAL INSTITUTES CAMPUS,
Ambegaon (BK) 411041

NAAC Accredited with 'A' Grade



CERTIFICATE

This is to certify that Mr. /Ms.
of class BE IT Div_____Roll No. _____Examination Seat No./PRN No.

has completed all the practical work in the: Lab Practice V [414454] satisfactorily, as
prescribed by Savitribai Phule Pune University, Pune in the academic year 2022-23 (Sem

I)

Place:

Date:

Course In-charge

Head of Department

Principal

Index

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

Assignment No. 1

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

Objectives: To develop a multi-threaded client/server process communication using Java RMI.

Infrastructure:

Software Used: Java, Eclipse IDE, JDK

Theory:

Remote method invocation (RMI) allows a java object to invoke method on an object running on another machine. RMI provide remote communication between java program. RMI is used for building distributed application.

RMI applications often comprise two separate programs, a server and a client.

1. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
2. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

Distributed object applications need to do the following:

- Locate remote objects.
- Communicate with remote objects.
- Load class definitions for objects that are passed around.

The RMI provides remote communication between the applications using two objects stub and skeleton. A remote object is an object whose method can be invoked from another JVM.

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

Skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and 3. It writes and transmits (marshals) the result to the caller.

Remote Interfaces, Objects, and Methods

Like any other Java application, a distributed application built by using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called remote objects.

An object becomes remote by implementing a remote interface, which has the following characteristics:

- A remote interface extends the interface `java.rmi.Remote`.
- Each method of the interface declares `java.rmi.RemoteException` in its throws clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine. Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote stub for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, only those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

Creating Distributed Applications by Using RMI

Using RMI to develop a distributed application involves these general steps:

1. Designing and implementing the components of your distributed application.
2. Compiling sources.
3. Making classes network accessible.
4. Starting the application.

1. Designing and Implementing the Application Components

First determine your application architecture, including the components of local objects and components accessible remotely.

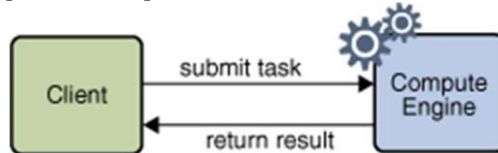
- Defining the remote interfaces. A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.
- Implementing the remote objects. Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.
- Implementing the clients. Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

1.1 Writing an RMI Server

The compute engine server accepts tasks from clients, runs the tasks, and returns any results. The server code consists of an interface and a class. The interface defines the methods that can be invoked from the client. Essentially, the interface defines the client's view of the remote object. The class provides the implementation.

1.1.1 Designing a Remote Interface:

At the core of the compute engine is a protocol that enables tasks to be submitted to the compute engine, the compute engine to run those tasks, and the results of those tasks to be returned to the client. This protocol is expressed in the interfaces that are supported by the compute engine.



Each interface contains a single method. The compute engine's remote interface, `Compute`, enables tasks to be submitted to the engine. The client interface, `Task`, defines how the compute engine executes a submitted task.

The `compute.Compute` interface defines the remotely accessible part, the compute engine itself.

```

1. package compute;
2.
3.     import java.rmi.Remote;
4.     import java.rmi.RemoteException;
5.
6.     public interface Compute extends Remote {
7.         <T> T executeTask(Task<T> t) throws RemoteException;
8.     }
9.
  
```

By extending the interface `java.rmi.Remote`, the `Compute` interface identifies itself as an interface whose methods can be invoked from another Java virtual machine. Any object that implements this interface can be a remote object.

As a member of a remote interface, the `executeTask` method is a remote method, it is capable of throwing `java.rmi.RemoteException`. It is a checked exception, so any code invoking a remote method needs to handle this exception by either catching it or declaring it in its `throws` clause.

The second interface needed for the compute engine is the `Task` interface, which is the type of the parameter to the `executeTask` method in the `Compute` interface. The `compute.Task` interface defines the interface between the compute engine and the work that it needs to do, providing the way to start the work.

```

1. package compute;
2.
3.     public interface Task<T> {
4.         T execute();
5.     }
6.
  
```

The `Compute` interface's `executeTask` method, in turn, returns the result of the execution of the `Task` instance passed to it. Thus, the `executeTask` method has its own type parameter, `T`, that associates its own return type with the result type of the passed `Task` instance.

RMI uses the Java object serialization mechanism to transport objects by value between Java virtual machines. For an object to be considered serializable, its class must implement the `java.io.Serializable` marker interface. Therefore, classes that implement the `Task` interface must also implement `Serializable`, as must the classes of objects used for task results.

1.1.2 Implementing a Remote Interface

In general, a class that implements a remote interface should at least do the following:

- Declare the remote interfaces being implemented
- Define the constructor for each remote object
- Provide an implementation for each remote method in the remote interfaces

An RMI server program needs to create the initial remote objects and export them to the RMI runtime, which makes them available to receive incoming remote invocations. This setup procedure can be either encapsulated in a method of the remote object implementation class itself or included in another class entirely. The setup procedure should do the following:

- Create and install a security manager
- Create and export one or more remote objects
- Register at least one remote object with the RMI registry (or with another naming service, such as a service accessible through the Java Naming and Directory Interface) for bootstrapping purposes

```

1. package engine; 2.
3.     import java.rmi.RemoteException;
4.     import java.rmi.registry.LocateRegistry;
5.     import java.rmi.registry.Registry;
6.     import java.rmi.server.UnicastRemoteObject;
7.     import compute.Compute; 8. import compute.Task; 9.
10. public class ComputeEngine implements Compute { 11.
12.     public ComputeEngine() {
13.         super(); 14.     } 15.
16.     public <T> T executeTask(Task<T> t) {
17.         return t.execute(); 18.     } 19.
20.     public static void main(String[] args) {
21.         if (System.getSecurityManager() == null) {
22.             System.setSecurityManager(new SecurityManager());
23.         }
24.         try {
25.             String name = "Compute";
26.             Compute engine = new ComputeEngine();
27.             Compute stub =
28.             (Compute) UnicastRemoteObject.exportObject(engine, 0);
29.             Registry registry = LocateRegistry.getRegistry();
30.             registry.rebind(name, stub);
31.             System.out.println("ComputeEngine bound");
32.         } catch (Exception e) {
33.             System.err.println("ComputeEngine exception:");
34.             e.printStackTrace();
35.         }
36.     }
37. } 38.

```

1.2 Creating a Client Program

A client needs to call the compute engine, but it also has to define the task to be performed by the compute engine.

The first class, ComputePi, looks up and invokes a Compute object.

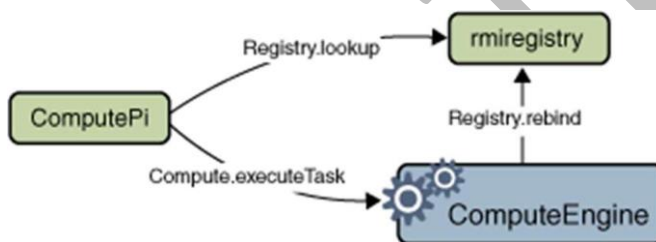
```

1. package client; 2.
3.     import java.rmi.registry.LocateRegistry;
4.     import java.rmi.registry.Registry;
5.     import java.math.BigDecimal; 6. import compute.Compute; 7.
8.     public class ComputePi {
9.         public static void main(String args[]) {
10.            if (System.getSecurityManager() == null) {
11.                System.setSecurityManager(new SecurityManager());
12.            }
13.            try {
14.                String name = "Compute";
15.                Registry registry = LocateRegistry.getRegistry(args[0]);
16.                Compute comp = (Compute) registry.lookup(name);
17.                Pi task = new Pi(Integer.parseInt(args[1]));
18.                BigDecimal pi = comp.executeTask(task);

19.                System.out.println(pi);
20.            } catch (Exception e) {
21.                System.err.println("ComputePi exception:");
22.                e.printStackTrace();
23.            }
24.        }
25.    } 26.

```

Next, the client creates a new Pi object, passing to the Pi constructor the value of the second command-line argument, args[1], parsed as an integer. This argument indicates the number of decimal places to use in the calculation. Finally, the client invokes the executeTask method of the Compute remote object. The object passed into the executeTask invocation returns an object of type BigDecimal, which the program stores in the variable result. Finally, the program prints the result. The following figure depicts the flow of messages among the ComputePi client, the rmiregistry, and the ComputeEngine.



2. Compiling the Example Programs

In a real-world scenario in which a service such as the compute engine is deployed, a developer would likely create a Java Archive (JAR) file that contains the Compute and Task interfaces for server classes to implement and client programs to use. Next, a developer, perhaps the same developer of the interface JAR file, would write an implementation of the Compute interface and deploy that service on a machine available to clients.

- 2.1 Building a JAR File of Interface Classes: First, you need to compile the interface source files in the compute package and then build a JAR file that contains their class files.
- 2.2 Building the Server Classes: The engine package contains only one server-side implementation class, ComputeEngine, the implementation of the remote interface Compute.
- 2.3 Building the Client Classes: The client package contains two classes, ComputePi, the main client program, and Pi, the client's implementation of the Task interface.

3. Running the Program

Before starting the compute engine, you need to start the RMI registry. The RMI registry is a simple server-side bootstrap naming facility that enables remote clients to obtain a reference to an initial remote object. It can be started with the rmiregistry command. Before you

execute `rmiregistry`, you must make sure that the shell or window in which you will run `rmiregistry` either has no `CLASSPATH` environment variable set or has a `CLASSPATH` environment variable that does not include the path to any classes that you want downloaded to clients of your remote objects.

3.1 To start the registry on the server, execute the `rmiregistry` command. This command produces no output and is typically run in the background.

1. start `rmiregistry`

By default, the registry runs on port 1099.

Once the registry is started, you can start the server. You need to make sure that both the `compute.jar` file and the remote object implementation class are in your class path. When you start the compute engine, you need to specify, using the `java.rmi.server.codebase` property, where the server's classes are network accessible.

Eg. `java -cp c:\home\ann\src;c:\home\ann\public_html\classes\compute.jar
-Djava.rmi.server.codebase=file:/c:/home/ann/public_html/classes/compute.jar
Djava.rmi.server.hostname=mycomputer.example.com -Djava.security.policy=server.policy engine.ComputeEngine`

3.2 Starting the Client

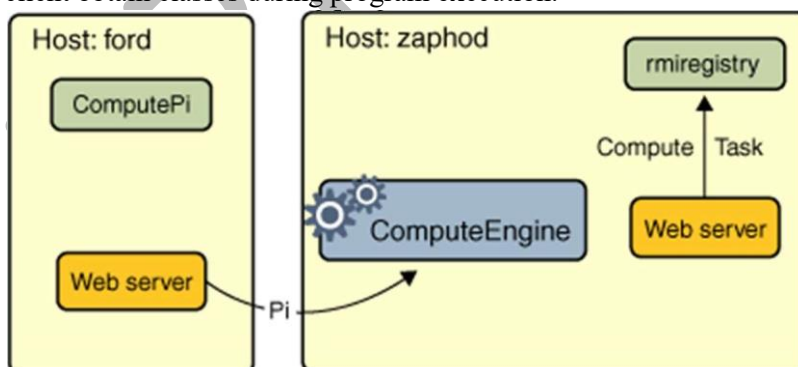
Once the registry and the compute engine are running, you can start the client, specifying the following:

- The location where the client serves its classes (the `Pi` class) by using the `java.rmi.server.codebase` property
- The `java.security.policy` property, which is used to specify the security policy file that contains the permissions you intend to grant to various pieces of code
- As command-line arguments, the host name of the server (so that the client knows where to locate the Compute remote object) and the number of decimal places to use in the calculation π

Start the client on another host (a host named `mysecondcomputer`, for example) as follows:

`java -cp c:\home\jones\src;c:\home\jones\public_html\classes\compute.jar -
Djava.rmi.server.codebase=file:/c:/home/jones/public_html/classes/
-Djava.security.policy=client.policy
client.ComputePi mycomputer.example.com 45`

The following figure illustrates where the `rmiregistry`, the `ComputeEngine` server, and the `ComputePi` client obtain classes during program execution.



When the `ComputeEngine` server binds its remote object reference in the registry, the registry downloads the `Compute` and `Task` interfaces on which the stub class depends. These classes are downloaded from either the `ComputeEngine` server's web server or file system, depending on the type of codebase URL used when starting the server.

Because the ComputePi client has both the Compute and the Task interfaces available in its class path, it loads their definitions from its class path, not from the server's codebase.

Finally, the Pi class is loaded into the ComputeEngine server's Java virtual machine when the Pi object is passed in the executeTask remote call to the ComputeEngine object. The Pi class is loaded by the server from either the client's web server or file system, depending on the type of codebase URL used when starting the client.

Conclusion:

We implemented a multi-thread client/server process communication using RMI.

NBNSTIC-IT-DS

Assignment No. 1

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

Objectives: To develop a multi-threaded client/server process communication using Java RMI.

Infrastructure:

Software Used: Java, Eclipse IDE, JDK

Theory:

Remote method invocation (RMI) allows a java object to invoke method on an object running on another machine. RMI provide remote communication between java program. RMI is used for building distributed application.

RMI applications often comprise two separate programs, a server and a client.

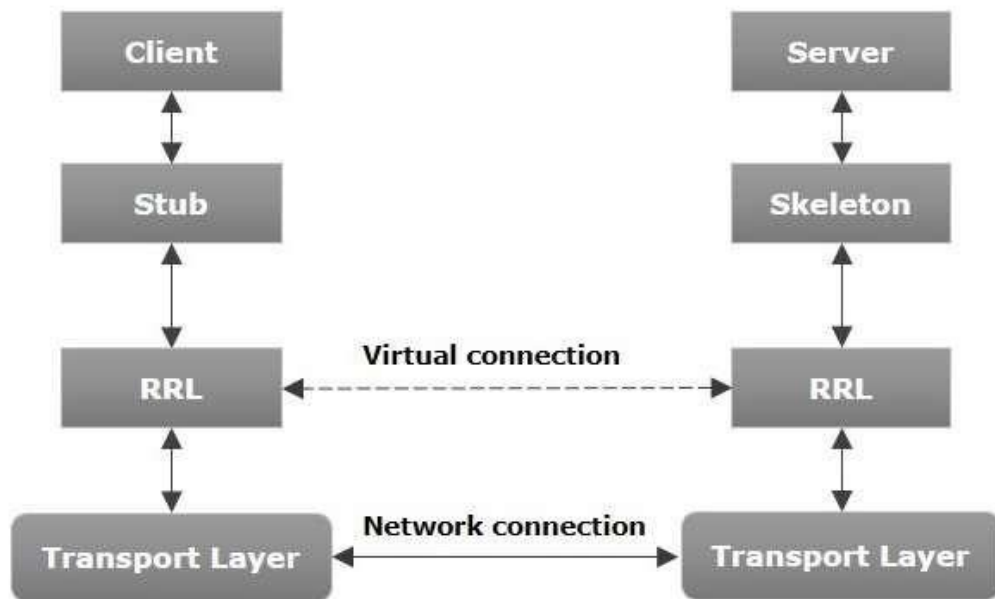
1. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
2. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

Distributed object applications need to do the following:

- Locate remote objects.
- Communicate with remote objects.
- Load class definitions for objects that are passed around.

Architecture of an RMI Application



- **Transport Layer** – This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** – A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** – This is the object which resides on the server side. stub communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer)** – It is the layer which manages the references made by the client to the remote object.

How RMI provide remote Communication?

The RMI provides remote communication between the applications using two objects stub and skeleton. A remote object is an object whose method can be invoked from another JVM.

1. stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and 5. It finally, returns the value to the caller.

2. Skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and 3. It writes and transmits (marshals) the result to the caller.

Remote Interfaces, Objects, and Methods

The distributed application built using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called remote objects.

An object becomes remote by implementing a remote interface, which has the following characteristics:

- A remote interface extends the interface `java.rmi.Remote`.
- Each method of the interface declares `java.rmi.RemoteException` in its throws clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine. Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote stub for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, only those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

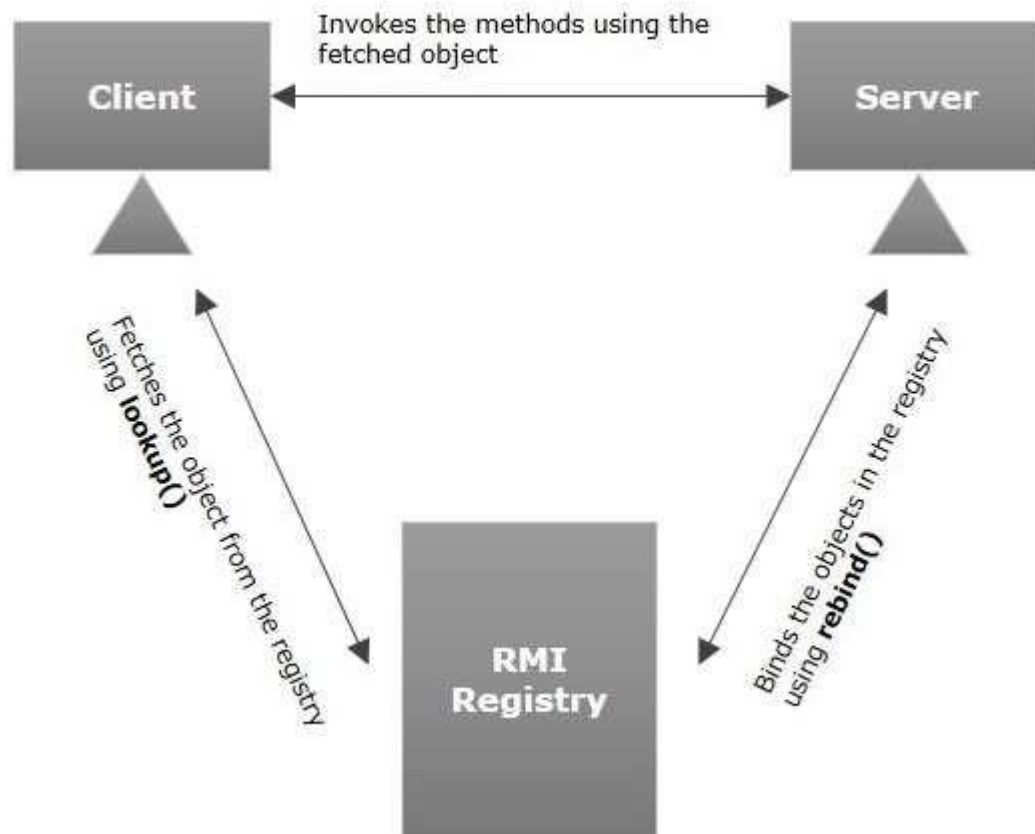
Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as marshalling.

At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as unmarshalling. RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the `RMIRegistry` (using `bind()` or `reBind()` methods). These are registered using a unique name known as bind name.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using `lookup()` method).



Creating Distributed Applications by Using RMI

Using RMI to develop a distributed application involves these general steps:

1. Designing and implementing the components of your distributed application.
2. Compiling sources.
3. Making classes network accessible.
4. Starting the application.

Goals of RMI

Following are the goals of RMI –

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

Conclusion:

We implemented a multi-thread client/server process communication using RMI.

Instruction:

1. Develop the following java files.

Interface

```

1. import java.rmi.Remote;
2. import java.rmi.RemoteException;
3. public interface Hello extends Remote {
4.     default void message() throws RemoteException {}
5. } 6.

```

Interface Implementation

```

1. public class ImplExample implements Hello {
2.     public void message() {
3.         System.out.println("This is at server");
4.     }
5. } 6.

```

Server

```

1. import java.rmi.registry.Registry;
2. import java.rmi.registry.LocateRegistry;
3. import java.rmi.RemoteException;
4. import java.rmi.server.UnicastRemoteObject;
5. public class Server extends ImplExample {
6.     public Server() {}
7.     public static void main(String args[]) {
8.         try {
9.             // Instantiating the implementation class
10.            ImplExample obj = new ImplExample();
11.            // Exporting the object of implementation class
12.            // (here we are exporting the remote object to the stub)
13.            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
14.            // Binding the remote object (stub) in the registry
15.            Registry registry = LocateRegistry.getRegistry();
16.            registry.bind("Hello", stub);
17.            System.err.println("Server ready");
18.        } catch (Exception e) {
19.            System.err.println("Server exception: " + e.toString());
20.            e.printStackTrace();
21.        }
22.    }
23. } 24.

```

Client

```

1. import java.rmi.registry.LocateRegistry;
2. import java.rmi.registry.Registry;
3. public class Client {
4.     private Client() {}
5.     public static void main(String[] args) {
6.         try {
7.             // Getting the registry
8.            Registry registry = LocateRegistry.getRegistry(null);
9.            // Looking up the registry for the remote object
10.           Hello stub = (Hello) registry.lookup("Hello");
11.           // Calling the remote method using the obtained object
12.           stub.message();
13.           // System.out.println("Remote method invoked");
14.        } catch (Exception e) {
15.            System.err.println("Client exception: " + e.toString());
16.            e.printStackTrace();
17.        }
18.    }
19. } 20.

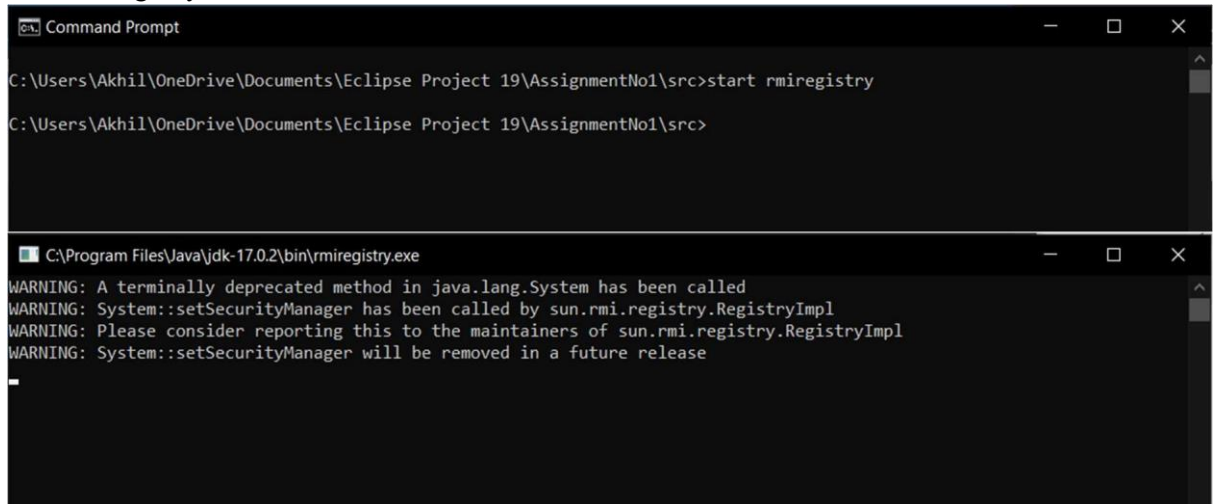
```

2. Open command-prompt and compile the java file
javac *.java
3. Start the rmiregistry start rmiregistry

4. Then start the “Server” class java Server
5. Then open another command-prompt and start the client java Client

Output

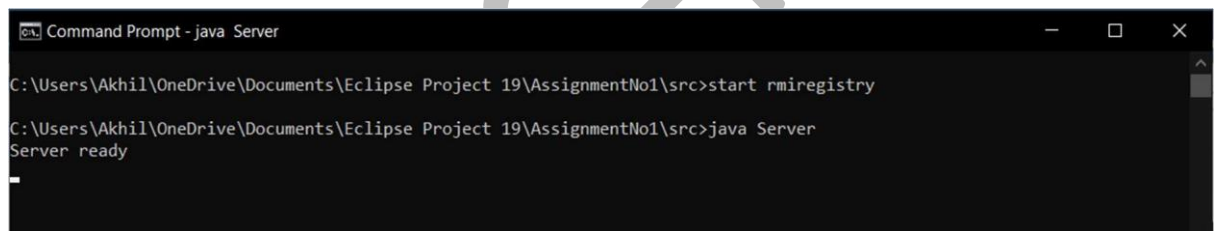
1. Start rmiregistry



```
Command Prompt
C:\Users\Akhil\OneDrive\Documents\Eclipse Project 19\AssignmentNo1\src>start rmiregistry
C:\Users\Akhil\OneDrive\Documents\Eclipse Project 19\AssignmentNo1\src>

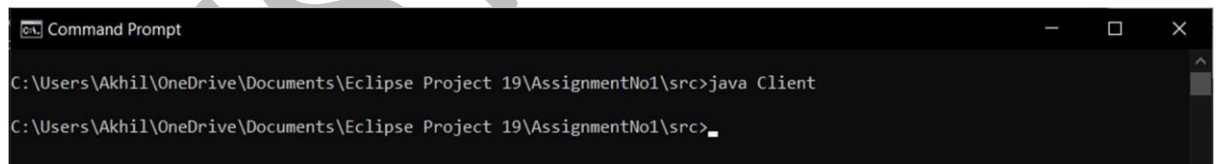
C:\Program Files\Java\jdk-17.0.2\bin\rmiregistry.exe
WARNING: A terminally deprecated method in java.lang.System has been called
WARNING: System::setSecurityManager has been called by sun.rmi.registry.RegistryImpl
WARNING: Please consider reporting this to the maintainers of sun.rmi.registry.RegistryImpl
WARNING: System::setSecurityManager will be removed in a future release
```

2. Start the server



```
Command Prompt - java Server
C:\Users\Akhil\OneDrive\Documents\Eclipse Project 19\AssignmentNo1\src>start rmiregistry
C:\Users\Akhil\OneDrive\Documents\Eclipse Project 19\AssignmentNo1\src>java Server
Server ready
```

3. Start client in different client



```
Command Prompt
C:\Users\Akhil\OneDrive\Documents\Eclipse Project 19\AssignmentNo1\src>java Client
C:\Users\Akhil\OneDrive\Documents\Eclipse Project 19\AssignmentNo1\src>
```

4. Server output



```
Command Prompt - java Server
C:\Users\Akhil\OneDrive\Documents\Eclipse Project 19\AssignmentNo1\src>start rmiregistry
C:\Users\Akhil\OneDrive\Documents\Eclipse Project 19\AssignmentNo1\src>java Server
Server ready
This is at server
```

Assignment No. 2

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

Objective: Demonstration of object brokering using CORBA in Java.

Infrastructure:

Software used: Java, JDK 1.8, IDE like Eclipse(optional)

Theory:

Common Object Request Broker Architecture (CORBA):

CORBA is an acronym for Common Object Request Broker Architecture, 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.

Why CORBA is used?

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.

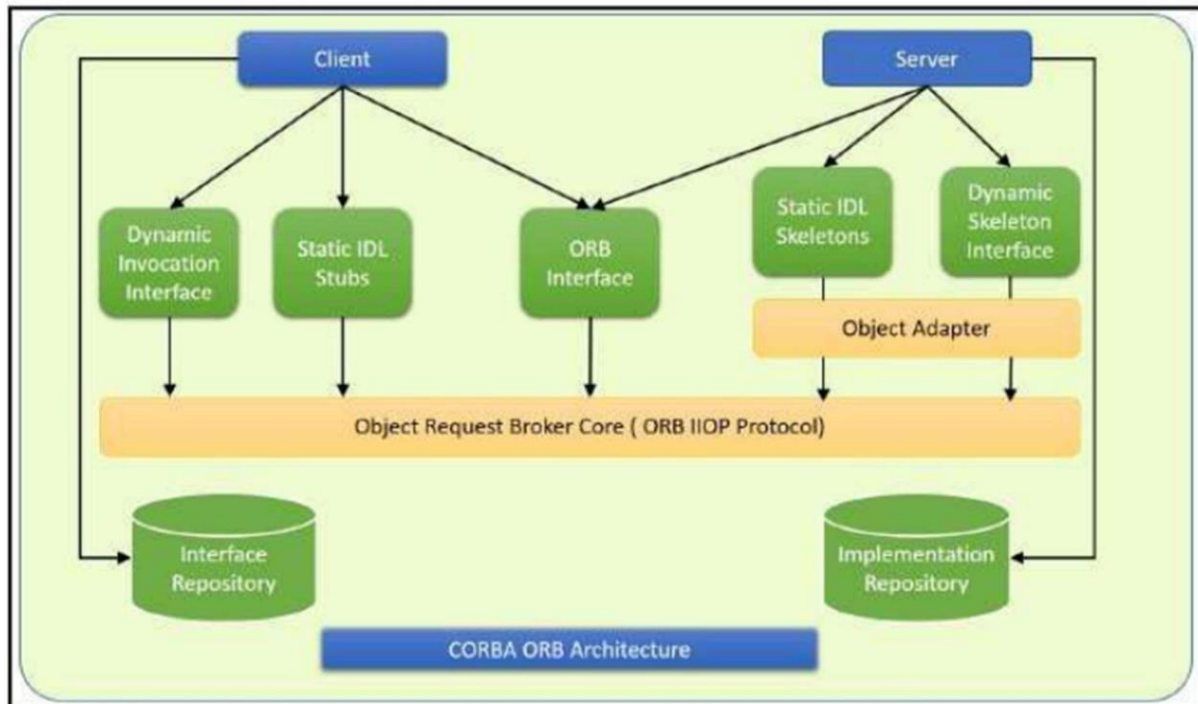
An application developed based on CORBA standards with standard Internet Inter-ORB Protocol (IIOP), irrespective of the vendor that develops it, should be able to smoothly integrate and operate with another application developed based on CORBA standards through the same or different vendor.

Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example, applications similar to "HR&Benefits" maintain an object model with details of the organization, employees benefits details. 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 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. The interface has to be defined clearly for each object. The systems encapsulate the actual implementation along with their respective data handling and processing, and only the methods are available to the rest of the world through the interface. Hence, the clients are forced to develop their invocation logic for the IDL interface exposed by the application they want to connect to with the method parameters (input and output) advised by the interface operation.



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

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

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

Java Support for CORBA

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

CORBA standards provide the proven, interoperable infrastructure to the Java platform. IIOP (Internet Inter-ORB Protocol) manages the communication between the object components that power the system. The Java platform provides a portable object infrastructure that works on every major operating system. CORBA provides the network transparency; Java provides the implementation transparency. An Object Request Broker (ORB) is part of the Java Platform. The ORB is a runtime component that can be used for distributed computing using IIOP communication. Java IDL is a Java API for interoperability and integration with CORBA. Java IDL included both a Java-based ORB, which supported IIOP, and the IDL-to-Java compiler, for generating client-side stubs and server-side code skeletons. J2SE v.1.4 includes an Object Request Broker Daemon (ORBD), which is used to enable clients to transparently locate and invoke persistent objects on servers in the CORBA environment.

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

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

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

The IDL Programming Model:

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

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

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

Portable Object Adapter (POA): An object adapter is the mechanism that connects a request using an object reference with the proper code to service that request. The Portable Object

Adapter, or POA, is a particular type of object adapter that is defined by the CORBA specification. The POA is designed to meet the following goals:

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

Designing the solution:

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

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

1. Creating CORBA Objects using Java IDL:

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

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

Interfaces in IDL are declared much like interfaces in Java.

1.2. Modules

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

```
1. // IDL
2. module jen {
3.   module corba {
4.     interface NeatExample ...
5.   };
6. }; 7.
```

1.3. Interfaces

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

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

2.

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

1.4 Data members and methods

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

1. `readonly attribute string myString;`

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

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

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

1.5 A complete IDL example

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

```

1.  module OS {
2.  module services {
3.  interface Server {
4.  readonly attribute string serverName;
5.  boolean init(in string sName);
6.  };
7.  interface Printable {
8.  boolean print(in string header);
9.  };
10. interface PrintServer : Server {
11. boolean printThis(in Printable p);
12. }; 13. };
14. }; 15.
```

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

2. Turning IDL Into Java

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

IDL interface:

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

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

- A holder class whose name is the name of the IDL interface with "Holder" appended to it

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

The idltoj tool generate 2 other classes:

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

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

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

3. Writing the Implementation

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

Implementing the solution:

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

1. Defining the Interface (Hello.idl)

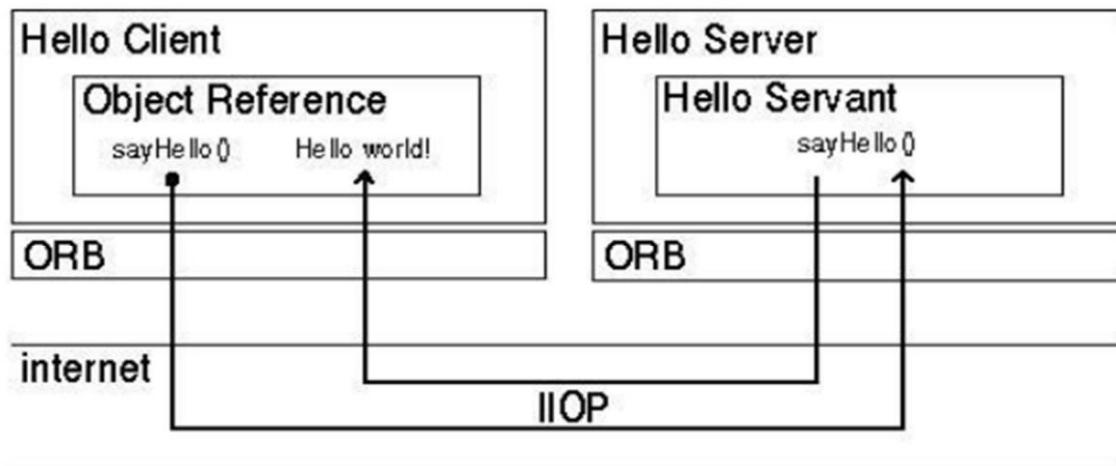
The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL). To complete the application, you simply provide the server (HelloServer.java) and client (HelloClient.java) implementations.

2. Implementing the Server (HelloServer.java)

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

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

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



3. Implementing the Client Application (HelloClient.java)

The example application client that follows:

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

Building and executing the solution:

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

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

1. Change to the directory that contains the file Hello.idl.
2. Run the IDL-to-Java compiler, idlj, on the IDL file to create stubs and skeletons.

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

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

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

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

o HelloPOA.java:

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

o _HelloStub.java:

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

o Hello.java:

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

HelloHelper.java

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

o HelloHolder.java

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

o HelloOperations.java

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

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

1. javac *.java HelloApp/*.java

4. Start orbd.

To start orbd from a UNIX command shell, enter:

```
orbd -ORBInitialPort 1050&
```


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

5. Start the HelloServer:

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

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

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

6. Run the client application:

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

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

HelloServer exiting... Note: After completion kill the name server (orbd)

Conclusion:

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

Assignment No. 3

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

Objective: Demonstration of object brokering using CORBA in Java.

Infrastructure:

Software used: Java, JDK 1.8, IDE like Eclipse(optional)

Theory:

Common Object Request Broker Architecture (CORBA):

CORBA is an acronym for Common Object Request Broker Architecture, 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.

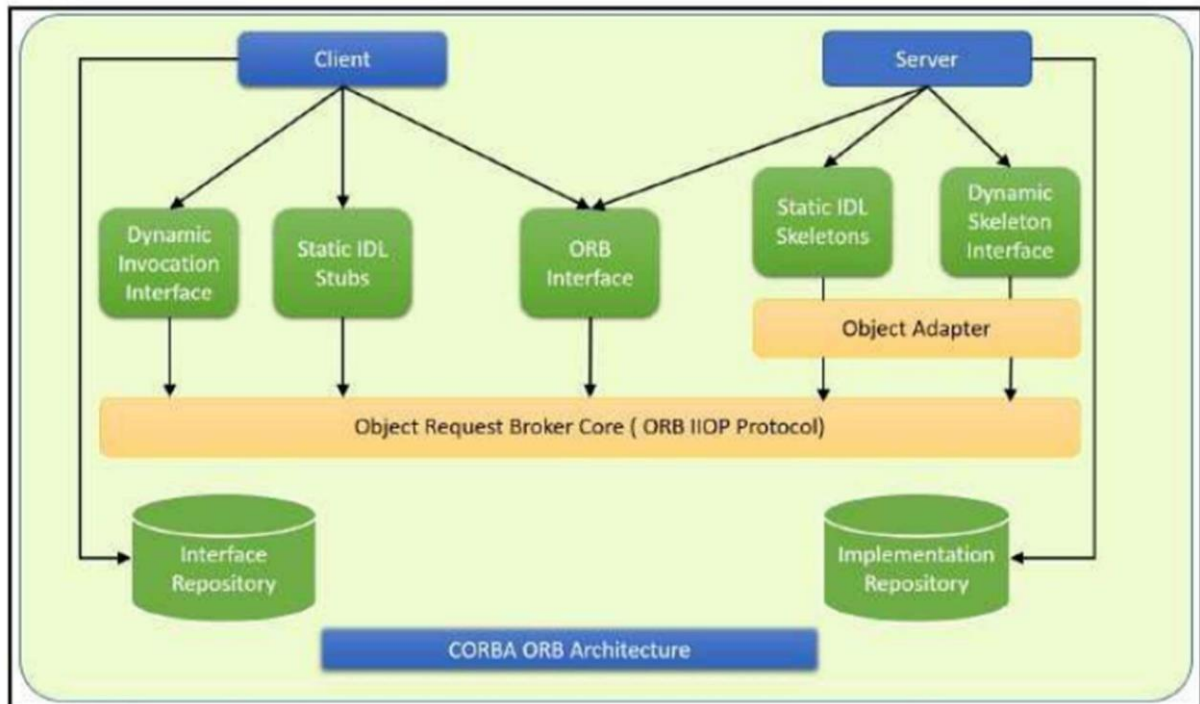
Why CORBA is used?

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.

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 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. The interface has to be defined clearly for each object. The systems encapsulate the actual implementation along with their respective data handling and processing, and only the methods are available to the rest of the world through the interface. Hence, the clients are forced to develop their invocation logic for the IDL interface exposed by the application they want to connect to with the method parameters (input and output) advised by the interface operation.



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

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

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

Java Support for CORBA

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.

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.

The IDL Programming Model:

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

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

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

1. Creating CORBA Objects using Java IDL:

1.1. First one has to define CORBA-enabled interface and its implementation, which involves:

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

Interfaces in IDL are declared much like interfaces in Java.

1.2. Modules

Modules are declared in IDL using the module keyword. 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.

```

1. // IDL
2. module jen {
3.   module corba {
4.     interface NeatExample ...
5.   };
6. }; 7.

```

1.3. Interfaces

It involves declaration of interface which includes an interface header and an interface body. For example:

```

1. interface PrintServer : Server { ...
2.

```

1.4 Data members and methods

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

```

1. readonly attribute string myString;

```

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

```

1. string parseString(in string buffer);

```

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

1.5 A complete IDL example

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

```

1. module OS {
2.   module services {
3.     interface Server {
4.       readonly attribute string serverName;
5.       boolean init(in string sName);
6.     };
7.     interface Printable {
8.       boolean print(in string header);
9.     };
10.    interface PrintServer : Server {
11.      boolean printThis(in Printable p);
12.    }; 13. };
14. }; 15.

```

2. Turning IDL Into Java

Now for implementing the remote interfaces in Java we use 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
- A holder class whose name is the name of the IDL interface with "Holder" appended to it

The idltoj tool generate 2 other classes:

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

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

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

3. Writing the Implementation

The IDL interface is written and generated the Java interface and support classes for it, including the client stub and the server skeleton.

Implementing the solution:

Now create a directory named `hello/` where you develop sample applications and create the files in this directory.

1. Defining the Interface (Hello.idl)

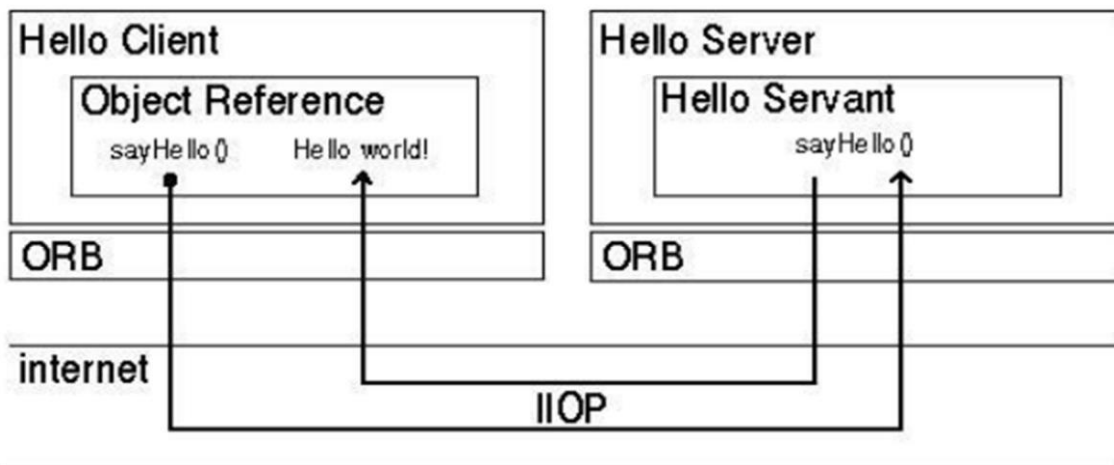
The first step to creating a CORBA application is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL). To complete the application, you simply provide the server (`HelloServer.java`) and client (`HelloClient.java`) implementations.

2. Implementing the Server (HelloServer.java)

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

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

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



3. Implementing the Client Application (HelloClient.java)

The example application client that follows:

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

Building and executing the solution:

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

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

1. Change to the directory that contains the file Hello.idl.
2. Run the IDL-to-Java compiler, idlj, on the IDL file to create stubs and skeletons.

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

`idlj -fall Hello.idl`

You must use the `-fall` option with the `idlj` compiler to generate both client and serverside bindings.

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

- HelloPOA.java
- _HelloStub.java
- Hello.java
- HelloHelper.java
- HelloHolder.java
- HelloOperations.java

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

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

4. Start orbd.

To start orbd from a UNIX command shell, enter:

```
orbd -ORBInitialPort 1050&
```

5. Start the HelloServer:

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

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

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

6. Run the client application:

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

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.

NBNSTIC-IT-DS

Code

Instructions:

Note: Use java 8 for this assignment.

1. Develop the interface code:

```

1.  module CalcApp
2.  {
3.  interface Calc
4.  {
5.  exception DivisionByZero {};
6.  float sum(in float a, in float b);
7.  float div(in float a, in float b) raises (DivisionByZero);
8.  float mul(in float a, in float b);
9.  float sub(in float a, in float b);
10. };
11. }; 12.

```

2. Compile the interface code with the following command:

idlj -fall CalcApp.idl

Note: This will create files stubs, and skeletons.

3. Develop the server side code:

```

1.  import CalcApp.*;
2.  import CalcApp.CalcPackage.DivisionByZero; 3.
4.  import org.omg.CosNaming.*;
5.  import org.omg.CosNaming.NamingContextPackage.*;
6.  import org.omg.CORBA.*;
7.  import org.omg.PortableServer.*; 8.
9.  import java.util.Properties; 10.
11. class CalcImpl extends CalcPOA { 12.
13.  @Override
14.  public float sum(float a, float b) {
15.  return a + b; 16.  } 17.
18.  @Override
19.  public float div(float a, float b) throws DivisionByZero {
20.  if (b == 0) {
21.  throw new CalcApp.CalcPackage.DivisionByZero();
22.  } else {
23.  return a / b;
24.  } 25.  } 26.
27.  @Override
28.  public float mul(float a, float b) {
29.  return a * b; 30.  } 31.
32.  @Override
33.  public float sub(float a, float b) {
34.  return a - b;
35.  }
36.  private ORB orb; 37.
38.  public void setORB(ORB orb_val) {
39.  orb = orb_val;
40.  }
41.  }
42.
43. public class CalcServer { 44.

```

```

45.     public static void main(String args[]) {
46.     try {
47.         // create and initialize the ORB 48.         ORB orb = ORB.init(args, null); 49.
50.         // get reference to rootpoa & activate the POAManager
51.         POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
52.         rootpoa.the_POAManager().activate(); 53.
54.         // create servant and register it with the ORB
55.         CalcImpl helloImpl = new CalcImpl(); 56.         helloImpl.setORB(orb); 57.
58.         // get object reference from the servant
59.         org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
60.         Calc href = CalcHelper.narrow(ref); 61.
62.         // get the root naming context
63.         // NameService invokes the name service
64.         org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService"); 65.         // Use NamingContextExt
which is part of the Interoperable 66.         // Naming Service (INS) specification.
67.         NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef); 68.
69.         // bind the Object Reference in Naming
70.         String name = "Calc";
71.         NameComponent path[] = ncRef.to_name(name);
72.         ncRef.rebind(path, href); 73.
74.         System.out.println("Ready.."); 75.
76.         // wait for invocations from clients
77.         orb.run();
78.     } catch (Exception e) {
79.         System.err.println("ERROR: " + e);
80.         e.printStackTrace(System.out); 81.     } 82.
83.     System.out.println("Exiting ..."); 84.
85.     }
86.     } 87.

```

4. Develop the client-side code:

```

1.     import java.io.BufferedReader;
2.     import java.io.IOException;
3.     import java.io.InputStreamReader; 4.
5.     import CalcApp.*;
6.     import CalcApp.CalcPackage.DivisionByZero; 7.
8.     import org.omg.CosNaming.*;
9.     import org.omg.CosNaming.NamingContextPackage.*;
10.    import org.omg.CORBA.*;
11.    import static java.lang.System.out; 12.
13.    public class CalcClient {
14.
15.        static Calc calcImpl;
16.        static BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); 17.
18.        public static void main(String args[]) { 19.
20.            try {
21.                // create and initialize the ORB 22.                ORB orb = ORB.init(args, null); 23.

```

```

24. // get the root naming context
25. org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService"); 26. // Use
NamingContextExt instead of NamingContext. This is 27. // part of the Interoperable naming Service.
28. NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef); 29.
30. // resolve the Object Reference in Naming
31. String name = "Calc";
32. calcImpl = CalcHelper.narrow(ncRef.resolve_str(name)); 33.
34. // System.out.println(calcImpl);
35.
36.
37. while (true) {
38. out.println("1. Sum"); 39. out.println("2. Sub"); 40. out.println("3.
Mul");
41. out.println("4. Div");
42. out.println("5. exit");
43. out.println("--");
44. out.println("choice: "); 45.
46. try {
47. String opt = br.readLine();
48. if (opt.equals("5")) {
49. break;
50. } else if (opt.equals("1")) {
51. out.println("a+b= " + calcImpl.sum(getFloat("a"), getFloat("b")));
52. } else if (opt.equals("2")) {
53. out.println("a-b= " + calcImpl.sub(getFloat("a"), getFloat("b")));
54. } else if (opt.equals("3")) {
55. out.println("a*b= " + calcImpl.mul(getFloat("a"), getFloat("b")));
56. } else if (opt.equals("4")) {
57. try {
58. out.println("a/b= " + calcImpl.div(getFloat("a"), getFloat("b")));
59. } catch (DivisionByZero de) {
60. out.println("Division by zero!!!");
61. }
62. }
63. } catch (Exception e) {
64. out.println("====");
65. out.println("Error with numbers");
66. out.println("====");
67. }
68. out.println(""); 69.
70. }
71. //calcImpl.shutdown();
72. } catch (Exception e) {
73. System.out.println("ERROR : " + e);
74. e.printStackTrace(System.out);
75. } 76. } 77.
78. static float getFloat(String number) throws Exception {
79. out.print(number + ": ");
80. return Float.parseFloat(br.readLine());
81. }
82. } 83.

```

5. Compile all the java files.

```
1. Javac *.java CalcApp/*.java 2.
```

6. Now start the orbd server though powershell.

```
1. orbd -ORBInitialPort 1050 2.
```

7. Now start the server program on new powershell window.

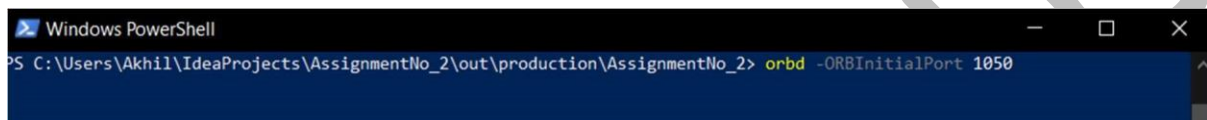
```
1. java CalcServer -ORBInitialPort 1050 -ORBInitialHost localhost 2.
```

8. Now start the client program on new powershell window. java CalcClient -
ORBInitialPort 1050 -ORBInitialHost localhost

9. Do the operations on the client end. Exit the program after usage.

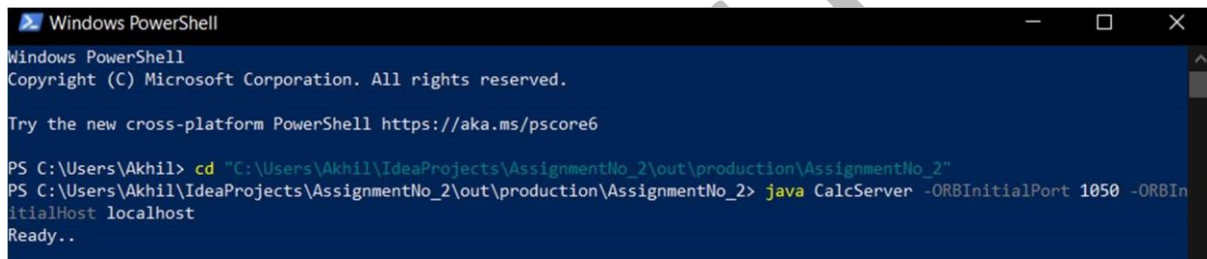
Output

1. Start ordb



```
Windows PowerShell
PS C:\Users\Akhil\IdeaProjects\AssignmentNo_2\out\production\AssignmentNo_2> orbd -ORBInitialPort 1050
```

2. Start the server



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Akhil> cd "C:\Users\Akhil\IdeaProjects\AssignmentNo_2\out\production\AssignmentNo_2"
PS C:\Users\Akhil\IdeaProjects\AssignmentNo_2\out\production\AssignmentNo_2> java CalcServer -ORBInitialPort 1050 -ORBInitialHost localhost
Ready..
```

3. Start the client

```
Windows PowerShell
PS C:\Users\Akhil\IdeaProjects\AssignmentNo_2\out\production\AssignmentNo_2> java CalcClient -ORBInitialPort 1050 -ORBInitialHost localhost
1. Sum
2. Sub
3. Mul
4. Div
5. exit
--
choice:
1
a: 2
b: 3
a+b= 5.0

1. Sum
2. Sub
3. Mul
4. Div
5. exit
--
choice:
2
a: 2
b: 3
a-b= -1.0

1. Sum
2. Sub
3. Mul
4. Div
5. exit
--
choice:
3
a: 3
b: 2
a*b= 6.0

1. Sum
2. Sub
3. Mul
4. Div
5. exit
--
choice:
4
a: 3
b: 1
a/b= 3.0

1. Sum
2. Sub
3. Mul
4. Div
5. exit
--
choice:
5
PS C:\Users\Akhil\IdeaProjects\AssignmentNo_2\out\production\AssignmentNo_2>
```

Assignment No. 3

Aim: 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.

Objectives: To learn about MPI, its benefits, and its implementation.

Infrastructure:

Software used: Python, mpi4py library, Microsoft MPI v10.0

(<https://www.microsoft.com/enus/download/details.aspx?id=57467>), Numpy

Theory:

With the advent of high-performance multicomputer, developers have been looking for messageoriented primitives that would allow them to easily write highly efficient applications. This means that the primitives should be at a convenient level of abstraction (to ease application development), and that their implementation incurs only minimal overhead.

Sockets were deemed insufficient for two reasons.

- First, they were at the wrong level of abstraction by supporting only simple send and receive primitives.
- Second, sockets had been designed to communicate across networks using general-purpose protocol stacks such as TCPIP.

They were not considered suitable for the proprietary protocols developed for high-speed interconnection networks, such as those used in high-performance server clusters. Those protocols required an 'interface that could handle more advanced features, such as different forms of buffering and synchronization.

The result was that most interconnection networks and high-performance multicomputers were shipped with proprietary communication libraries. These libraries offered a wealth of high-level and generally efficient communication primitives. Of course, all libraries were mutually incompatible, so that application developers now had a portability problem.

The need to be hardware and platform independent eventually led to the definition of a standard for message passing, simply called the Message-Passing Interface or MPI. MPI is designed for parallel applications and as such is tailored to transient communication. It makes direct use of the underlying network. Also, it assumes that serious failures such as process crashes or network partitions are fatal and do not require automatic recovery.

MPI assumes communication takes place within a known group of processes. Each group is assigned an identifier. Each process within a group is also assigned a (local) identifier. A (group/D, process/D) pair therefore uniquely identifies the source or destination of a message, and is used instead of a transport-level address. There may be several, possibly overlapping groups of processes involved in a computation and that are all executing at the same time.

At the core of MPI are messaging primitives to support transient communication, of which the most intuitive ones are summarized.

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isead	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

Figure 4-16. Some of the most intuitive message-passing primitives of MPI.

How MPI Works?

Transient asynchronous communication is supported by means of the MPI_bsend primitive. The sender submits a message for transmission, which is generally first copied to a local buffer in the MPI runtime system. When the message has been copied, the sender continues. The local MPI runtime system will remove the message from its local buffer and take care of transmission as soon as a receiver has called a receive primitive.

There is also a blocking send operation, called MPLsend, of which the semantics are implementation dependent. The primitive MPLsend may either block the caller until the specified message has been copied to the MPI runtime system at the sender's side, or until the receiver has initiated a receive operation. Synchronous communication by which the sender blocks until its request is accepted for further processing is available through the MPI_ssend primitive. Finally, the strongest form of synchronous communication is also supported: when a sender calls MPLsendrecv, it sends a request to the receiver and blocks until the latter returns a reply. Basically, this primitive corresponds to a normal RPC.

Both MPLsend and MPLssend have variants that avoid copying messages from user buffers to buffers internal to the local MPI runtime system. These variants correspond to a form of asynchronous communication. With MPI_isead, a sender passes a pointer to the message after which the MPI runtime system takes care of communication. The sender immediately continues. To prevent overwriting the message before communication completes, MPI offers primitives to check for completion, or even to block if required. As with MPLsend, whether the message has actually been transferred to the receiver or that it has merely been copied by the local MPI runtime system to an internal buffer is left unspecified. Likewise, with MPLissend, a sender also passes only a pointer to the MPI runtime system. When the runtime system indicates it has processed the message, the sender is then guaranteed that the receiver has accepted the message and is now working on it. The operation MPLrecv is called to receive a message; it blocks the caller until a message arrives. There is also an asynchronous variant, called MPLirecv, by which a receiver indicates that is prepared to accept a message. The receiver can check whether or not a message has indeed arrived, or block until one does.

What is mpi4py?

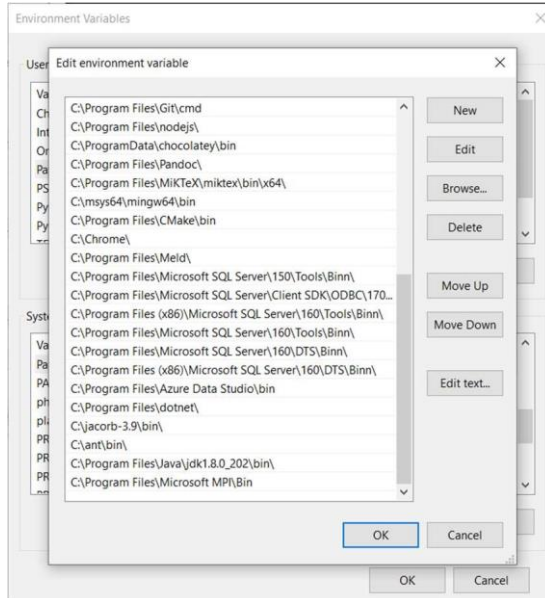
MPI for Python provides MPI bindings for the Python language, allowing programmers to exploit multiple processor computing systems. mpi4py is constructed on top of the MPI-1/2 specifications and provides an object oriented interface which closely follows MPI-2 C++ bindings. Installing mpi4py

```
pip install mpi4py
```

Microsoft MPI V10

Installation Process

1. Download the MPI exe file from the URL provided above.
2. Install the exe file.
3. Setup the path in the System Variable to the installed locations bin folder.



i.e. As visible in the last line.

Note: This path may differ according to the path set at the installation time.

4. To verify the MPI software is installed correctly, open a new command prompt and type.

```
1. mpiexec -help
```

If everything is installed correctly, it will show the list of options available in the MPI.

Developing the code

1. For distributed addition of the array of numbers we are going to use Reduce method available in the MPI operation.
2. For this first of all we are going to initialize the MPI, followed by the accepting the number of processes for performing the computation.
3. Then we assign rank to each process.
4. Initialize the array, and its value.
5. Send the sub-array of equal size to each process, where it gets computed.
6. This computed value from each process is added together to get the global value or the total sum of the array.

Compiling the Code

We run the code by the following command.

```
1. mpiexec -np 3 python sum.py
```

Here, 3 represent the number of processes.

Conclusion:

We learnt about MPI, how it was introduced and how to implement distributed computing with the help of MPI.

Code:

```

1. from mpi4py import MPI 2.
import numpy as np 3.
4. comm =
MPI.COMM_WORLD 5. rank =
comm.rank 6.
7. send_buf = []
8.
9.     if rank == 0:
10.         arr = np.array([12,21241,5131,1612251,161,6,161,1613,161363,12616,367,8363])
11.         arr.shape = (3, 4)
12.         send_buf = arr
13.         v = comm.scatter(send_buf, root=0)
14.         print("Local sum at rank{0}: {1}".format(comm.rank, np.sum(v))) 15. recvbuf = comm.reduce(v, root=0)
16.         if comm.rank == 0:
17.             global_sum = np.sum(recvbuf)
18.             print("Global sum: "+ str(global_sum)) 19.

```

Compile

1. Open terminal, and change the directory to the python files directory.
2. Type the following command on terminal.

```
mpiexec -np 3 python sum.py
```

3. The local sum and global sum values would be displayed.

Output

MPI of array of values.

```

PS D:\College\BE\Sem 8\DS\Lab\DS-LAB\Code\Implementation\Assignment No. 3> mpiexec -np 3 python sum.py

Local sum at rank1: 8
Local sum at rank2: 12
Local sum at rank0: 4
Global sum: 24
PS D:\College\BE\Sem 8\DS\Lab\DS-LAB\Code\Implementation\Assignment No. 3>

```

Assignment No. 4

Aim: Implement Berkeley algorithm for clock synchronization.

Objective: To understand Berkeley algorithm for clock synchronization and its implementation.

Infrastructure: Python environment.

Software Requirements: Python 3.0.

Theory:

Berkeley Algorithm

Berkeley's Algorithm is a distributed algorithm for computing the correct time in a network of computers. The algorithm is designed to work in a network where clocks may be running at slightly different rates, and some computers may experience intermittent communication failures.

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

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

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

Benefit: It is relatively simple to implement and understand.

Limitation: The time computed by the algorithm is based on the network conditions and time of sending and receiving timestamps which can't be very accurate and also it has a requirement of a master computer which if failed can cause the algorithm to stop working.

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

Scope of Improvement

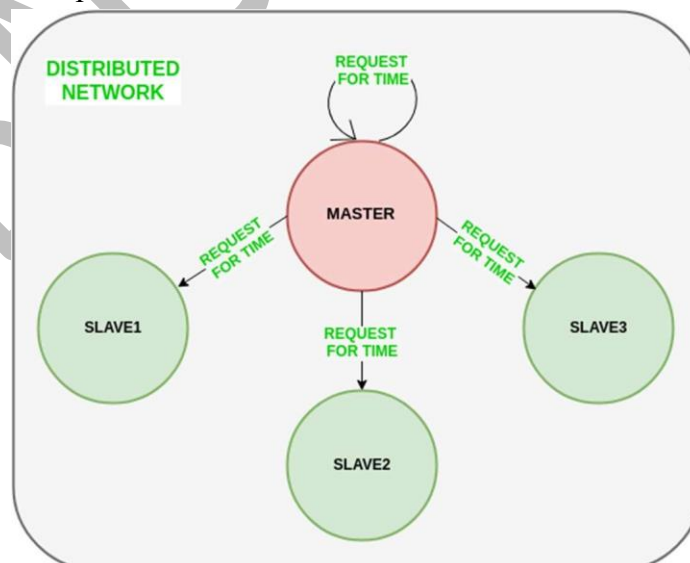
There are several areas where Berkeley's Algorithm can be improved –

- Accuracy – The algorithm calculates the average time based on the timestamps received from all the computers in the network, which can lead to inaccuracies, especially if the network has a high degree of jitter or delay.
- Robustness – The algorithm requires a master computer, which if it fails, can cause the algorithm to stop working. If the master computer is a single point of failure, it can make the algorithm less robust.
- Synchronization Quality – The algorithm assumes that all the clocks in the network are running at the same rate, but in practice, clocks may drift due to temperature, aging, or other factors. The algorithm doesn't consider this drift and may fail to achieve a high degree of synchronization between the clocks in the network.
- Security – There is no security measures in the algorithm to prevent malicious computers from tampering with the timestamps they send to the master computer, which can skew the results of the algorithm.
- Adaptability – The algorithm doesn't adapt well to changes in the network, for example, if a new computer is added to the network or if the network topology changes.

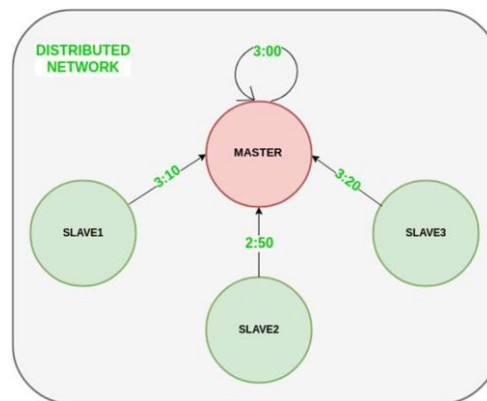
How to use Berkeley's Algorithm

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

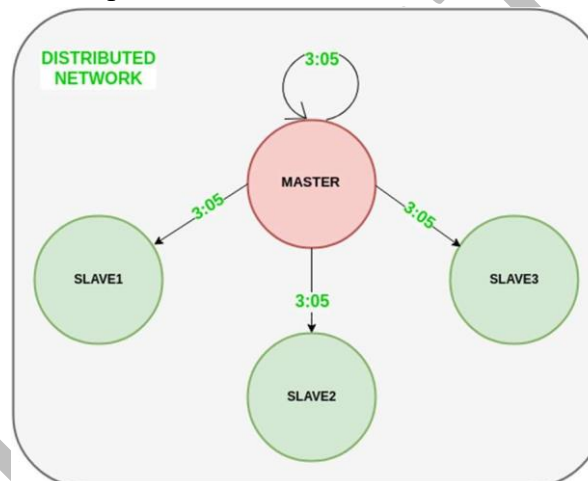
1. Designate one computer in the network as the master computer. This computer will be responsible for receiving timestamps from all the other computers in the network and computing the correct time. The master node is chosen using an election process/leader election algorithm.
2. On each computer in the network, set up a timer to periodically send the computer's local time to the master computer.



3. On the master computer, set up a mechanism for receiving timestamps from all the computers in the network.



4. On the master computer, implement the logic for calculating the average time based on the received timestamps.
5. On the master computer, set up a mechanism for sending the calculated average time back to all the computers in the network.
6. On each computer in the network, set up a mechanism for receiving the time from the master computer and setting the computer's clock to that time.



7. Repeat the process periodically, for example, every 30 seconds or 1 minute, to ensure that the clocks in the network stay synchronized.

Conclusion:

We learnt about Berkeley's algorithm for clock synchronization and its implementation.

Code

1. Server

```

1. import threading
2. import datetime
3. import socket
4. import time
5. from dateutil import parser
6.
7. # Client data to be stored in variable client_data 8. client_data = {}
9.
10. # Nested thread function used to receive clock time from a connected client 11. def
startReceivingClockTime(connector, address):
12.     while True:
13.         # receive clock time
14.         clock_time_string = connector.recv(1023).decode()
15.         clock_time = parser.parse(clock_time_string)
16.         clock_time_diff = datetime.datetime.now() - clock_time
17.         client_data[address] = {
18.             "clock_time": clock_time,
19.             "time_difference": clock_time_diff,
20.             "connector": connector
21.         }
22.         print("Client Date updated with: " + str(address), end="\n\n")
23.         time.sleep(5)
24.
25. # This opens up the master server to accept clients over given port 26. def
startConnecting(master_server):
27.
28.     #fetching clock tome at slaves / clients 29.     while True:
30.         master_slave_connector, addr = master_server.accept()
31.         slave_address = str(addr[0]) + ":" + str(addr[1])
32.         print(slave_address + " got connected successfully")
33.         current_thread = threading.Thread(
34.             target = startReceivingClockTime,
35.             args = (master_slave_connector, slave_address,)
36.         )
37.         current_thread.start()
38.
39.
40.
41. # Used to fetch average clock difference 42. def
getAverageClockDiff():
43.     current_client_data = client_data.copy()
44.     time_difference_list = list(client['time_difference'] for clientadd, client in client_data.items())
45.     sum_of_clock_difference = sum(time_difference_list, datetime.timedelta(0,0))
46.     average_clock_difference = sum_of_clock_difference / len(client_data)
47.     return average_clock_difference
48.
49. # Master sync thread function used to generate cycles of clock synchronization in the newtwork
50. def synchronizeAllClocks():
51.     while True:
52.         print("New synchronization cycle started.")
53.         print("Number of clients to be synchronized: " + str(len(client_data)))
54.         if len(client_data) > 0:
55.             average_clock_difference = getAverageClockDiff()
56.
57.         for client_addr, client in client_data.items():
58.             try:
59.                 synchronized_time = datetime.datetime.now() + average_clock_difference
60.                 client["connector"].send(str(synchronized_time).encode())
61.
62.             except Exception as e:
63.

```

```

64.         print("Something went wring while sending synchronizied time through"+str(client_addr)) 65.
else:
66.     print("No client data. Synchronization not applicable.") 67.
68.     print("\n\n")
69.
70.     time.sleep(5) 71.
72.     def initiateClockServer(port = 8080):
73.         master_server = socket.socket()
74.         master_server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) 75.
76.     print("Socket at master node created successfully") 77.
78.     HOST= "127.0.0.1"
79.     master_server.bind((HOST, port)) 80.
81.     # Starts listening to requests
82.     master_server.listen(10)
83.     print("Clock server started...") 84.
85.     # start making connections
86.     print("Starting to make connections...")
87.     master_thread = threading.Thread(
88.         target = startConnecting,
89.         args = (master_server,)
90.     )
91.     master_thread.start() 92.
93.     # start synchronization
94.     print("Starting synchronization parallely...")
95.     sync_thread = threading.Thread(
96.         target = synchronizeAllClocks,
97.         args = ()
98.     )
99.     sync_thread.start()
100.
101.
102.     # Driver function
103.     if __name__ == '__main__':
104.         # Trigger the Clock Server
105.         initiateClockServer(port=8080) 106.

```

2. Client

```

1.     from timeit import default_timer as timer
2.     from dateutil import parser
3.     import threading
4.     import datetime
5.     import socket 6. import time 7.
8.
9. # client thread function used to send time at client side 10. def
startSendingTime(slave_client):
11.
12.     while True:
13.         # provide server with clock time at the client
14.         slave_client.send(str(datetime.datetime.now()).encode()) 15.
16.         print("Recent time sent successfully", end = "\n\n")
17.         time.sleep(5) 18.
19.
20. # client thread function used to receive synchronized time

```

```

21. def startReceivingTime(slave_client):
22.
23.     while True:
24.         # receive data from the server
25.         Synchronized_time = parser.parse(slave_client.recv(1024).decode()) 26.
27.         print("Synchronized time at the client is: " + str(Synchronized_time), end = "\n\n") 28.
29.
30. # function used to Synchronize client process time 31. def
initiateSlaveClient(port = 8080):
32.
33.     slave_client = socket.socket() 34.
35.     # connect to the clock server on local computer 36.
slave_client.connect(('127.0.0.1', port)) 37.
38.     # start sending time to server
39.     print("Starting to receive time from server\n")
40.     send_time_thread = threading.Thread(
41.         target = startSendingTime,
42.         args = (slave_client, )) 43.     send_time_thread.start() 44.
45.
46.     # start receiving synchronized from server
47.     print("Starting to receiving " + "synchronized time from server\n")
48.     receive_time_thread = threading.Thread(
49.         target = startReceivingTime,
50.         args = (slave_client, )) 51.     receive_time_thread.start() 52.
53.
54. # Driver function
55. if __name__ == '__main__':
56.
57.     # initialize the Slave / Client
58.     initiateSlaveClient(port = 8080) 59.

```

Steps to run:

1. Create the server and client python files.
2. Open PowerShell can change directory to the source code directory.
3. Run the server file

```
1. python server.py 2.
```

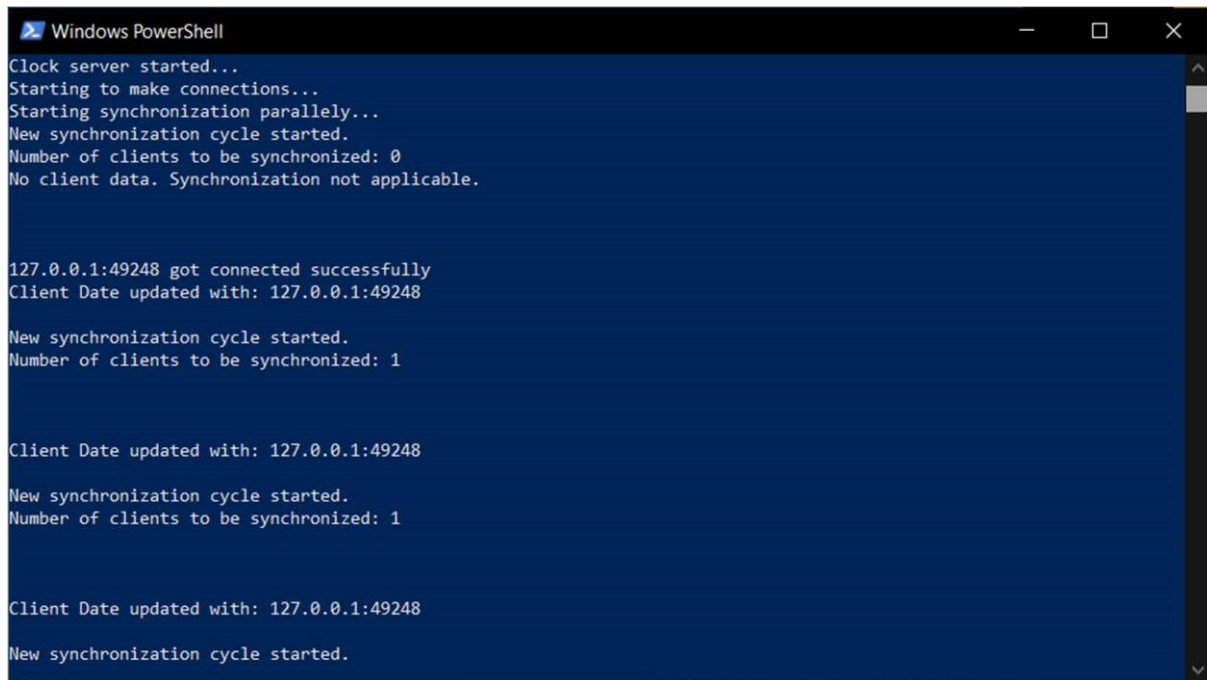
4. Open new PowerShell and run the client program.

```
1. python client.py 2.
```

5. The number of times you create new PowerShell and run the client code, so many clients will get connected to the server and will be displayed at server end. And the clients will display the timestamp.

Output

1. Server Started



```
Windows PowerShell
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.

127.0.0.1:49248 got connected successfully
Client Date updated with: 127.0.0.1:49248

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

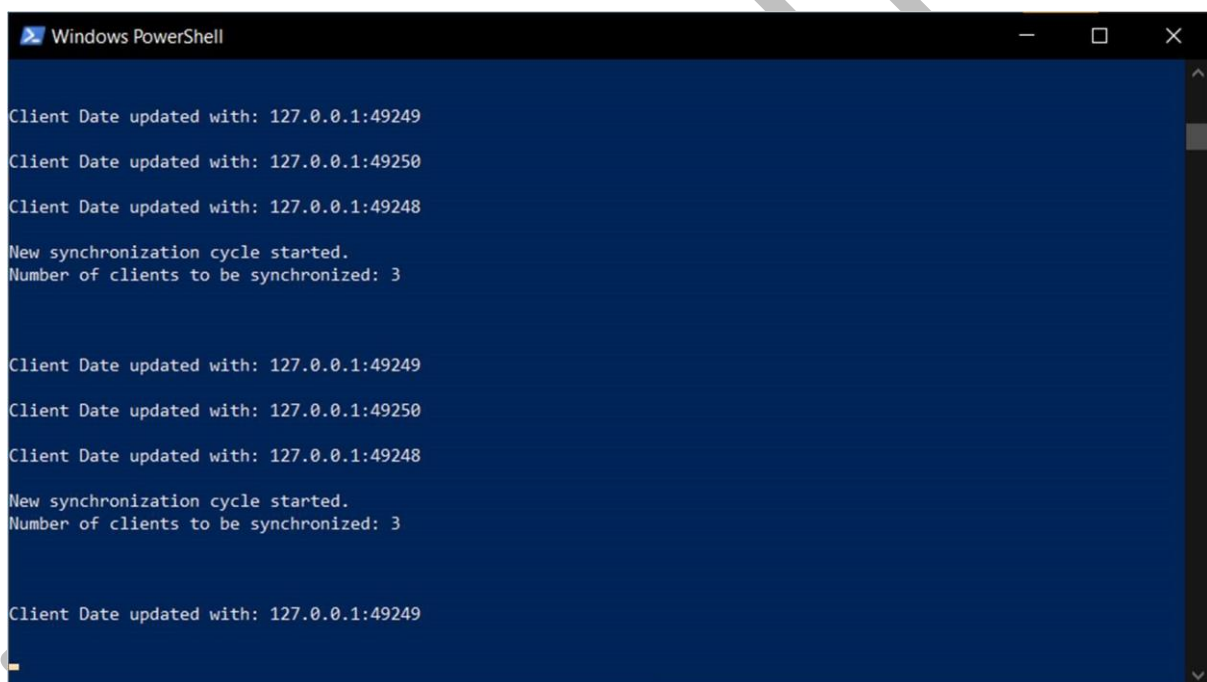
Client Date updated with: 127.0.0.1:49248

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

Client Date updated with: 127.0.0.1:49248

New synchronization cycle started.
```

2. After receiving request from clients.



```
Windows PowerShell

Client Date updated with: 127.0.0.1:49249
Client Date updated with: 127.0.0.1:49250
Client Date updated with: 127.0.0.1:49248

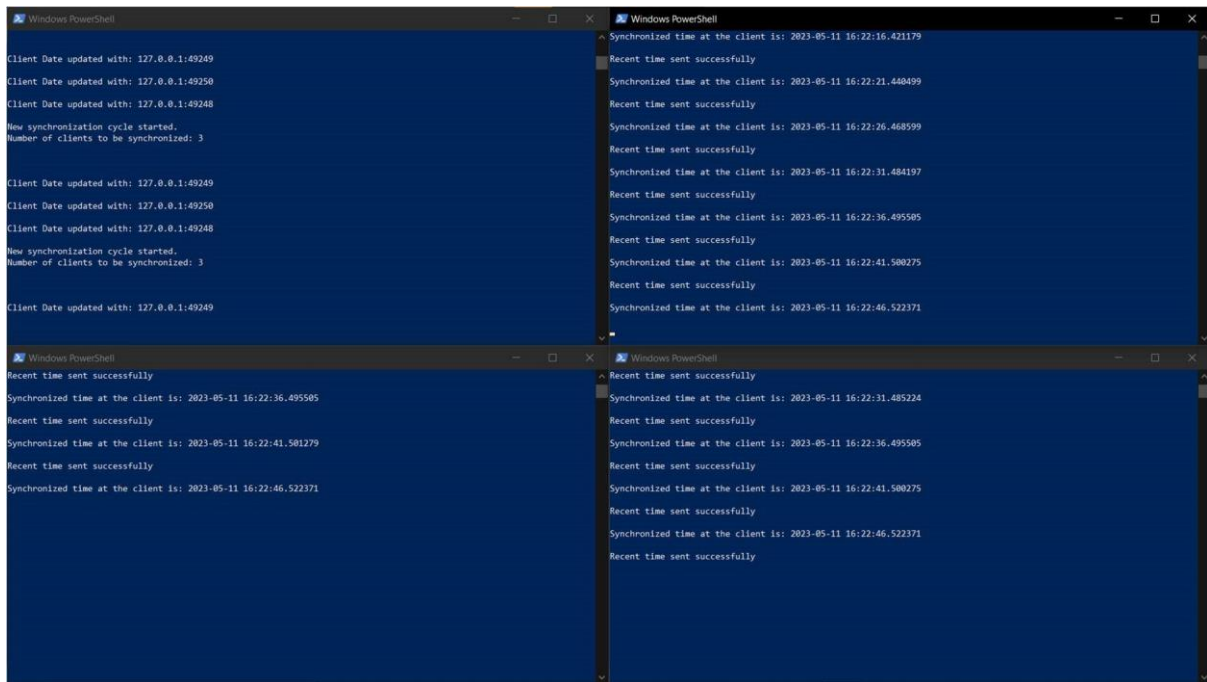
New synchronization cycle started.
Number of clients to be synchronized: 3

Client Date updated with: 127.0.0.1:49249
Client Date updated with: 127.0.0.1:49250
Client Date updated with: 127.0.0.1:49248

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

Client Date updated with: 127.0.0.1:49249
```

3. The server terminal and clients terminal.



The image displays four Windows PowerShell terminal windows arranged in a 2x2 grid, showing logs for a distributed system synchronization process. The logs indicate that a new synchronization cycle has started, with 3 clients to be synchronized. The logs show the client date being updated and the recent time sent successfully for each client. The logs also show the synchronized time at the client for each client.

```
Client Date updated with: 127.0.0.1:49249
Client Date updated with: 127.0.0.1:49250
Client Date updated with: 127.0.0.1:49248
New synchronization cycle started.
Number of clients to be synchronized: 3

Client Date updated with: 127.0.0.1:49249
Client Date updated with: 127.0.0.1:49250
Client Date updated with: 127.0.0.1:49248
New synchronization cycle started.
Number of clients to be synchronized: 3

Client Date updated with: 127.0.0.1:49249

Synchronized time at the client is: 2023-05-11 16:22:16.421179
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:21.440499
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:26.468599
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:31.484197
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:36.495505
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:41.508275
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:46.522371

Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:36.495505
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:41.508279
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:46.522371

Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:31.485224
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:36.495505
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:41.508275
Recent time sent successfully
Synchronized time at the client is: 2023-05-11 16:22:46.522371
Recent time sent successfully
```

Assignment No.5

Aim: Implement Token Ring based Mutual Exclusion Algorithm. Tools and Environment: Java Runtime Environment, Java JDK Theory:

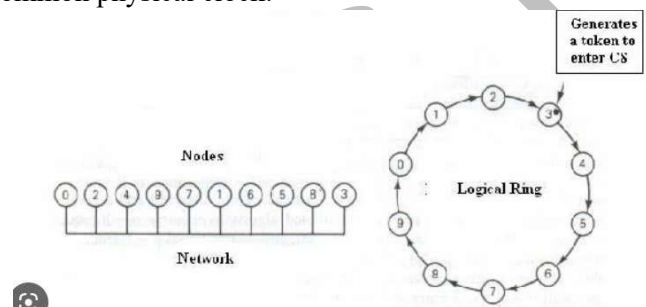
Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process cannot enter its critical section while another concurrent process is currently present or executing in its critical section i.e. only one process is allowed to execute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

Mutual exclusion in single computer system Vs. distributed system:

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

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

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



Requirements of Mutual exclusion Algorithm:

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

Solution to distributed mutual exclusion:

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

Token Based Algorithm:

A unique token is shared among all the sites.

If a site possesses the unique token, it is allowed to enter its critical section

This approach uses sequence number to order requests for the critical section.

Each request for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.

This approach ensures Mutual exclusion as the token is unique Example:
Suzuki-Kasami's Broadcast Algorithm

Conclusion: Hence Token Ring algorithm achieves mutual exclusion in a distributed system by creating a bus network of processes.

Code

1. Token

```

1. import threading 2.
import time 3.
4. class TokenRing:
5.     def __init__(self, numProcesses):
6.         self.num_processes = numProcesses
7.         self.threads = []
8.         self.mutex = threading.Semaphore(1)
9.         self.tokens = [threading.Event() for _ in range(numProcesses)]
10.        self.current_token = 0 11.
12.        for i in range(numProcesses):
13.            t = threading.Thread(target=self.process, args=(i,)) 14.            self.threads.append(t) 15.
16.        def start(self):
17.            for thread in self.threads: 18.                thread.start() 19.
20.        def process(self, process_id):
21.            while True:
22.                self.tokens[process_id].wait() 23.
24.                self.mutex.acquire()
25.                print("Process id: ", process_id, " is in critical section." )
26.                time.sleep(2) 27.
28.                self.mutex.release()
29.                print("Process id: ", process_id, "is released") 30.
31.
32.            next_process_id = (process_id + 1)%self.num_processes
33.            self.tokens[next_process_id].set() 34.
35.            self.tokens[process_id].clear() 36.
37.        def initalize_token_ring(self): 38.
39.            self.tokens[0].set() 39.
40.            if __name__ == "__main__":
41.                num_processes = 4
42.                tokenRing = TokenRing(num_processes)
43.                tokenRing.start()
44.                tokenRing.initalize_token_ring() 45.

```

Output

Token ring accessing critical section and releasing critical section.

```
8\DS\Lab\DS-LAB'; & 'C:\Users\Akhil\anaconda3\python.exe' 'c:\Users\Akhil\.vscode\extensions\ms-python  
.python-2023.8.0\pythonFiles\lib\python\debugpy\adapter\..\..\debugpy\launcher' '55406' '--' 'D:\Colle  
ge\BE\Sem 8\DS\Lab\DS-LAB\Code\Implementation\Assignment No. 5\TokenRing.py'  
Process id: 0 is in critical section.  
Process id: 0 is released  
Process id: 1 is in critical section.  
Process id: 1 is released  
Process id: 2 is in critical section.  
Process id: 2 is released  
Process id: 3 is in critical section.  
Process id: 3 is released  
Process id: 0 is in critical section.  
Process id: 0 is released  
Process id: 1 is in critical section.  
Process id: 1 is released  
Process id: 2 is in critical section.
```

Assignment No. 6

Problem Statement: Implementing Bully and Ring Algorithm for Leader Election. Tools and environment: C++ Programming environment.

Theory:

Election algorithms choose a process from a group of processors to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor. Election algorithm basically determines where a new copy of the coordinator should be restarted. Election algorithm assumes that every active process in the system has a unique priority number. The process with highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has highest priority number. Then this number is sent to every active process in the distributed system. We have two election algorithms for two different configurations of a distributed system.

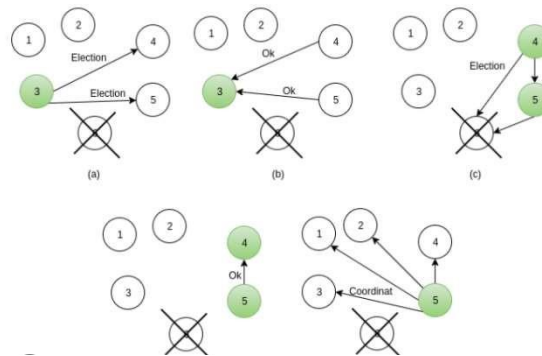
1. The Bully Algorithm

This algorithm applies to system where every process can send a message to every other process in the system.

Algorithm –

Suppose process P sends a message to the coordinator. If the coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed. Now process P sends an election message to every process with high priority number. It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator. Then it sends a message to all lower priority number processes that it is elected as their new coordinator. However, if an answer is received within time T from any other process Q,

- (I) Process P again waits for time interval T' to receive another message from Q that it has been elected as coordinator.
- (II) If Q doesn't respond within time interval T' then it is assumed to have failed and algorithm is restarted.



2. The Ring Algorithm –

This algorithm applies to systems organized as a ring (logically or physically). In this algorithm we assume that the link between the process is unidirectional and every process can message to the process on its right only. Data structure that this algorithm uses is active list, a list that has a priority number of all active processes in the system.

Algorithm

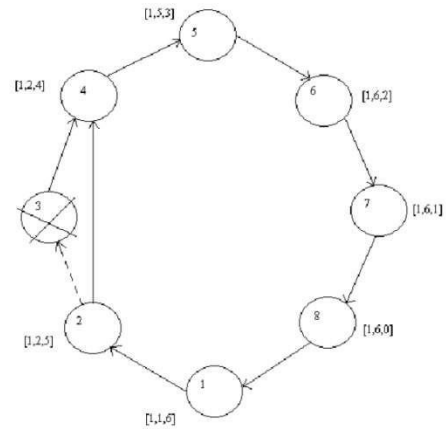
If process P1 detects a coordinator failure, it creates new active list which is empty initially. It sends election message to its neighbor on right and adds number 1 to its active list.

If process P2 receives message elect from processes on left, it responds in 3 ways:

- (I) If message received does not contain 1 in active list, then P1 adds 2 to its active list and forwards the message.
- (II) If this is the first election message it has received or sent, P1 creates new active list with numbers 1 and 2. It then sends election message 1 followed by 2.
- (III) If Process P1 receives its own election message 1 then active list for P1 now contains numbers of all the active processes in the system. Now Process P1 detects highest priority number from list and elects it as the new coordinator.

Time Complexity: $O(n^2)$ in the worst-case scenario, where n is the number of processes.

Space complexity: $O(n)$



Conclusion: The bully algorithm is a type of Election algorithm which is mainly used for choosing a coordinate. Hence in a distributed system, we need some election algorithms such as bully and ring to get a coordinator that performs functions needed by other processes.

Code

Bully Algorithm

Bully.py

```
1. from statistics import mode 2.
3.     class Process:
4.     def __init__(self, process_id, total_count):
5.     self.process_id = process_id
6.     self.total_count = total_count
7.     self.leader_id = -1
8.     self.is_active = True 9.
10.    def crash(self):
11.    self.is_active = False 12.
13.    def start(self):
14.    self.is_active = True 15.
16.    def is_leader(self):
17.    if self.process_id == self.leader_id:
18.    return True 19.    return False 20.
21.    def set_leader(self, leader): 22.
self.leader_id = leader 23.
24.    def get_leader(self):
25.    return self.leader_id 26.
27.    def sendRequest(self, toProcess):
28.    print(f"Sending request to process {toProcess.process_id} from {self.process_id}") 29.
if(toProcess.recvRequest(self.process_id)):
30.    print(f"Ok recived from {toProcess.process_id}")
31.    self.set_leader(toProcess.process_id)
```



```

32.         else:
33.             print(f"No response from {toProcess.process_id}")
34.
35.
36.         def receiveRequest(self, fromProcess):
37.             if(self.is_active):
38.                 print(f"Received request from process {fromProcess}.")
39.                 return self.recivedMessage()
40.             return False
41.
42.     def recivedMessage(self):
43.         return True
44.
45.     class Bully:
46.         def __init__(self, total_count):
47.             self.processes = []
48.             self.total_count = total_count
49.             # self.leader = None
50.
51.             def intializeProcesses(self):
52.                 self.processes = []
53.                 for i in range(self.total_count):
54.                     self.processes.append(Process(i, total_count = self.total_count))
55.                 self.elect_leader()
56.                 self.coordinator()
57.
58.                 def elect_leader(self, current=0):
59.                     for i in range(current, self.total_count):
60.                         if self.processes[i].is_active:
61.                             # [self.processes[i].sendRequest(self.processes[j]) for j in range(i, self.total_count)]
62.                             for j in range(i+1, self.total_count):
63.                                 if(self.processes[j].is_active):
64.                                     self.processes[i].sendRequest(self.processes[j])
65.                                     elif(not self.processes[j].is_active and i+1==self.total_count-1):
66.                                         self.processes[i].sendRequest(self.processes[i])
67.
68.                     if self.processes[i].get_leader() == -1:
69.                         self.processes[i].sendRequest(self.processes[i])
70.                         # if(i==self.total_count-1):
71.                         # self.processes[i].sendRequest(self.processes[i])
72.
73.                 def crash(self, crash_id):
74.                     if(crash_id < self.total_count and crash_id >= 0):
75.                         self.processes[crash_id].crash()
76.                         # print(f"Process id {Process.process_id} crashed.")
77.                         if(self.processes[crash_id].is_leader()):
78.                             print("Leader process Down.\n Initiating the leader lookout.")
79.                             self.elect_leader(0)
80.
81.             def start(self, process_id):
82.                 if(self.processes[process_id].is_active):
83.                     print("Process already active")
84.                 else:
85.                     self.processes[process_id].start()
86.                     self.elect_leader()
87.                     # if(self.processes[process_id].is_active):
88.                     #     if process_id > self.processes[self.leader].get_leader():
89.                         # self.elect_leader(self.leader)
90.
91.             def coordinator(self):
92.                 leader = []
93.                 for p in self.processes:
94.
95.                     if p.is_active:
96.                         print(p.get_leader())
97.                         leader.append(p.get_leader())
98.                 self.leader = mode(leader)
99.
100.

```

Driver.py

```

1. from Bully import Bully 2.
#Dummy Processes 3.
4.     process_count = int(input("Enter Number of Processes"))
5.     bully = Bully(process_count) 6. bully.intializeProcesses() 7.
8. state = True
9.
10.    while state:
11.        print("1. Initialize the process\n2. Bring Down process\n3. Activate Process\n4. Exit \n
5. Current Coordinator\n")
12.    choice = int(input()) 13.
14.    if(choice==1):
15.        bully.intializeProcesses() 15.
16.    elif(choice==2):
17.        crash_id = int(input("Enter the process you want to crash"))
18.        bully.crash(crash_id) 19.
20.    elif(choice==3):
21.        process_id = int(input("Enter the process you want to start"))
22.        bully.start(process_id) 23.
24.    elif(choice==4):
25.        state=False
26.        print("Exiting the program") 27.
28.    elif(choice==5):
29.        bully.coordinator() 30.    else:
31.        print("Invalid Input") 32.

```

Ring Algorithm

```

1.     class Pro:
2.     def __init__(self, id):
3.         self.id = id 4.         self.act = True
5.
6.     class GFG:
7.     def __init__(self):
8.         self.TotalProcess = 0
9.         self.process = [] 10.
11.         def initialiseGFG(self):
12.             print("No of processes 5")
13.             self.TotalProcess = 5
14.             self.process = [Pro(i) for i in range(self.TotalProcess)]
15.
16.         def Election(self):
17.             print("Process no " + str(self.process[self.FetchMaximum()].id) + " fails")
18.             self.process[self.FetchMaximum()].act = False
19.             print("Election Initiated by 2")
20.             initializedProcess = 2
21.
22.             old = initializedProcess
23.             newer = old + 1
24.
25.             while (True):
26.                 if (self.process[newer].act):
27.                     print("Process " + str(self.process[old].id) + " pass
Election(" + str(self.process[old].id) + ") to" + str(self.process[newer].id))

```

```

28.         old = newer
29.         newer = (newer + 1) % self.TotalProcess
30.         if (newer == initializedProcess):
31.             break
32.
33.         print("Process " + str(self.process[self.FetchMaximum()].id) + " becomes coordinator")
34.         coord = self.process[self.FetchMaximum()].id
35.
36.         old = coord
37.         newer = (old + 1) % self.TotalProcess
38.         while (True):
39.             if (self.process[newer].act):
40.                 print("Process " + str(self.process[old].id) + " pass
Coordinator(" + str(coord) + ") message to process " + str(self.process[newer].id))
41.                 old = newer
42.                 newer = (newer + 1) % self.TotalProcess
43.                 if (newer == coord):
44.                     print("End Of Election ")
45.                     break
46.
47.         def FetchMaximum(self):
48.             maxId = -9999
49.             ind = 0
50.             for i in range(self.TotalProcess):
51.                 if (self.process[i].act and self.process[i].id > maxId):
52.                     maxId = self.process[i].id
53.                     ind = i
54.             return ind
55.
56.     def main():
57.         object = GFG()
58.         object.initialiseGFG()
59.         object.Election()
60.         if __name__ == "__main__":
61.             main()
62.
63.

```

Output

Bully Algorithm for leader election.

1. Initializing the Processes

```
D:\College\BE\Sem 8\DS\Lab\DS-LAB>set MSMPI_LIB32=""
PS D:\College\BE\Sem 8\DS\Lab\DS-LAB> & 'C:\Users\Akhil\anaconda3\python.exe' 'c:\Users\Akhil\.vscode\extensions\ms-python.python-2023.8.0\pythonFiles\lib\python\debugpy\adapter\..\..\debugpy\launcher' '55424' '--' 'D:\College\BE\Sem 8\DS\Lab\DS-LAB\Code\Implementation\Assignment No. 6\BullyAlgorithm\Driver.py'
Enter Number of Processes5
Sending request to process 1 from 0
Recived request from process 0.
Ok recived from 1
Sending request to process 2 from 0
Recived request from process 0.
Ok recived from 2
Sending request to process 3 from 0
Recived request from process 0.
Ok recived from 3
Sending request to process 4 from 0
Recived request from process 0.
Ok recived from 4
Sending request to process 2 from 1
Recived request from process 1.
Ok recived from 2
Sending request to process 3 from 1
Recived request from process 1.
Ok recived from 3
Sending request to process 4 from 1
Recived request from process 1.
Ok recived from 4
Sending request to process 3 from 2
Recived request from process 2.
Ok recived from 3
Sending request to process 4 from 2
Recived request from process 2.
Ok recived from 4
Sending request to process 4 from 3
Recived request from process 3.
Ok recived from 4
Sending request to process 4 from 4
Recived request from process 4.
Ok recived from 4
4
4
4
4
4
```

2. Bringing down one process

```

1. Initialize the process
2. Bring Down process
3. Activate Process
4. Exit
5. Current Coordinator

2
Enter the process you want to crash4
Leader process Down.
  Initialing the leader lookout.
Sending request to process 1 from 0
Recived request from process 0.
Ok recived from 1
Sending request to process 2 from 0
Recived request from process 0.
Ok recived from 2
Sending request to process 3 from 0
Recived request from process 0.
Ok recived from 3
Sending request to process 2 from 1
Recived request from process 1.
Ok recived from 2
Sending request to process 3 from 1
Recived request from process 1.
Ok recived from 3
Sending request to process 3 from 2
Recived request from process 2.
Ok recived from 3
Sending request to process 3 from 3
Recived request from process 3.
Ok recived from 3

```

3. Seeing current Coordinator

```

Ok recived from 3
1. Initialize the process
2. Bring Down process
3. Activate Process
4. Exit
5. Current Coordinator

5
3
3
3
3
1. Initialize the process
2. Bring Down process
3. Activate Process
4. Exit
5. Current Coordinator

```

4. Starting the process

```

3
Enter the process you want to start4
Sending request to process 1 from 0
Recived request from process 0.
Ok recived from 1
Sending request to process 2 from 0
Recived request from process 0.
Ok recived from 2
Sending request to process 3 from 0
Recived request from process 0.
Ok recived from 3
Sending request to process 4 from 0
Recived request from process 0.
Ok recived from 4
Sending request to process 2 from 1
Recived request from process 1.
Ok recived from 2
Sending request to process 3 from 1
Recived request from process 1.
Ok recived from 3
Sending request to process 4 from 1
Recived request from process 1.
Ok recived from 4
Sending request to process 3 from 2
Recived request from process 2.
Ok recived from 3
Sending request to process 4 from 2
Recived request from process 2.
Ok recived from 4
Sending request to process 4 from 3
Recived request from process 3.
Ok recived from 4
1. Initalize the process
2. Bring Down process
3. Activate Process
4. Exit
5. Current Coordinator

5
4
4
4
4
4
4

```

Ring Algorithm for Leader Election

```
PS D:\College\BE\Sem 8\DS\Lab\DS-LAB> d:; cd 'd:\College\BE\Sem 8\DS\Lab\DS-LAB'; & 'C:\Users\Akhil\anaconda3\python.exe' 'c:\Users\Akhil\.vscode\extensions\ms-python.python-2023.8.0\pythonFiles\lib\python\debugpy\adapter\..\..\debugpy\launcher' '55782' '--' 'D:\College\BE\Sem 8\DS\Lab\DS-LAB\Code\Implementation\Assignment No. 6\RingElectionAlgorithm.py'
No of processes 5
Process no 4 fails
Election Initiated by 2
Process 2 pass Election(2) to3
Process 3 pass Election(3) to0
Process 0 pass Election(0) to1
Process 3 becomes coordinator
Process 3 pass Coordinator(3) message to process 0
Process 0 pass Coordinator(3) message to process 1
Process 1 pass Coordinator(3) message to process 2
End Of Election
```


Assignment No. 7

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

Objective: To understand web services, and how distributed applications can be developed to consume web services.

Infrastructure: Python environment.

Software Requirements: Python 3.0, Flask, request library

Theory:

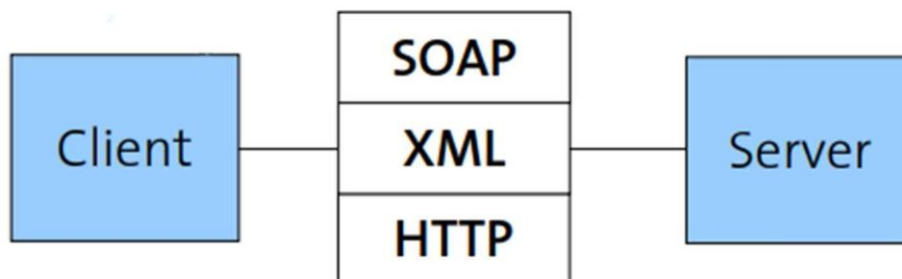
What are Web Services?

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer.

Components of Web Services

The basic web services platform is XML + HTTP. All the standard web services work using the following components –

- SOAP (Simple Object Access Protocol)
- UDDI (Universal Description, Discovery and Integration)
- WSDL (Web Services Description Language)



How Does a Web Service Work?

A web service enables communication among various applications by using open standards such as HTML, XML, WSDL, and SOAP. A web service takes the help of -

- XML to tag the data
- SOAP to transfer a message
- WSDL to describe the availability of service.

Types of Web Services

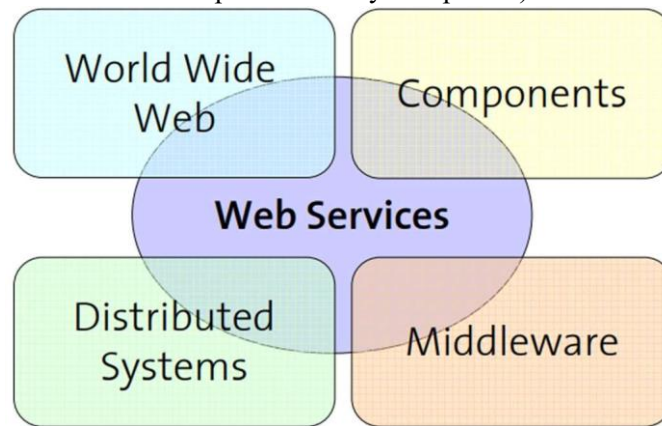
There are mainly two types of web services.

- (I) SOAP web services.
- (II) RESTful web services.

Distributed Systems and Web Services

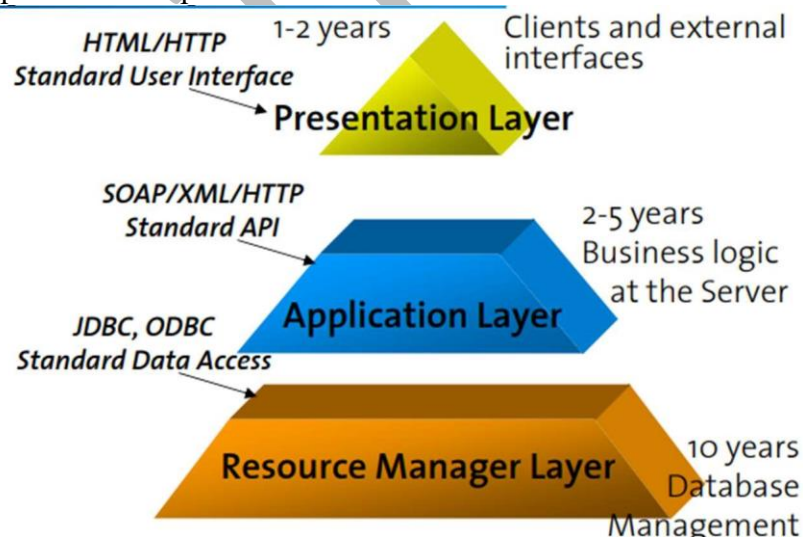
- Web services provide standards for developing large scale distributed system.

- Web services on the path of success while CORBA distributed objects failed (This is nothing technical, only a matter of widespread industry acceptance)



Layers in Distributed Systems

- Client is any user or program that wants to perform an operation over the system. To support a client, the system needs to have a presentation layer through which the user can submit operations and obtain a result.
- The application logic establishes what operations can be performed over the system and how they take place. It takes care of enforcing the business rules and establish the business processes. The application logic can be expressed and implemented in many different ways: constraints, business processes, server with encoded logic ...
- The resource manager deals with the organization (storage, indexing, and retrieval) of the data necessary to support the application logic. This is typically a database but it can also be a text retrieval system or any other data management system providing querying capabilities and persistence.



Steps involved in development of Web service, and a distributed application to utilize this Web services are:

1. Setting up the Web Service:
 - Choose a programming language and framework for your web service.

- Define the functionality and endpoints of your web service. For simplicity, let's assume you want to create a basic calculator API with two endpoints: POST /add and POST /multiply.
- 2. Deploying the Web Service:
You can deploy the web service on any cloud platform.
Alternatively, you can run a local server of the web service.
- 3. Building the Distributed Application:
 - Define the functionality of your distributed application. In this case, you'll create an application that consumes the calculator API endpoints.
 - Implement the logic to make HTTP requests to the web service endpoints. You can use libraries like axios in JavaScript or requests in Python to send HTTP requests.
 - Parse the responses from the web service and handle any errors that may occur.
- 4. Test and Run the Distributed Application:
 - Set up the development environment for your distributed application.
 - Run the distributed application and ensure it consumes the web service correctly. □
Debug and fix any issues that may arise.

Conclusion:

We learnt:

- how the web service works,
- how to use web service in a distributed application,
- implementation of web service and distributed application.

Code

1. app.py

```

1. from flask import Flask, request, jsonify 2.
3. app = Flask(__name__) 4.
5. @app.route('/add', methods=['POST']) 6. def add():
7.     data = request.get_json()
8.     num1 = data['num1']
9.     num2 = data['num2']
10.    num3 = num1 + num2
11.    return jsonify({"result": num3}) 12.
13. @app.route('/multiply', methods=["POST"]) 14. def multiply():
15.    data = request.get_json()
16.    num1 = data['num1']
17.    num2 = data['num2']
18.    num3 = num1 * num2
19.    return jsonify({"result": num3}) 20.
21. if __name__ == '__main__': 22.
app.run(debug=True) 23.

```

2. client.py

```

1. import requests 2.
3. url = 'http://127.0.0.1:5000/' 4.
5. def add_num(num1, num2):
6.     endpoint = url + '/add'
7.     data = {"num1": num1, "num2": num2}
8.     response = requests.post(endpoint, json=data)
9.     result = response.json()['result'] 10. return result 11.
12. def multiply_num(num1, num2):
13.     endpoint = url + '/multiply'
14.     data = {"num1": num1, "num2": num2}
15.     response = requests.post(endpoint, json=data)
16.     result = response.json()["result"] 17. return result 18.
19. state = True
20. while(state):
21.     try:
22.         print("Enter the first number:")
23.         num1 = int(input())
24.         print("Enter the second number:") 25.         num2 = int(input()) 26.
27.         print("Do you want \n1. Add \n2. Multiply \n3. Exit") 28.         choice =
int(input("")) 29.         if choice==1):
30.             print(add_num(num1, num2))
31.             print("Do you wish to continue? (Yes, No)") 32.             if (input().lower()=="no"): 33.
state=False 34.
35.         elif(choice==2):
36.             print(multiply_num(num1, num2))
37.             print("Do you wish to continue? (Yes, No)") 38.             if (input().lower()=="no"):

```

```

39.         state=False 40.
elif(choice==3):
41.     print("Thank you for using the service") 42.         state=False
43.     else:
44.         print("Invalid Input") 45.         if(state):
46.             print("New Calculation") 47.
print("_"*10,end="\n") 48.     except:
49.         print("Encountered Error")
50.         print("Restarting interface", end="\n") 51.

```

Instructions:

1. Install Flask.

```
1. pip install flask 2.
```

2. Develop the app.py, and run it by following command.

```
1. python app.py 2.
```

This would start the app server on your local machine.

3. Develop the client.py for interaction with the web service.

4. Run the client.py

```
1. python client.py 2.
```

Output

1. Starting the server

```

* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with watchdog (windowsapi)
* Debugger is active!
* Debugger PIN: 289-260-437
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

```

2. Requesting the Web Service

```
PS D:\College\BE\Sem 8\DS\Lab\DS-LAB\Code\Implementation\Assignment No. 7> python .\client.py
Enter the first number:
2
Enter the second number:
3
Do you want
1.Add
2. Multiply
3. Exit
1
5
Do you wish to continue? (Yes, No)
yes
New Calculation
-----
Enter the first number:
3
Enter the second number:
4
Do you want
1.Add
2. Multiply
3. Exit
2
12
Do you wish to continue? (Yes, No)
no
-----
PS D:\College\BE\Sem 8\DS\Lab\DS-LAB\Code\Implementation\Assignment No. 7> █
```