ASSIGNMENT NO. :01

AIM:

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

PRE-REQUISITE:

1. Knowledge of Multi-threading.
2. Knowledge of Client Server Programming.
3. Knowledge of RMI.

OBJECTIVE:

1. To Remote Method Invocation works.
2. To RMI allows objects to invoke methods on remote objects.
3. To write Distributed Object Application.

THEORY:

What is thread?

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a schedule which is typically a part of the operating systems. The implementation of threads and processes differs between operating systems. A process consists of an execution environment together with one or more threads. A threads is the operating system abstraction of an activity the term derives from the phrase 'thread of execution'. An execution environment is the unit of resource management: a collection of local kernel managed resources to which its threads have access. An execution environment primarily consists of:

- an address space;
- thread synchronization and communication resources such as semaphores and communication interfaces for example sockets);
- higher-level resources such as open files and windows.

Threads can be created and destroyed dynamically is needed. The central aim of having multiple threads of execution between operations thus enabling the overlap of computation with input and output and enabling concurrent processing on multiprocessors. This can be particularly helpful within severs where concurrent processing of client's requests can reduce the tendency for servers to become bottlenecks.

Multithreading in the Server:

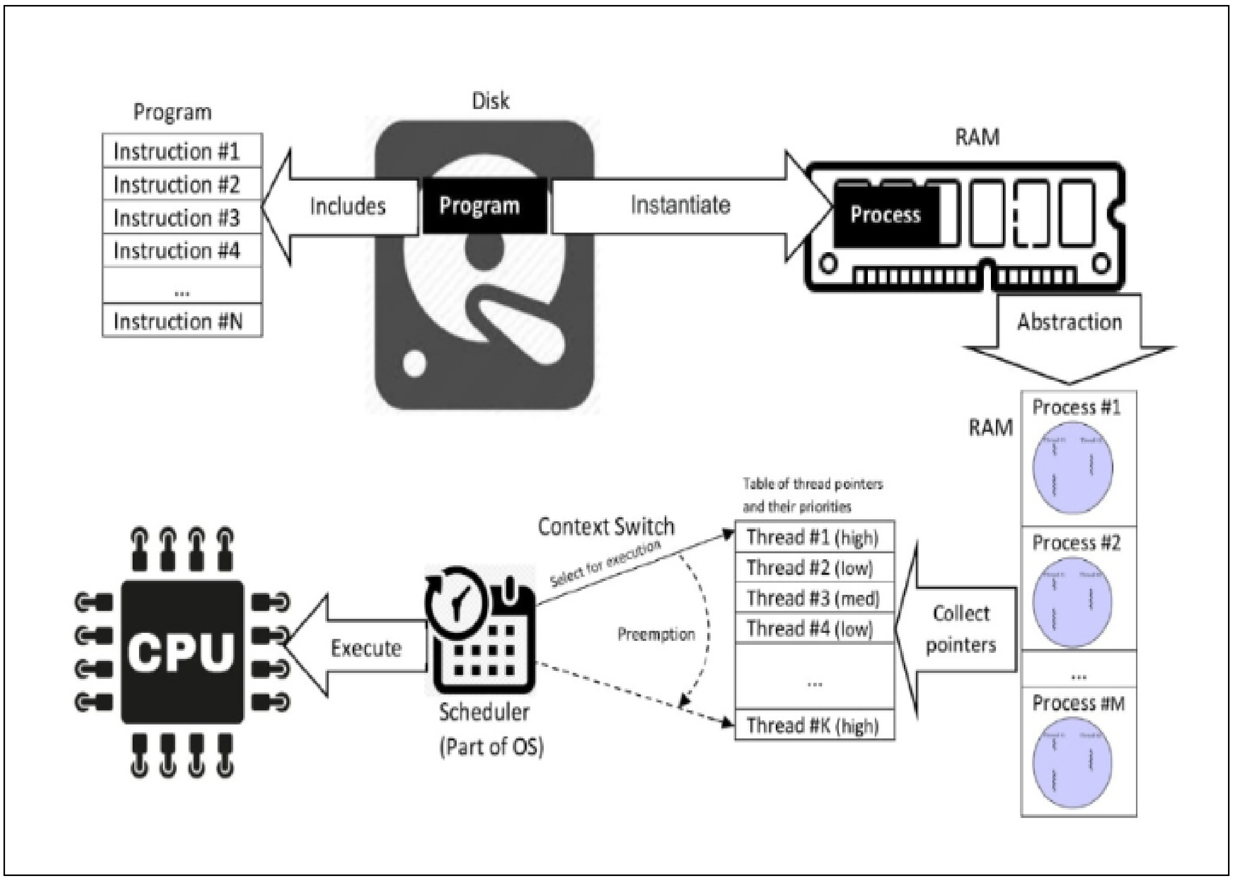
RMI automatically will execute each client in a separate thread. There will only be one instance of your server object no matter how many clients, so you need to make sure that shared data is synchronized appropriately. In particular, any data that may be accessed by more than one client at a time must be synchronized.

Also note that you cannot depend on data in the server to remain the same between two successive calls from a client. In our rental car example from the first class, a client may geta list of cars in one call, and in a later call try to reserve the car. In the meantime, another client may have already reserved that car. The server must double-check all data coming from the client to protect against this sort of error. Also for this reason, the server should never pass to the client any direct pointers to its internal data structures (such as an array index), as that type of data is highly likely to change before the client tries to use it.

Multithreading can also introduce very difficult to find bugs into your program. The types of bugs introduced because of multithreading are called "race conditions". A race condition isa bug that is timing sensitive. In other words, the bug only happens when several conditions happen at exactly the same time. With multithreading, the possibility of race conditions increases.

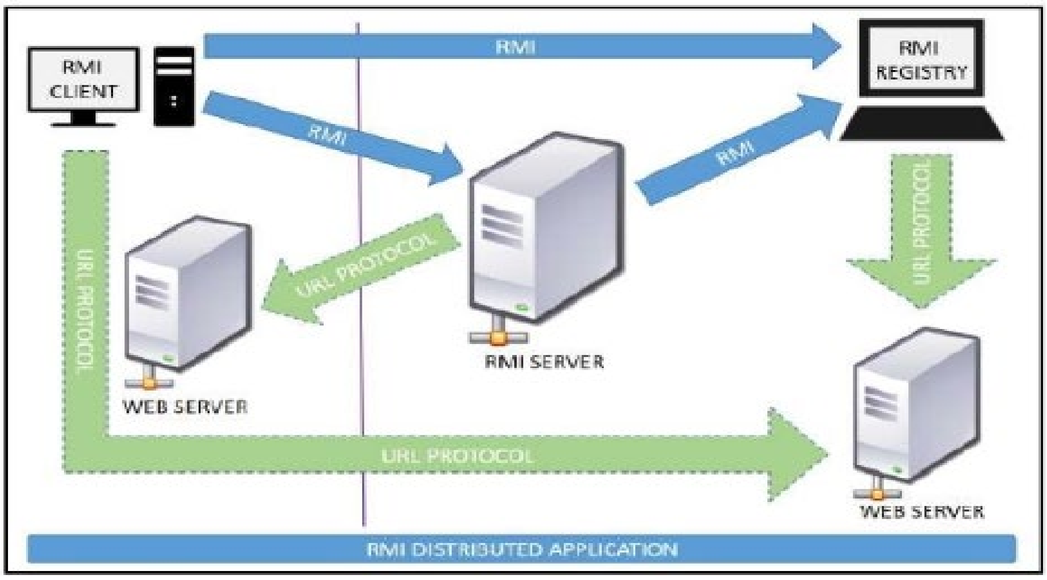
Unfortunately, it's almost impossible to know if a multithreading program has bugs or not. Each thread runs at the same time as the other threads. A computer with a single CPU, however, cannot really run multiple instructions at the same time. So it runs instructions from one thread for a while, and then runs some instructions from another thread. You have no way of knowing exactly when a thread may be interrupted and another thread run.

Structure of Multithreading in the Server:



What is RMI?

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



RMI Registry :

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

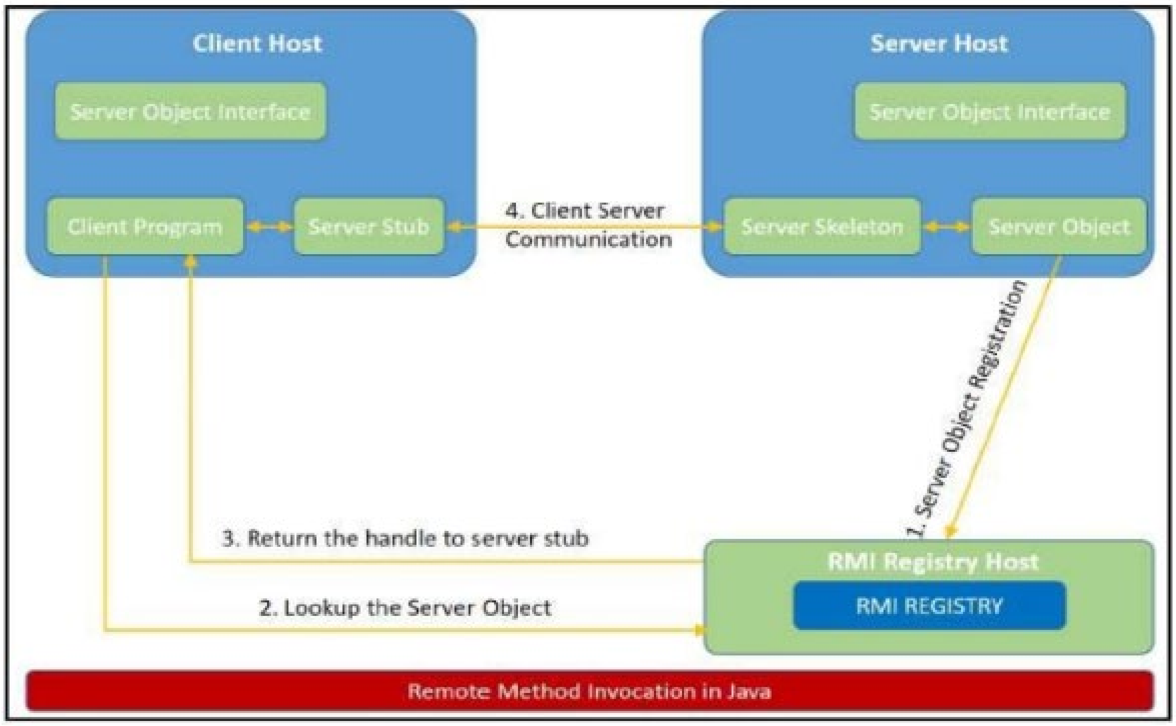
Key Terminologies of RMI:

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

- **Remote object**:
  This is an object in a specific JVM whose methods are exposed so they could be invoked by another program deployed on a different JVM.
- **Remote interface:**
  This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.
- **RMI:**

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

- **Stub:**
  This is a Java object that acts as an entry point for the client object to route any outgoing requests. It exists on the client JVM and represents the handle to the remote object.
- If any object invokes a method on the stub object, the stub establishes RMI by following these steps:
  1. It initiates a connection to the remote machine JVM.
  2. It marshals (write and transmit) the parameters passed to it via the remote JVM.
  3. It waits for a response from the remote object and unmarshals (read) the returned value or exception, then it responds to the caller with that value or exception.
- **Skeleton:**
  This is an object that behaves like a gateway on the server side. It acts as a remote object with which the client objects interact through the stub. This means that any requests coming from the remote client are routed through it. If the skeleton receives a request, it establishes RMI through these steps:
  1. It reads the parameter sent to the remote method.
  2. It invokes the actual remote object method.
  3. It marshals (writes and transmits) the result back to the caller (stub).



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

The Above diagram demonstrates RMI communication with stub and skeleton involved.

Source Code for Add Client

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

RMI Implemenation Steps:

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

Create all java files and compile using **javac** command , it will generate **.class** files.

Step 2: Generate a Stub

Generate stubs invoking **rmic AddServerImpl** it will generate **AddServerImpl_Stub.class** file.

Step 3: Install Files on the Client and Server Machines

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

Step 4: Add Directroy on the Server Machine

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

Step 5: Start the RMI Registry on the Server Machine

Java provides a program called rmiregistry, which executes on the server machine.

Step 6: Start the Server using java AddServer in new terminal

Step 7: Start the Client java AddClient servername/ip_address 8 9 in new terminal where servername is first argument and 8 , 9 are second & third argument respectively.

CONCLUSION:

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

OUTPUT:

ASSIGNMENT NO. :02

AIM:

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

PRE-REQUISITE:

1. Knowledge of CORBA and its Architecture.
2. Knowledge of the Object Request Broker (ORB) and its role in CORBA.
3. Knowledge of how to create and run Java applications using command line.

OBJECTIVE:

1. To CORBA can be used as a middleware to allow objects running on different machines to communicate and interact with each other.
2. To Client can use CORBA to RMI on server objects that perform String Operations such as concatenation and comparison.
3. To using CORBA allows these operations to be performed across a distributed system transparently.

THEORY:

What is CORBA?

CORBA (Common Object Request Broker Architecture):-

CORBA (Common Object Request Broker Architecture) is a middleware technology that enables distributed computing between different platforms and programming languages. It provides a standard protocol for communication between objects, and a mechanism for locating and invoking remote objects in a network. In this lab, we will use Java and CORBA to develop a distributed application that demonstrates object brokering with string operations.

The Common Object Request Broker Architecture (CORBA) is a first step by the Object Management Group towards achieving application portability and interoperability across heterogeneous computing platforms. It is a standard for the development and deployment of applications in distributed, heterogeneous environments. CORBA 1.1 was first introduced in 1991. In CORBA 2.0, adopted in December 1994, true interoperability is defined by specifying how ORBs from different vendors can interoperate.

A client makes a request for a service provided by an object implementation through the Object Request Broker (ORB). The client is not required to care about the location of the object implementation, its programming language, or any other details. To support portability and interoperability, the ORB is responsible for all of the mechanisms required to locate the object implementation, and to prepare the object implementation to receive the request and its data. Figure 4 shows the main components of the ORB architecture and their interconnections.

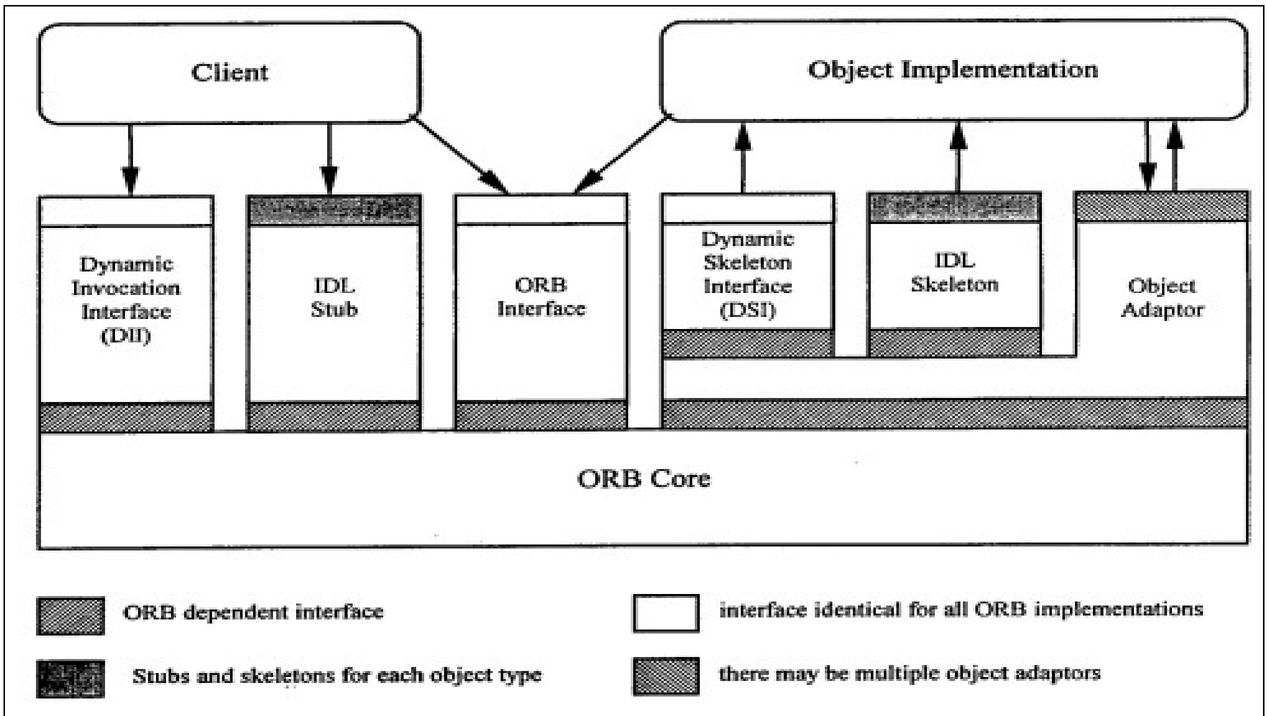

Client
Object Implementation
Dynamic Invocation Interface (DII)
IDL Stub
ORB Interface
Dynamic Skeleton Interface (DSI)
IDL Skeleton
Object Adaptor
ORB Core
ORB dependent interface
interface identical for all ORB implementations
Stubs and skeletons for each object type
there may be multiple object adaptors

Figure 4 : The Structure of Object Request Broker Interfaces

The client makes a request using either the Dynamic Invocation Interface (DII) or an OMG IDL stub. The client can also directly interact with the ORB through the ORB Interface for some functions. Similarly, the object implementation receives a request either through the OMG IDL skeleton or through a Dynamic Skeleton Interface (DSI). The object implementation may call the Object Adaptor and the ORB Interface for the ORB services.

Having access to the object reference as a handle to the object implementation, the client initiates the request by statically calling the IDL stub, or by constructing the request dynamically through the Interface Repository. An OMG IDL stub is the specific stub depending on the interface of the target object, whereas the DII interface is independent of the interface of the target object, which therefore can be used even without thorough knowledge of that interface. Nevertheless, both the static and dynamic invocation methods generate the same request semantics to the object implementation.

The ORB locates the target object, transmits parameters and transfers control to the object implementation through an IDL skeleton or a dynamic skeleton. Defining the interface of an object in IDL generates an IDL skeleton that is specific to the interface and the object adaptor. The object implementation information provided at installation time is stored in the Implementation Repository, which is available for use during request delivery.

Figure 5 shows how interface and implementation information is made available to clients and object implementations. The interface is defined in OMG IDL and/or in the Interface Repository. The client stubs and the object implementation are generated from the IDL definition of the interface.

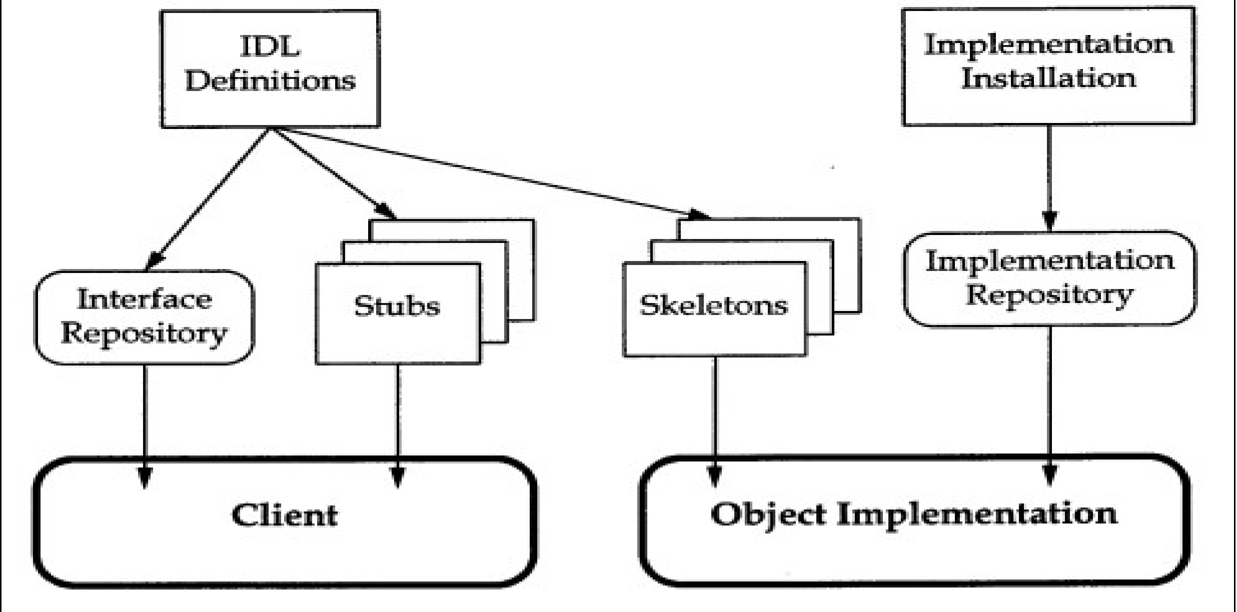


Figure 5 : Interface and Implementation Repositories

Source code

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

class ReverseClient
{

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

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


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

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

            String name = "Reverse";

           // narrow converts generic object into string type
            ReverseImpl = ReverseHelper.narrow(ncRef.resolve_str(name));

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

            String tempStr= ReverseImpl.reverse_string(str);

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

String Reverse using corba , idl and java implementation Steps:

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

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

```
idlj -fall ReverseModule.idl
```

The **idlj** compiler generates a number of files.

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

```
javac *.java ReverseModule/*.java
```

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

```
orbd -ORBInitialPort 1050&
```

Step 5: Start the server. To start the server from a UNIX command shell, enter:
```
java ReverseServer -ORBInitialPort 1050& -ORBInitialHost localhost&
```

Step 6: Run the client application :
```
java ReverseClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

CONCLUSION:

we have successfully demonstrated the concept of object brokering using CORBA to create a distributed application for performing string operations. We have implemented a server-side code that registers a string helper object with the naming service, and a client-side code that accesses the string helper object to perform operations on a string. The string helper object is a remote object that resides on the server, and its methods are invoked by the client using the ORB. By using CORBA, we have achieved the interoperability between different programming languages and platforms, and have shown how object brokering can be used to create distributed applications that can communicate seamlessly across different systems.

OUTPUT:

ASSIGNMENT NO. :03

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 Open MP. Demonstrate by displaying the intermediate sums calculated at different processors.

PRE-REQUISITE:

1. Knowledge of MPI
2. Knowledge about its role.
3. Knowledge of how to create and run Java applications using command line.

OBJECTIVE:

1. To Develop a widely used standard for Writing Message Passing Program
2. Design an application program interface.

THEORY:

What is MPI?

MPI (Message Passing Interface):-

Message Passing Interface (MPI) is a subroutine or a library for passing messages between processes in a distributed memory model. MPI is not a programming language. MPI is a programming model that is widely used for parallel programming in a cluster. In the cluster, the head node is known as the master, and the other nodes are known as the workers. By using MPI, programmers are able to divide up the task and distribute each task to each worker or to some specific workers. Thus, each node can work on its own task simultaneously.

Since this is a small module, we will be focusing on only important and common MPI functions and techniques. For further study, there are a lot of free resources available on the Internet.

Why MPI?

There are many reasons for using MPI as our parallel programming model:

- MPI is a standard message passing library, and it is supported on all high-performance computer platforms.
- An MPI program is able to run on different platforms that support the MPI standard without changing your source codes.

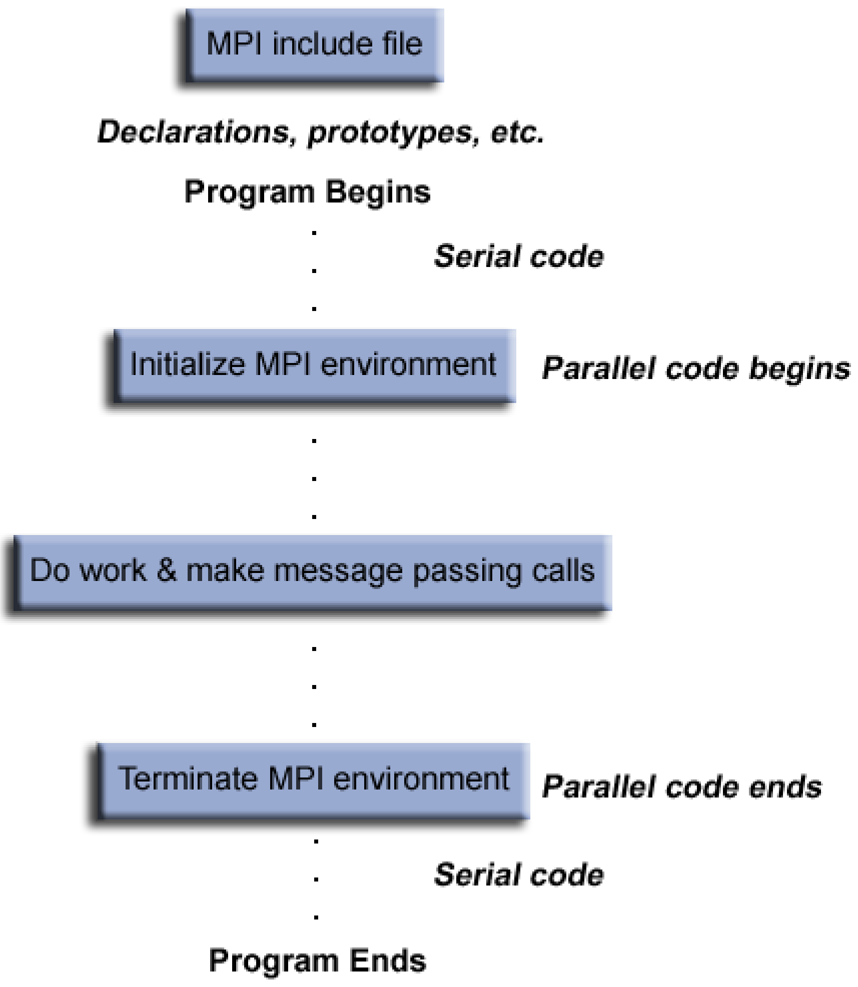
- Because of its parallel features, programmers are able to work on a much larger problem size with the faster computation.
- There are many useful functions available in the MPI Library.
- A variety of implementations are available.

How do I write my MPI Program?

In order to get the MPI library working, you need to include the header file **#include <mpi.h>** or **#include "mpi.h"** in your C code.

MPI Program Structure

Like other programming languages you have seen, program that includes MPI library has its structure. The structure is shown in the figure below:


MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

A MPI program is basically a C program that uses the MPI library, SO DON'T BE SCARED. The program has two different parts, one is serial, and the other is parallel. The serial part contains variable declarations, etc., and the parallel part starts when MPI execution environment has been initialized, and ends when MPI_Finalize() has been called.

Communicator: a set of processes that have a valid rank of source or destination fields. The predefined communicator is MPI_COMM_WORLD, and we will be using this communicator all the time in this module. MPI_COMM_WORLD is a default communicator consisting all processes. Furthermore, a programmer can also define a new communicator, which has a smaller number of processes than MPI_COMM_WORLD does.

Some Common Function

The following functions are the functions that are commonly used in MPI programs:

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

This function has to be called in every MPI program. It is used to initialize the MPI execution environment.

```
MPI_Comm_size(comm, &size)
```

This function determines the number of processes in the communicator. The number of processes get store in the variable *size*. All processes in a communicator have the same value of size.

```
MPI_Comm_rank(comm, &rank)
```

This function determines the rank of the calling process within the communicator. Each process is assigned uniquely by integer rank from 0 to number of processes - 1, and its rank gets stored in the variable rank.

```
MPI_Get_processor_name(name, &len)
```

This function returns the unique processor name. Variable name is the array of char for storing the name, and len is the length of the name.

```
MPI_Wtime()
```

This function returns an elapsed wall clock time in seconds on the calling processor. This function is often used to measure the running time of an MPI program. There is no defined starting point; therefore, in order to measure the running time, a programmer needs to call two different MPI_Wtime(), and find the difference.

MPI_Finalize()

This function terminates the MPI execution environment. MPI_Finalize() has to be called by all processes before exiting.

Implementing the solution:

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

express library.

a. Download MPJ Express (mpj.jar) and unpack it.

b. Set MPJ_HOME and PATH environment variables:

c. export MPJ_HOME=/path/to/mpj/

d. export PATH=$MPJ_HOME/bin:$PATH

2. Write Hello World parallel Java program and save it as HelloWorld.java (Asign2.java).

3. Compile a simple Hello World (Asign) parallel Java program

4. Running MPJ Express in the Multi-core Configuration.

Compiling and Executing the solution:

Compile: javac -cp $MPJ_HOME/lib/mpj.jar Asign2.java

(mpj.jar is inside lib folder in the downloaded MPJ Express)

CONCLUSION:

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

OUTPUT:

ASSIGNMENT NO. :04

AIM:

Implement Berkeley algorithm for clock synchronization.

PRE-REQUISITE:

1. To perform this lab, you should have a basic understanding of Javan programming language and socket programming.
2. You should also have access to a computer running a Unix-based operating system (such as Linux or macOS) that can run Python code.
3. In addition, you should have a basic understanding of distributed systems and clock synchronization algorithms.
4. It would also be helpful to have some experience working with simulated distributed systems.

OBJECTIVE:

1. To understand the basic principles behind the Berkeley algorithm for clock synchronization.
2. To implement the Berkeley algorithm in Python.
3. To test the implementation of the Berkeley algorithm in a simulated distributed system.
4. To analyze the results of the simulation and evaluate the performance of the algorithm.

THEORY:

What is Berkeley Algorithm?

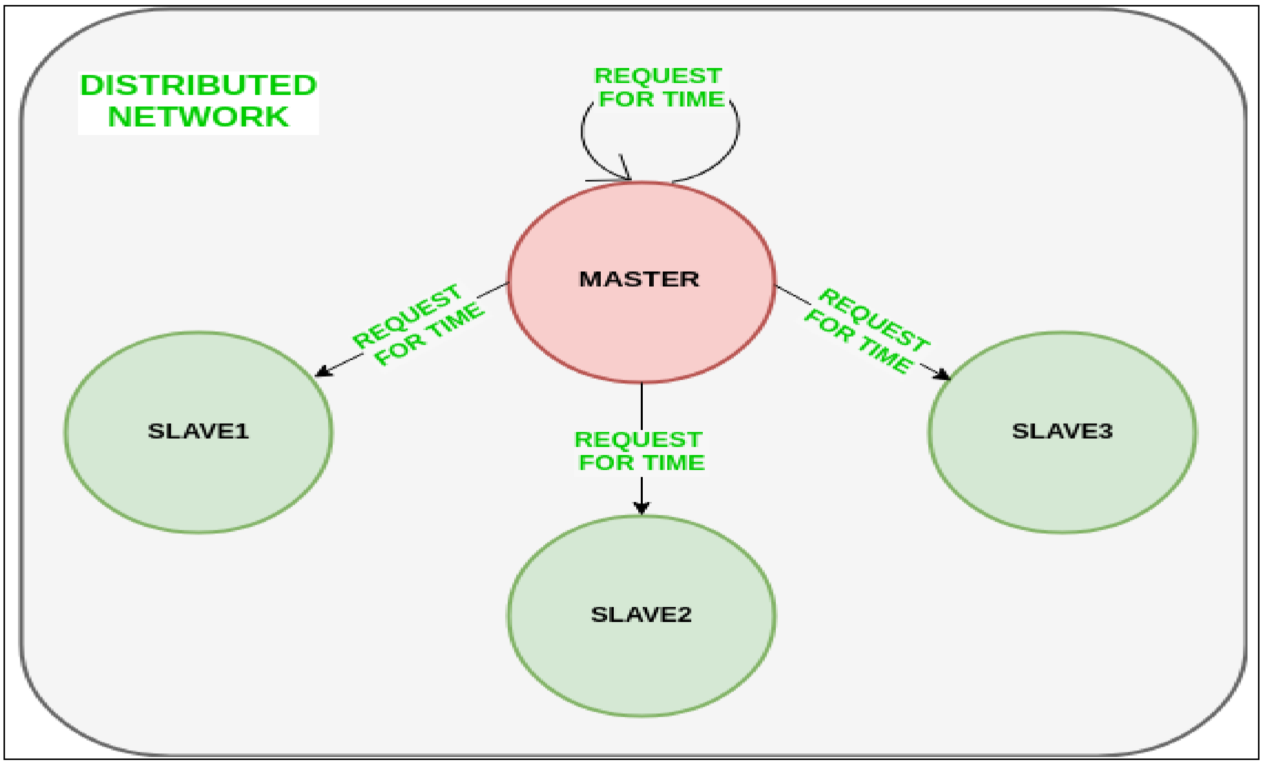
In a distributed system, it is important to have all the clocks on different computers synchronized to the same time. This is because many distributed applications require a common notion of time in order to function correctly.

The Berkeley algorithm for clock synchronization is a widely used algorithm for synchronizing the clocks of computers in a distributed system. The basic idea behind the algorithm is to periodically synchronize the clocks of all the computers in the system by adjusting the clock of each computer based on the average of the clocks of all the other computers.

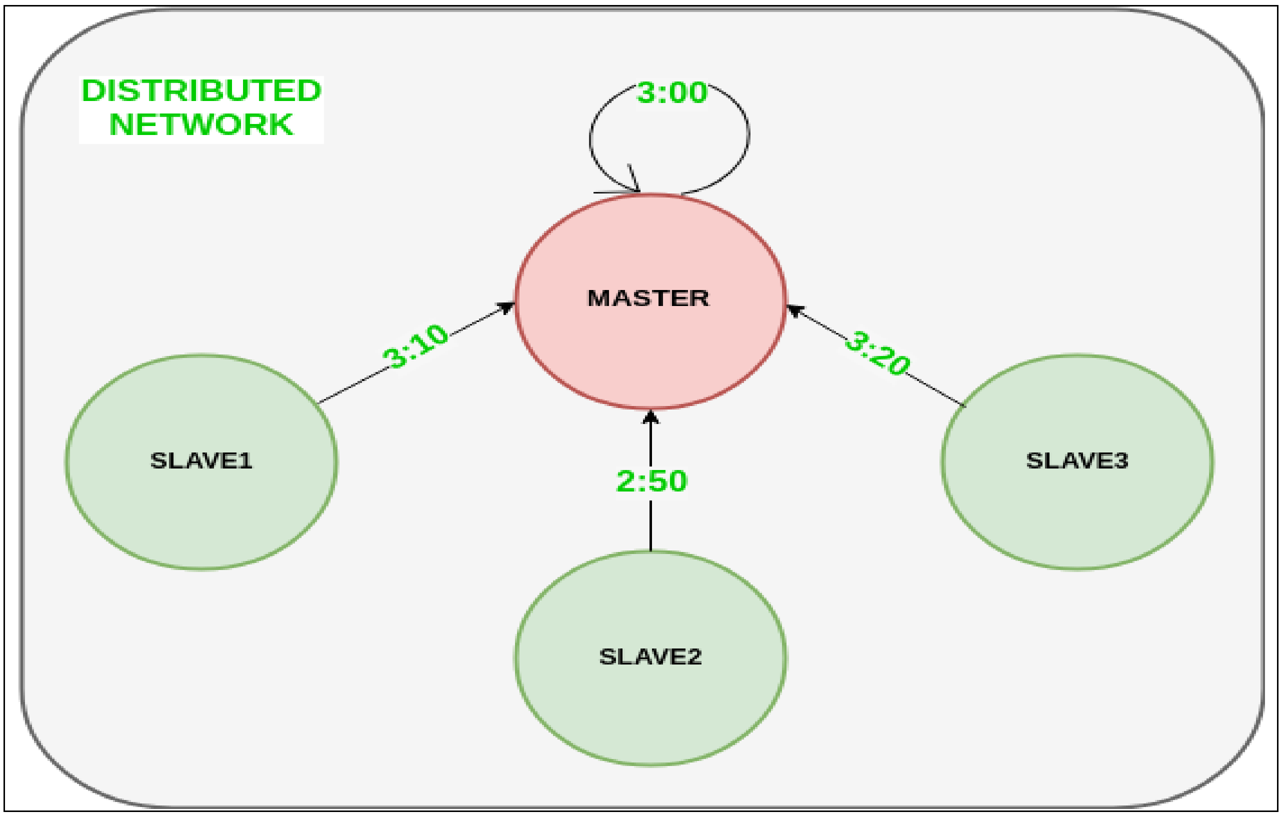
Algorithm:

1) An individual node is chosen as the master node from a pool node in the network. This node is the main node in the network which acts as a master and the rest of the nodes act as slaves. The master node is chosen using an election process/leader election algorithm.
2) Master node periodically pings slaves nodes and fetches clock time at them using Cristian's algorithm.

The diagram below illustrates how the master sends requests to slave nodes.

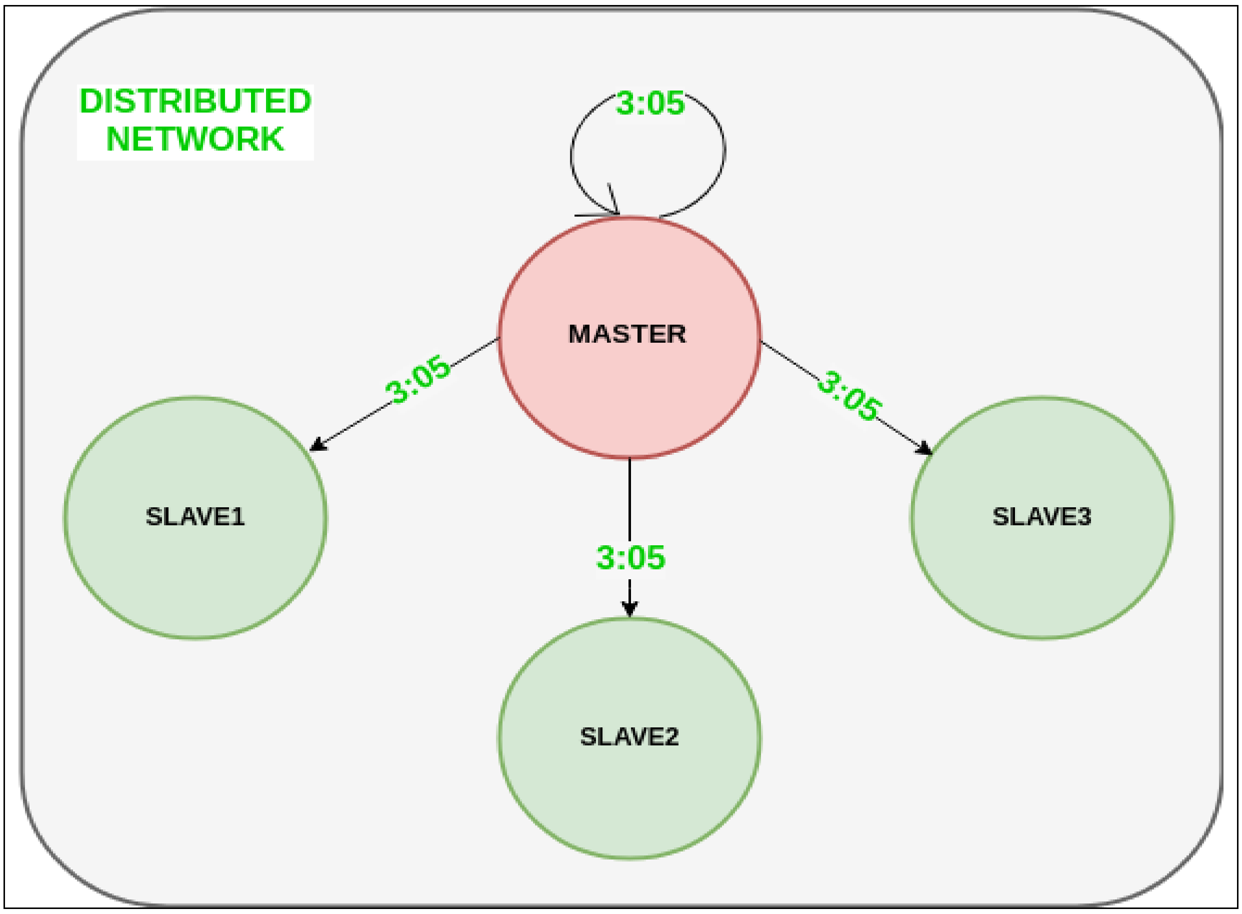


The diagram below illustrates how slave nodes send back time given by their system clock.



3) Master node calculates the average time difference between all the clock times received and the clock time given by the master's system clock itself. This average time difference is added to the current time at the master's system clock and broadcasted over the network.

The diagram below illustrates the last step of Berkeley's algorithm.



Scope of Improvement

• Improvision inaccuracy of Cristian's algorithm.
• Ignoring significant outliers in the calculation of average time difference
• In case the master node fails/corrupts, a secondary leader must be ready/pre-chosen to take the place of the master node to reduce downtime caused due to the master's unavailability.
• Instead of sending the synchronized time, master broadcasts relative inverse time difference, which leads to a decrease in latency induced by traversal time in the network while the time of calculation at slave node.

Features of Berkeley's Algorithm:
Centralized time coordinator: Berkeley's Algorithm uses a centralized time coordinator, which is responsible for maintaining the global time and distributing it to all the client machines.

Clock adjustment: The algorithm adjusts the clock of each client machine based on the difference between its local time and the time received from the time coordinator.
Average calculation: The algorithm calculates the average time difference between the client machines and the time coordinator to reduce the effect of any clock drift.
Fault tolerance: Berkeley's Algorithm is fault-tolerant, as it can handle failures in the network or the time coordinator by using backup time coordinators.
Accuracy: The algorithm provides accurate time synchronization across all the client machines, reducing the chances of errors due to time discrepancies.
Scalability: The algorithm is scalable, as it can handle a large number of client machines, and the time coordinator can be easily replicated to provide high availability.
Security: Berkeley's Algorithm provides security mechanisms such as authentication and encryption to protect the time information from unauthorized access or tampering.

Source Code

```
import java.io.*;
import java.net.*;
import java.util.*;

public class BerkeleyAlgorithm {

    // Define the port number that will be used for communication
    private static final int PORT = 1024;

    public static void main(String[] args) throws Exception {

        // Create a server socket to listen for incoming messages
        ServerSocket serverSocket = new ServerSocket(PORT);

        // Create a list to store the time differences for each node
        List<Long> timeDiffs = new ArrayList<Long>();

        // Create a new thread to handle the time requests from nodes
        Thread timeServerThread = new Thread(new Runnable() {
            public void run() {
                while (true) {
                    try {
                        // Wait for a node to connect and request the current time
                        Socket clientSocket = serverSocket.accept();
                        ObjectInputStream in = new
ObjectInputStream(clientSocket.getInputStream());

                        // Read the current time from the node's request
```

```
Date clientTime = (Date) in.readObject();
```

Berkeley algorithm for clock synchronization implementation Steps:

Step 1: Establish communication:

Create a server socket to listen for incoming time requests from nodes.
Each node periodically sends time requests to the server.

Step 2: Record time differences:

When a node sends a time request to the server, record the time difference between the node's local time and the time received from the server.
When the server receives a time request from a node, respond with the current time.

Step 3: Compute the average time difference:

Wait for a sufficient number of time differences to be recorded.
Compute the average time difference by summing up all the time differences and dividing by the number of recorded differences.

Step 4: Adjust the node's clock:

Adjust the node's clock by adding the average time difference to the local time.
Output the adjusted time.

Step 5: Repeat:

Periodically repeat steps 2-4 to maintain synchronization between the node and the server.

CONCLUSION:

In conclusion, the Berkeley algorithm is a simple and effective way to synchronize the clocks of multiple nodes in a network. By measuring the time differences between nodes and a central time server, the algorithm can compute an average time difference that can be used to adjust each node's clock to be synchronized with the others.

OUTPUT:

ASSIGNMENT NO. :05

AIM:

Implement token ring based mutual exclusion algorithm.

PRE-REQUISITE:

1. To A programming language of your choice (Python, Java, C++, etc.)
2. An understanding of socket programming and interprocess communication
3. A development environment (e.g., an IDE such as PyCharm or Visual Studio Code)

OBJECTIVE:

1. To Understand the basics of the token ring-based mutual exclusion algorithm.
2. Gain experience with socket programming and interprocess communication.
3. Learn how to implement a distributed algorithm in a programming language of your choice.
4. Practice debugging and troubleshooting skills.
5. Learn how to test a distributed algorithm to verify correctness.

THEORY:

What is Ring Based Mutual exclusion Algorithm?

A token circulates around the ring from process to process.
A process can access the shared resource only when it receives the token.
When a process finishes accessing the shared resource, it passes the token to the next process in the ring.
If a process wants to access the shared resource but does not have the token, it sends a request message to the next process in the ring.
The process that receives the request message sets a flag to indicate that it has received a request and passes the token to the next process in the ring.
When the process that has the token receives the request message flag, it does not release the token after accessing the shared resource, but instead passes the token to the process that requested it.

Source Code

```
import java.io.*;
import java.util.*;
```

```
class Tok {

    public static void main(String args[]) throws Throwable {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the num of nodes:");
        int n = scan.nextInt();
        int m = n - 1;
        // Decides the number of nodes forming the ring
        int token = 0;
        int ch = 0, flag = 0;
        for (int i = 0; i < n; i++) {
            System.out.print(" " + i);
        }
        System.out.println(" " + 0);
        do{
            System.out.println("Enter sender:");
            int s = scan.nextInt();
            System.out.println("Enter receiver:");
            int r = scan.nextInt();
            System.out.println("Enter Data:");
            int a;
            a = scan.nextInt();
            System.out.print("Token passing:");
            for (int i = token, j = token; (i % n) != s; i++, j = (j + 1) % n) {
                System.out.print(" " + j + "->");
            }
```

CONCLUSION:

In conclusion, this lab provides an opportunity to learn and implement a token ring-based mutual exclusion algorithm. The algorithm ensures that only one process can access a shared resource at a time in a distributed system. By completing this lab, you will gain experience with socket programming, interprocess communication, and implementing a distributed algorithm in a programming language of your choice.

OUTPUT:

ASSIGNMENT NO. :06

AIM:

Implement Bully and Ring algorithm for leader election.

PRE-REQUISITE:

1. Basic knowledge of distributed systems.
2. Familiarity with programming language Java.
3. Knowledge of communication protocols like TCP or UDP.
4. A computer or a set of computers connected in a network.

OBJECTIVE:

1. To gain a better understanding of distributed systems and leader election algorithms.
2. To be able to implement and test the Bully and Ring algorithms in a simulated distributed environment.
3. To compare and contrast the performance and effectiveness of the Bully and Ring algorithms in leader election scenarios.
4. To identify the strengths and weaknesses of the Bully and Ring algorithms in terms of scalability, fault tolerance, and resource utilization.

THEORY:

Introduction:

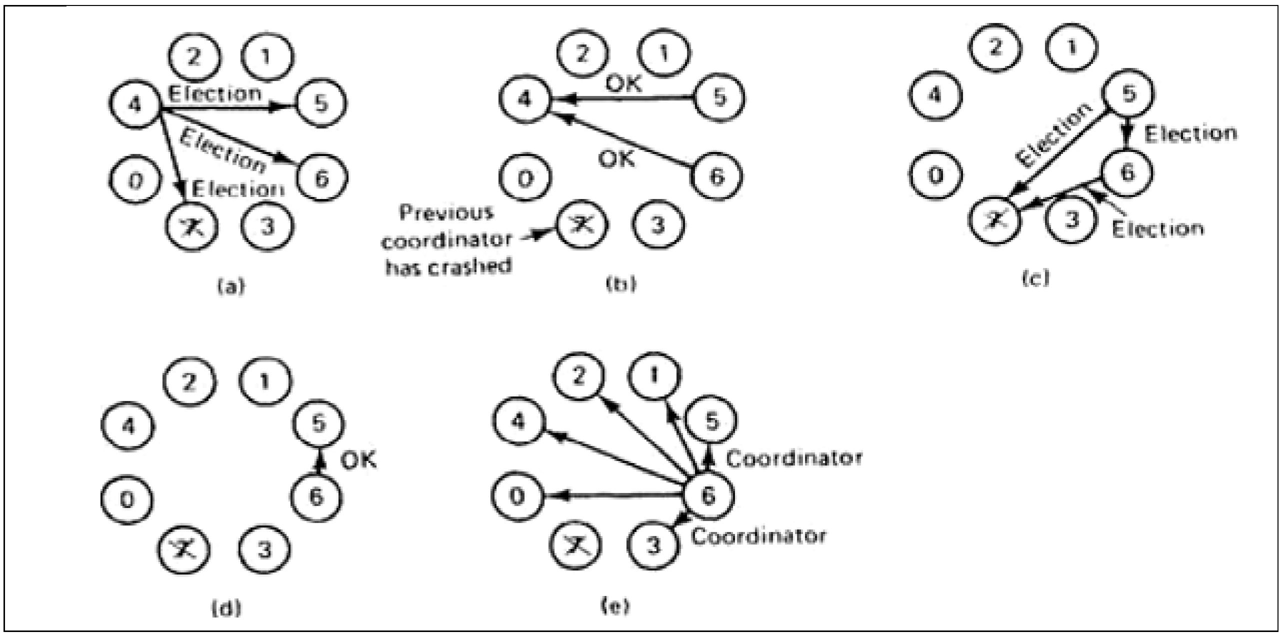
Leader Election is an important problem in distributed systems where a group of processes or nodes are running independently and want to choose a leader among themselves.

What is Bully Algorithm?

The Bully Algorithm is a centralized algorithm, where the process with the highest process ID is chosen as the leader. If the current leader fails, then the process with the next highest ID takes over. The algorithm works as follows:

1. When a process detects that the current leader has failed, it sends an election message to all processes with a higher ID number.
2. If a process receives an election message, it sends an "OK" message back to the sender, indicating that it is still alive.

3. If a process does not receive an "OK" message after sending an election message, it assumes that the sender has failed and takes over as the leader.



What is Ring Algorithm?

The Ring Algorithm is a decentralized algorithm, where each process in the network is connected to its immediate neighbours. The process with the lowest process ID is chosen as the leader. The algorithm works as follows:

1. Each process in the network sends a message to its right neighbour to determine if it has a higher ID.
2. If the right neighbour has a higher ID, the process forwards the message to the right neighbour.
3. If the right neighbour has a lower ID or the message has made a full round, the process declares itself as the leader.

Source Code

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

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

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

Implementation Steps:

Step 1: Create project in eclipse

Step 2: Create the class **Ring.java** or **Bully.java** and execute

CONCLUSION:

Leader election is an important problem in distributed systems, and there are several algorithms to solve it. In this lab, we explored two algorithms: the Bully Algorithm and the Ring Algorithm. We implemented the algorithms and tested them by simulating node failures. The Bully Algorithm is a centralized algorithm, while the Ring Algorithm is a decentralized algorithm.

OUTPUT:

ASSIGNMENT NO. :07

AIM:

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

PRE-REQUISITE:

1 Knowledge of distributed application
2 Knowledge about its role.
3 Knowledge of how to create and run Java applications using command line.

OBJECTIVE:

3. To Develop a widely used standard for Web services
4. Design an application program interface.

THEORY:

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

A service can be treated as a web service if:

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

Types of Web Services:

There are two types of web services:

1. SOAP: SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it's XML based, it's platform and language independent. So, our server can be based on JAVA and client can be on .NET, PHP etc. and vice versa.

2. REST: REST (Representational State Transfer) is an architectural style for developing web services. It's getting popularity recently because it has small learning curve when compared to SOAP. Resources are core concepts of Restful web services and they are uniquely identified by their URIs.

Web service architectures:

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

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

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

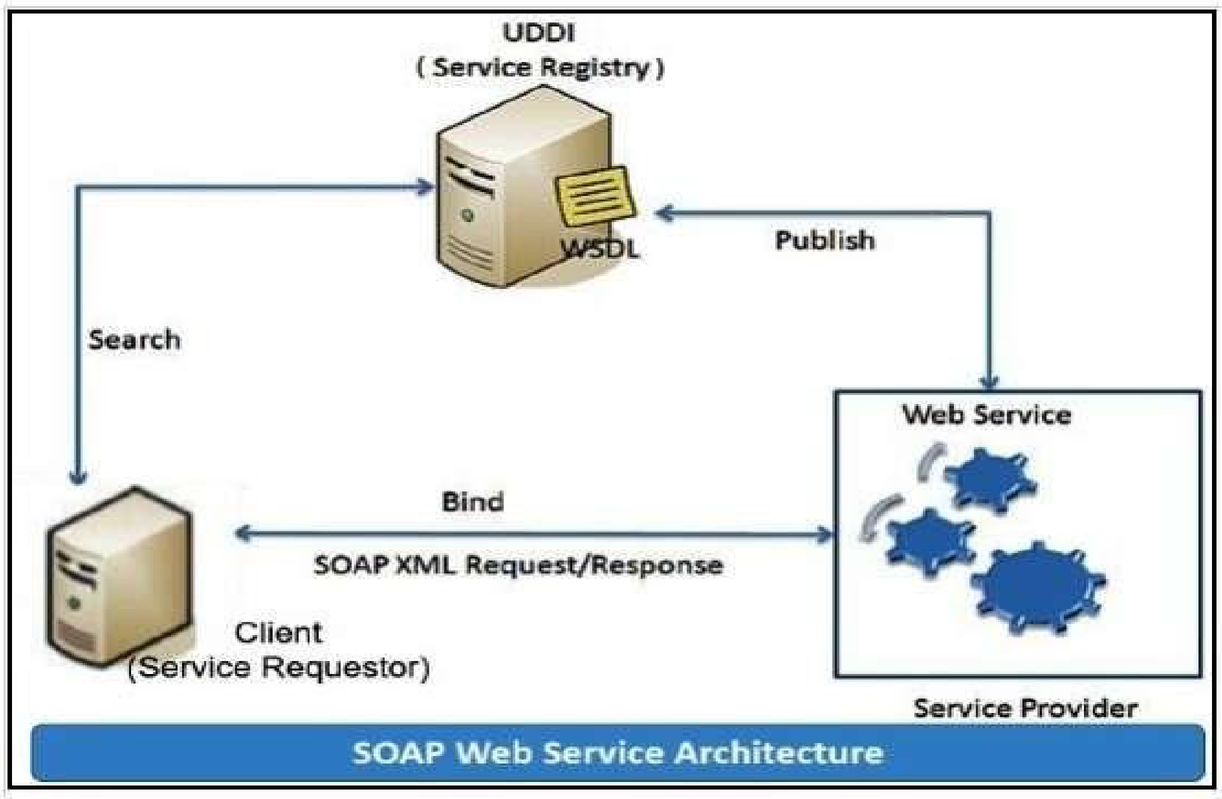
SOAP web services:

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

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

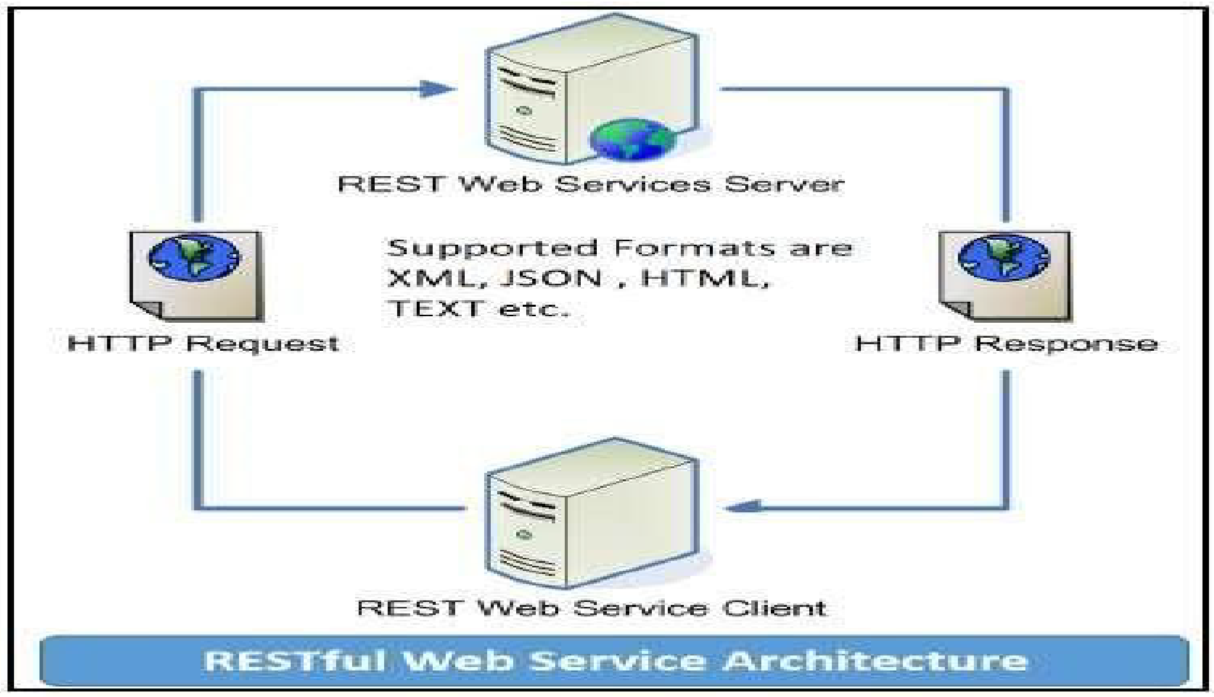
Universal Description, Discovery, and Integration (**UDDI**)*:* UDDI is an XMLbased framework for describing, discovering, and integrating web services. It acts as a directory of web service interfaces described in the WSDL language.

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



RESTful web services

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


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

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

Implementing the solution:

Creating a web service CalculatorWSApplication:

- Create New Project for CalculatorWSApplication.
- Create a package org.calculator
- Create class CalculatorWS.
- Right-click on the CalculatorWSand create New Web Service.

IDE starts the glassfish server, builds the application and deploys the application onserver.

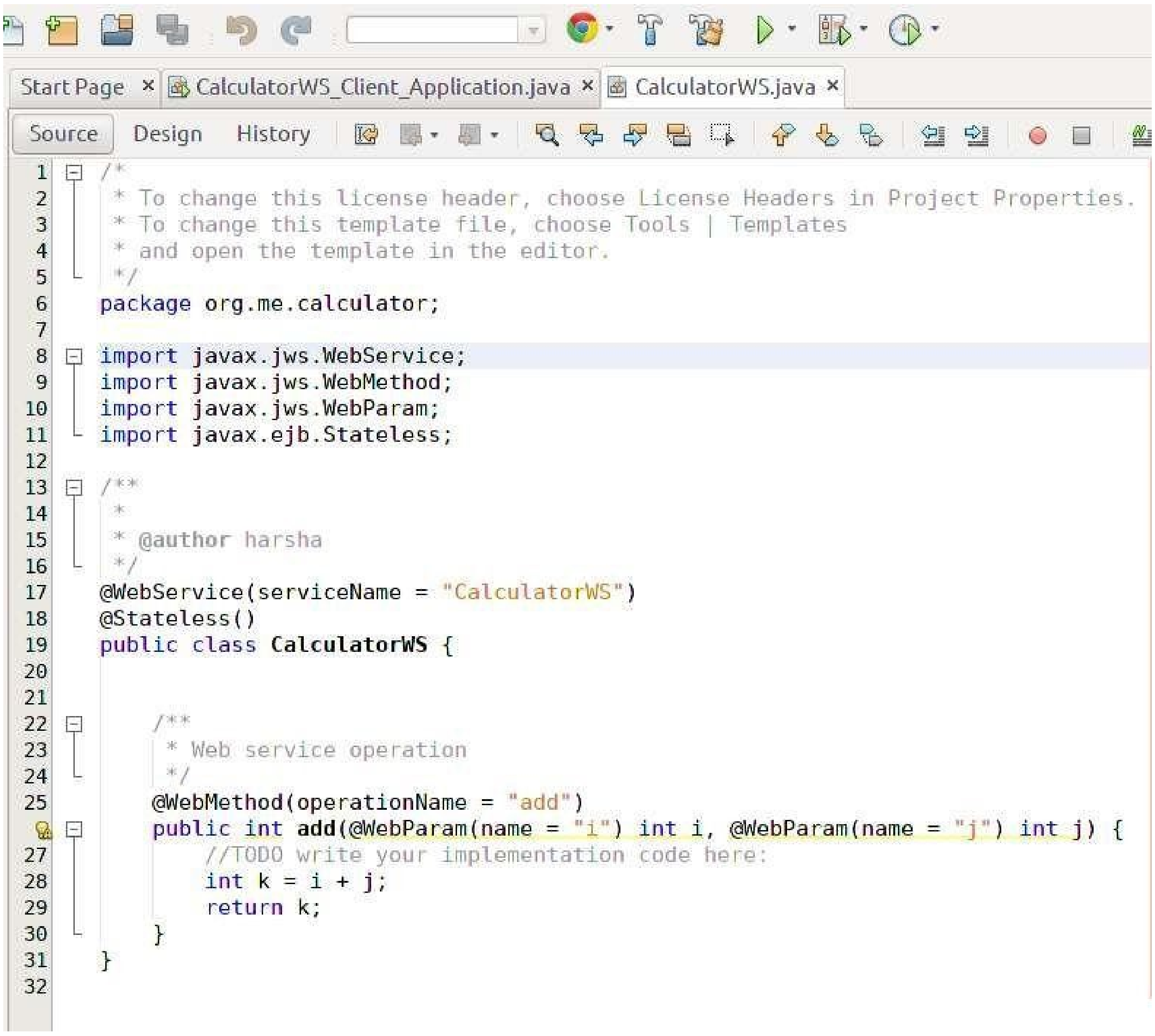
Consuming the Webservice:

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

Creating servlet in web application

- Create new jsp page for creating user interface.

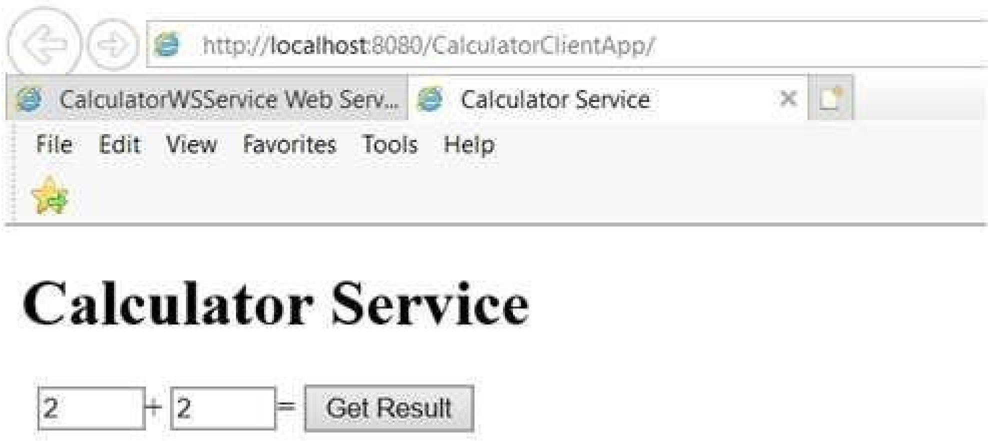
Writing the source code:



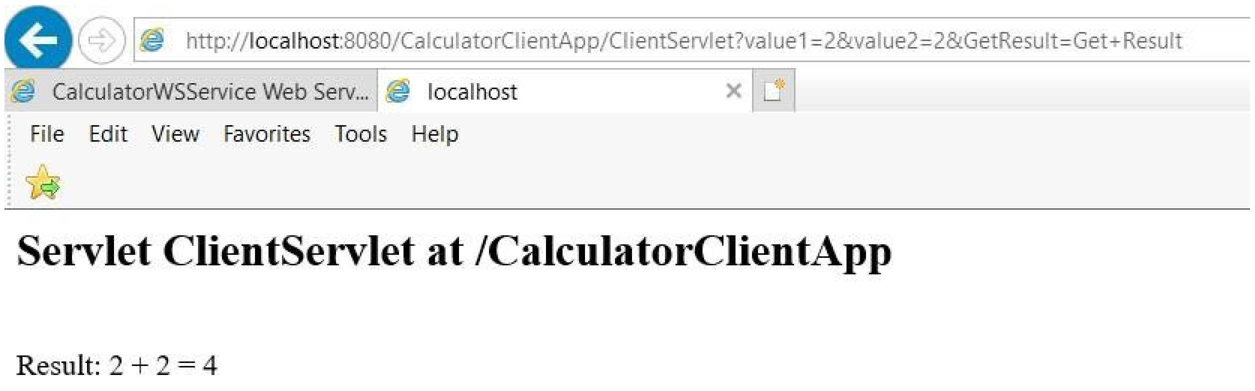
```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package org.me.calculator;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.ejb.Stateless;

/**
 *
 * @author harsha
 */
@WebService(serviceName = "CalculatorWS")
@Stateless()
public class CalculatorWS {


    /**
     * Web service operation
     */
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "i") int i, @WebParam(name = "j") int j) {
        //TODO write your implementation code here:
        int k = i + j;
        return k;
    }
}
```

Compiling and Executing the solution

Right Click on the Project and Choose Run.


http://localhost:8080/CalculatorClientApp/
CalculatorWSService Web Serv...
Calculator Service
File
Edit
View
Favorites
Tools
Help

Calculator Service


http://localhost:8080/CalculatorClientApp/ClientServlet?value1=2&value2=2&GetResult=Get+Result
CalculatorWSService Web Serv...
localhost
File Edit View Favorites Tools Help

Servlet ClientServlet at /CalculatorClientApp

Result: 2 + 2 = 4

CONCLUSION:

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

OUTPUT: