ASSIGNMENT NO: 1

Title: Socket Programming

Aim: To develop any distributed application through implementing client-server communication programs based on Java Sockets.

Objectives:

1. To study inter process communication
2. To learn client server communication using socket

Tools / Environment:

Java Programming Environment, rmi registry, jdk 1.8, Eclipse IDE.

Related Theory:

Socket: In distributed computing, network communication is one of the essential parts of any system, and the socket is the endpoint of every instance of network communication. In Java communication, it is the most critical and basic object involved.

A socket is a handle that a local program can pass to the networking API to connect to another machine. It can be defined as the terminal of a communication link through which two programs /processes/threads running on the network can communicate with each other. The TCP layer can easily identify the application location and access information through the port number assigned to the respective sockets.

During an instance of communication, a client program creates a socket at its end and tries to connect it to the socket on the server. When the connection is made, the server creates a socket at its end and then server and client communication is established.

Designing the solution:

The **java.net** package provides classes to facilitate the functionalities required for networking. The **socket** class programmed through Java using this package has the capacity of being independent of the platform of execution; also, it abstracts the calls specific to the operating system on which it is invoked from other Java interfaces. The **ServerSocket** class offers to observe connection invocations, and it accepts such invocations from different clients through another socket. High-level wrapper classes, such as **URLConnection** and **URLEncoder**, are more appropriate. If you want to establish a connection to the Web using a URL, then these classes will use the socket internally.

The **java.net** package provides support for the two common network protocols −

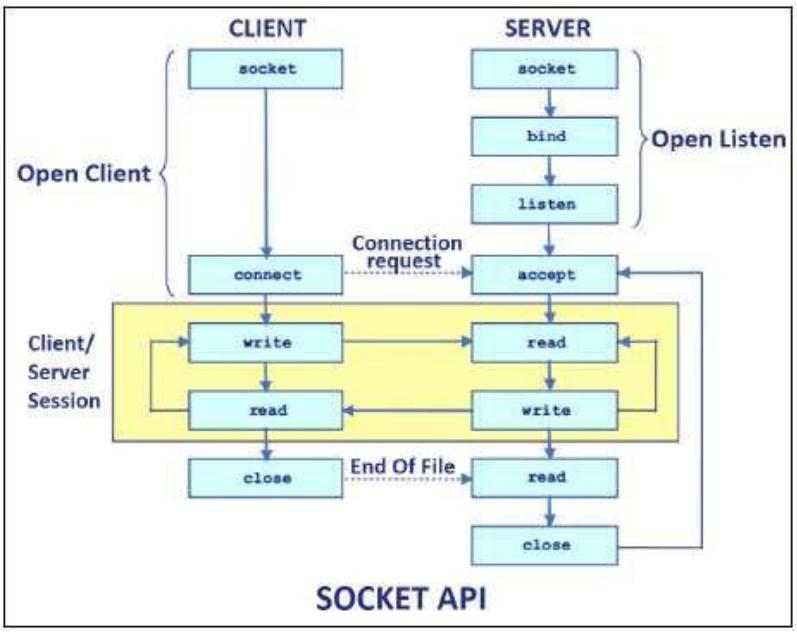
- **TCP** − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

- **UDP** − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

Socket programming for TCP:

The following steps occur when establishing a TCP connection between two computers using sockets −

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.
- After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.
- TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.



Socket programming for UDP:

UDP is used only when the entire information can be bundled into a single packet and there is no dependency on the other packet. Therefore, the usage of UDP is quite limited, whereas TCP is widely used in IP applications. UDP sockets are used where limited bandwidth is available, and the overhead associated with resending packets is not acceptable.

To connect using a UDP socket on a specific port, use the following code:

DatagramSocket udpSock = new DatagramSocket(3000);

A datagram is a self-contained, independent message whose time of arrival, confirmation of arrival over the network, and content cannot be guaranteed. atagramPacket objects are used to send data over DatagramSocket. Every DatagramPacket object consists of a data buffer, a remote host to whom the data needs to be sent, and a port number on which the remote agent would be listened.

Implementing the solution:

Socket Programming for TCP:

Client Programming:

1. **Establish a Socket Connection:** The **java.net Socket** class represents a Socket. To open a socket:

   Socket socket = new Socket("127.0.0.1", 5000)

2. **Communication:** To communicate over a socket connection, streams are used to both input and output the data.
3. **Closing the connection:** The socket connection is closed explicitly once the message to server is sent.

Server Programming:

1. **Establish a Socket Connection:** To write a server application two sockets are needed. A ServerSocket which waits for the client requests (when a client makes a new Socket()). A plain old Socket socket to use for communication with the client.
2. **Communication:** getOutputStream() method is used to send the output through the socket.
3. **Close the Connection:** After finishing, it is important to close the connection by closing the socket as well as input/output streams.

Compilation and Executing the solution:

If you're using Eclipse :

1. Compile both of them on two different terminals or tabs
2. Run the Server program first
3. Then run the Client program
4. Type messages in the Client Window which will be received and showed by the Server Window simultaneously if you are developing echo server application.
5. Close the socket connection by typing something like "Exit".

Conclusion:

In this assignment, the students learned about client-server communication through different protocols and sockets. They also learned about Java support through the socket API for TCP and UDP programming.

Outcome:

1. Students learned about interprocess communication in DS
2. Students develop application through implementing client-server communication programs based on Java Sockets

FAQ:

1. What is interprocess communication?
2. What is socket?
3. Difference between TCP and UDP socket communication?
4. What is shared memory programming?
5. What is port? State application of port.

Assignment 1

Title: Socket Programming

Code:

MyClient.java

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

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

      //The socket object takes ip and port number of the server which client wants to connect
      Socket s = new Socket("127.0.0.1",5555);
      System.out.println("Connected to Server, Please type your message and hit Enter to send");

      //Reading input from KeyBoard
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

      //OutputStream object to write to Server
      OutputStream ostream = s.getOutputStream();

      //PrintWriter object to send the data to the outputstream
      PrintWriter pw = new PrintWriter(ostream, true);

      //InputStream objects to recieve from Server
      InputStream istream = s.getInputStream();

      //Reading receieved message from Server
      BufferedReader recieve = new BufferedReader(new InputStreamReader(istream));
```

```
//Client Message and Server Message objects
String clientmessage = "";
String servermessage = "";

while(true)
{
   //Input Message to be sent to Server
   System.out.print("Client: ");
   clientmessage = br.readLine();

   //print writer object sending the message to the socket through outputstream
   pw.println(clientmessage);

   //if the message is bye end the communication here
   if(clientmessage.equals("bye"))
   {
      break;
   }

   //Read the inputstream of the server from the socket
   servermessage = recieve.readLine();
   System.out.println("Server: "+servermessage);

   //if the message is bye end the communication here
   if(servermessage.equals("bye"))
   {
      break;
   }
}
```

```
        //closing all the streams and sockets
        s.close();
        istream.close();
        ostream.close();

        System.out.println("Connection Terminated");
    }
}
```

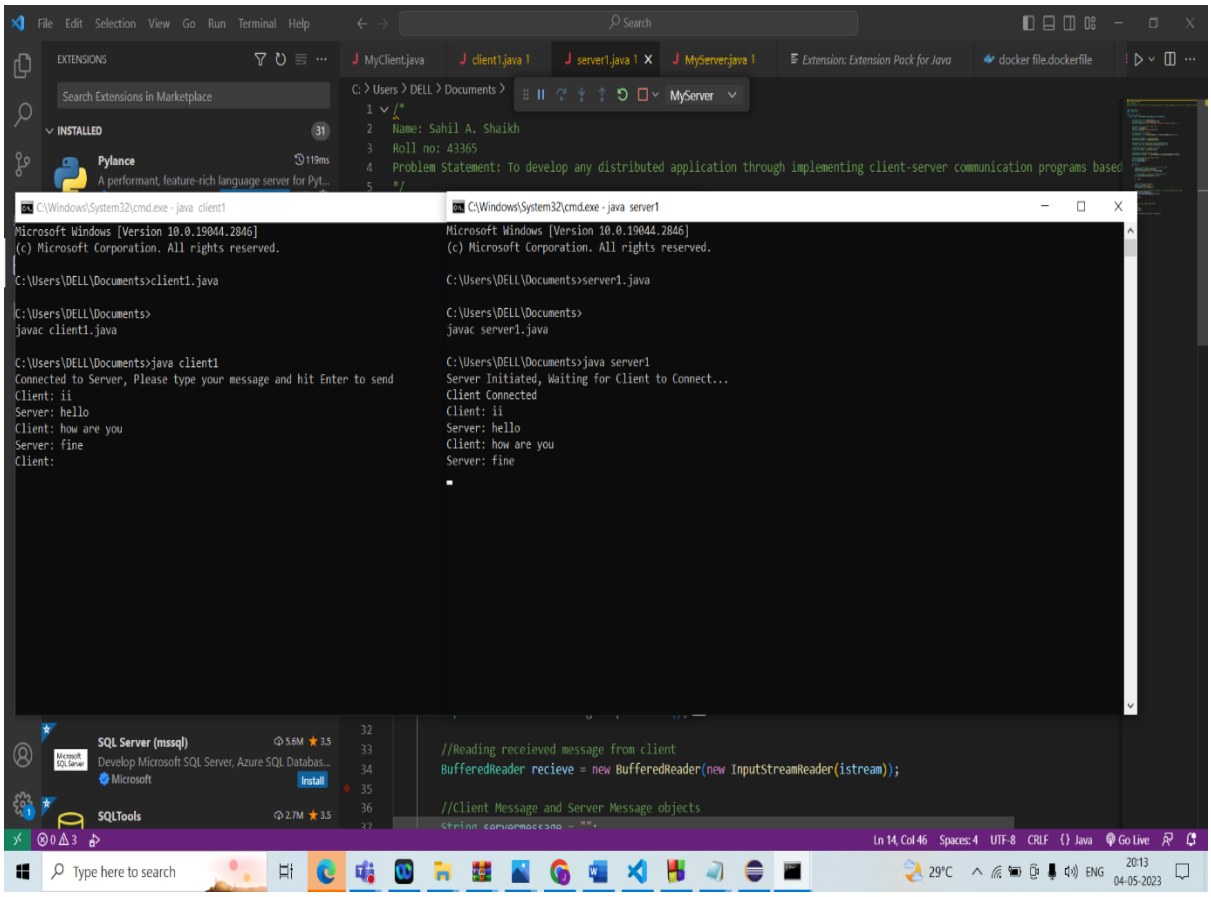
MyServer.java

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

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

        //Creating a port for communication
        ServerSocket ss = new ServerSocket(5555);
        System.out.println("Server Initiated, Waiting for Client to Connect...");

        //Binding Client and Server on port 5555
        Socket s = ss.accept();
        System.out.println("Client Connected");

        //Reading input from KeyBoard
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        //OutputStream object to write to clients
```

```
OutputStream ostream = s.getOutputStream();

//PrintWriter object to send the data to the outputstream
PrintWriter pw = new PrintWriter(ostream,true);

//InputStream objects to recieve from Client
InputStream istream = s.getInputStream();

//Reading receieved message from client
BufferedReader recieve = new BufferedReader(new InputStreamReader(istream));

//Client Message and Server Message objects
String servermessage = "";
String clientmessage = "";

while(true)
{
   //Read the inputstream of the client from the socket
   clientmessage = recieve.readLine();
   System.out.println("Client: "+clientmessage);

   //if the message is bye end the communication here
   if(clientmessage.equals("bye"))
   {
      break;
   }

   //Server writing its message
   System.out.print("Server: ");
   servermessage = br.readLine();
```

```
            //print writer object sending the message to the socket through outputstream
            pw.println(servermessage);
            if(servermessage.equals("bye"))
            {
                break;
            }
        }

        //closing all the streams and sockets
        s.close();
        ss.close();
        istream.close();
        ostream.close();

        System.out.println("Connection Terminated");
    }
}
```

Output:


C:\Windows\System32\cmd.exe - java client1
Microsoft Windows [Version 10.0.19044.2846]
(c) Microsoft Corporation. All rights reserved.
C:\Users\DELL\Documents>client1.java
C:\Users\DELL\Documents>
javac client1.java
C:\Users\DELL\Documents>java client1
Connected to Server, Please type your message and hit Enter to send
Client: ii
Server: hello
Client: how are you
Server: fine
Client:
C:\Windows\System32\cmd.exe - java server1
Microsoft Windows [Version 10.0.19044.2846]
(c) Microsoft Corporation. All rights reserved.
C:\Users\DELL\Documents>server1.java
C:\Users\DELL\Documents>
javac server1.java
C:\Users\DELL\Documents>java server1
Server Initiated, Waiting for Client to Connect...
Client Connected
Client: ii
Server: hello
Client: how are you
Server: fine
//Reading receieved message from client
BufferedReader recieve = new BufferedReader(new InputStreamReader(istream));
//Client Message and Server Message objects

Assignment 1

Q1. What is interprocess communication?

Ans: Interprocess communication (IPC) is the mechanism used by different processes or programs running on a computer system to communicate with each other and share data. IPC allows processes to exchange information and coordinate their actions, making it possible to build complex distributed systems and multi-tasking applications.

Q2. What is socket?

Ans: A socket is a software abstraction that represents an endpoint of a network connection. Sockets provide a standard interface for communication between processes over a network, and they can be used with different networking protocols, such as TCP, UDP, and IP. Sockets are commonly used in client-server applications, where a client process connects to a server process using a socket.

Q3. Difference between TCP and UDP socket communication?

Ans: TCP and UDP are two different transport layer protocols that can be used with sockets to communicate over a network. TCP provides reliable, ordered, and error-checked delivery of data between applications, while UDP provides unreliable, unordered, and unacknowledged delivery of data. TCP is best suited for applications that require a guaranteed delivery of data, while UDP is more suitable for applications that prioritize speed and efficiency over reliability, such as real-time multimedia streaming or online gaming.

Q4. What is shared memory programming?

Ans: Shared memory programming is a technique used in multi-process programming to allow multiple processes to share access to the same region of memory. Shared memory can be used to exchange data between processes, synchronize their actions, or speed up inter-process communication. Shared memory programming can be challenging and requires careful synchronization to avoid race conditions and other concurrency issues.

Q5. What is port? State application of port.

Ans: A port is a logical endpoint of a network connection that is used to identify a specific process or service on a computer system. Ports are identified by numbers between 0 and 65535, with well-known ports reserved for standard services such as HTTP (port 80), FTP (port 21), or SSH (port 22). Applications can use different ports to listen for incoming connections or initiate outgoing connections to remote servers. The state of a port refers to whether it is open, closed, or filtered by a firewall, and it can be used to troubleshoot network connectivity issues or secure network services.

ASSIGNMENT NO. 2

Title: Remote Method Invocation

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

Objectives:

1. To understand the basics of multi-threaded client/server Process communication.
2. To develop interprocess communication application using RMI.

Tools / Environment:

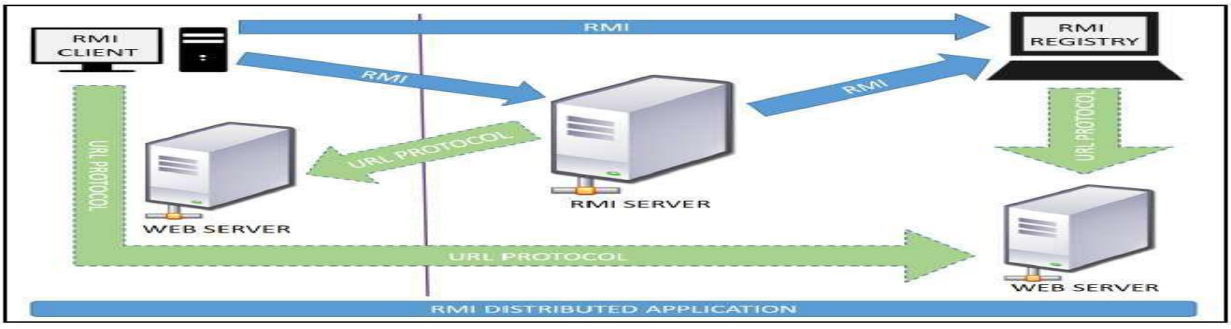
Java Programming Environment, jdk 1.8, rmiregistry

Related Theory:

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

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

RMI registry:



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

Key terminologies of RMI:

The following are some of the important terminologies used in a Remote Method Invocation. **Remote object**: This is an object in a specific JVM whose methods are exposed so they could be invoked by another program deployed on a different JVM.

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

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

Stub: This is a Java object that acts as an entry point for the client object to route any outgoing requests. It exists on the client JVM and represents the handle to the remote object.

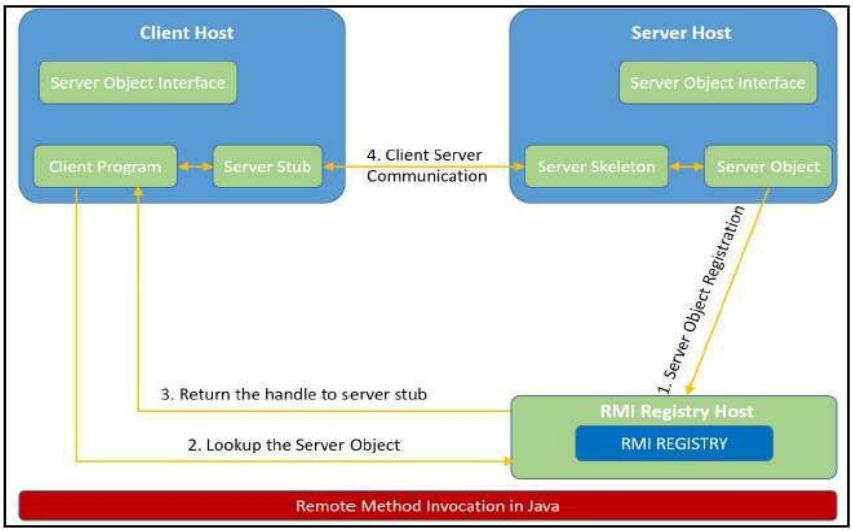
If any object invokes a method on the stub object, the stub establishes RMI by following these steps:

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

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

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

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



Designing the Solution:

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

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

2. Ensure that the components that participate in the RMI calls are accessible across networks.

3. Establish a network connection between applications that need to interact using the RMI.

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

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

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

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

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

Implementing the solution:

Consider building an application to perform diverse mathematical operations.

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

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

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

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

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

server as "AddServer". Its second argument is a reference to an instance of **AddServerImpl**.

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

The application begins by forming a string that follows the URL syntax. This URL uses the **rmi** protocol. The string includes the IP address or name of the server and the string "AddServer". The

program then invokes the **lookup( )** method of the **Naming** class. This method accepts one argument, the **rmi** URL, and returns a reference to an object of type **AddServerIntf**. All remote method invocations can then be directed to this object.

The program continues by displaying its arguments and then invokes the remote **add()** method.

The sum is returned from this method and is then printed.

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

2. Generate a Stub

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

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

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

rmic AddServerImpl.

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

3. Install Files on the Client and Server Machines

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

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

4. Start the RMI Registry on the Server Machine

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

start rmiregistry

5. Start the Server

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

java AddServer

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

6. Start the Client

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

java AddClient 192.168.13.14 7 8

Conclusion:

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

Outcome:

1. Students learn basics of multi-threaded client server process communication.
2. Students develop interprocess communication application using JAVA RMI.

FAQ:

1. What is Heterogeneity?
2. Example of is marshaling and unmarshalling?
3. Explain RMI with diagram.
4. What is binding?
5. What is role of RMI registry? why we start RMI registry first.
6. What is use of "UnicastRemoteObject","lookup(),rebind().
7. What is stub and skeleton?
8. What is difference between Exception and RemoteException

Assignment 2

Title: Remote Method Invocation

Code:

Server.java

```
import java.rmi.*;
import java.net.*;

public class Server {
   public static void main(String[] args) {
      try {
         Servant s = new Servant();
         Naming.rebind("Server", s);
      } catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

Servant.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.*;
import java.rmi.server.*;

public class Servant extends UnicastRemoteObject implements ServerInterface {
   protected Servant() throws RemoteException {
      super();
   }
```

```
    @Override
    public String concat(String a, String b) throws RemoteException {
        return a + b;
    }
}
```

ServerInterface.java

```
import java.rmi.*;

public interface ServerInterface extends Remote {
    String concat(String a, String b) throws RemoteException;
}
```

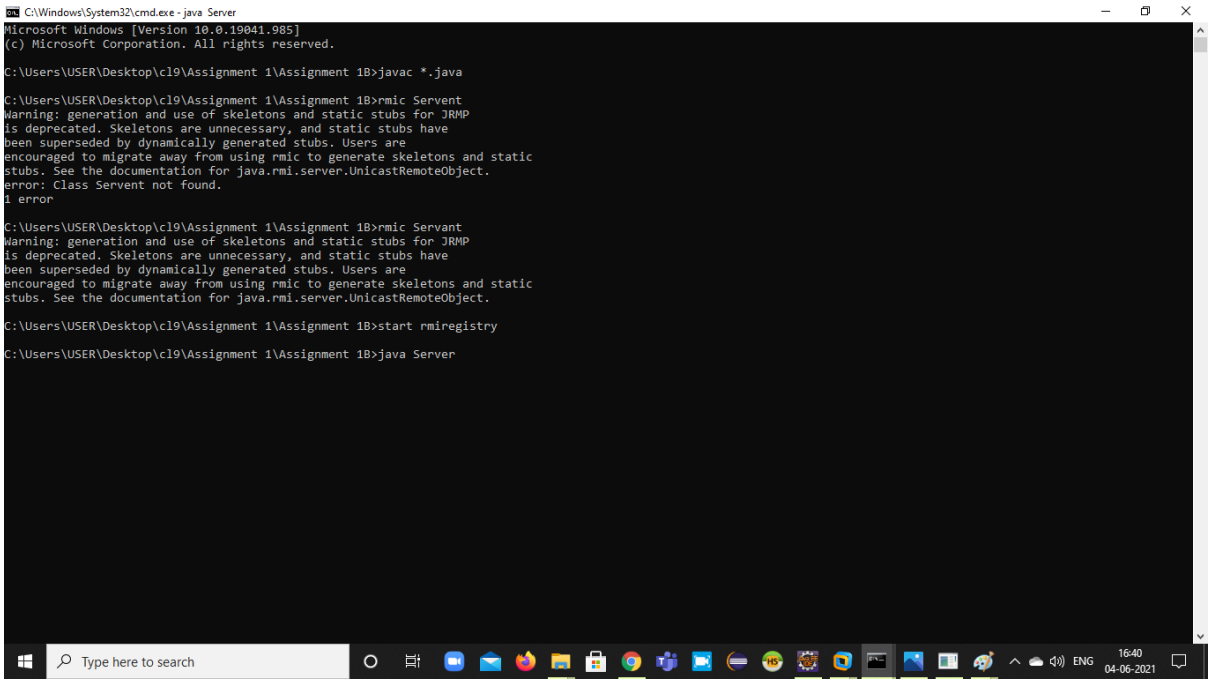
Client.java

```
import java.rmi.*;
import java.util.Scanner;

public class Client {
    public static void main(String args[]) {
        try {
            Scanner s = new Scanner(System.in);
            System.out.println("Enter the Server address : ");
            String server = s.nextLine();
            ServerInterface si = (ServerInterface) Naming.lookup("rmi://" + server + "/Server");
            System.out.println("Enter first string : ");
```

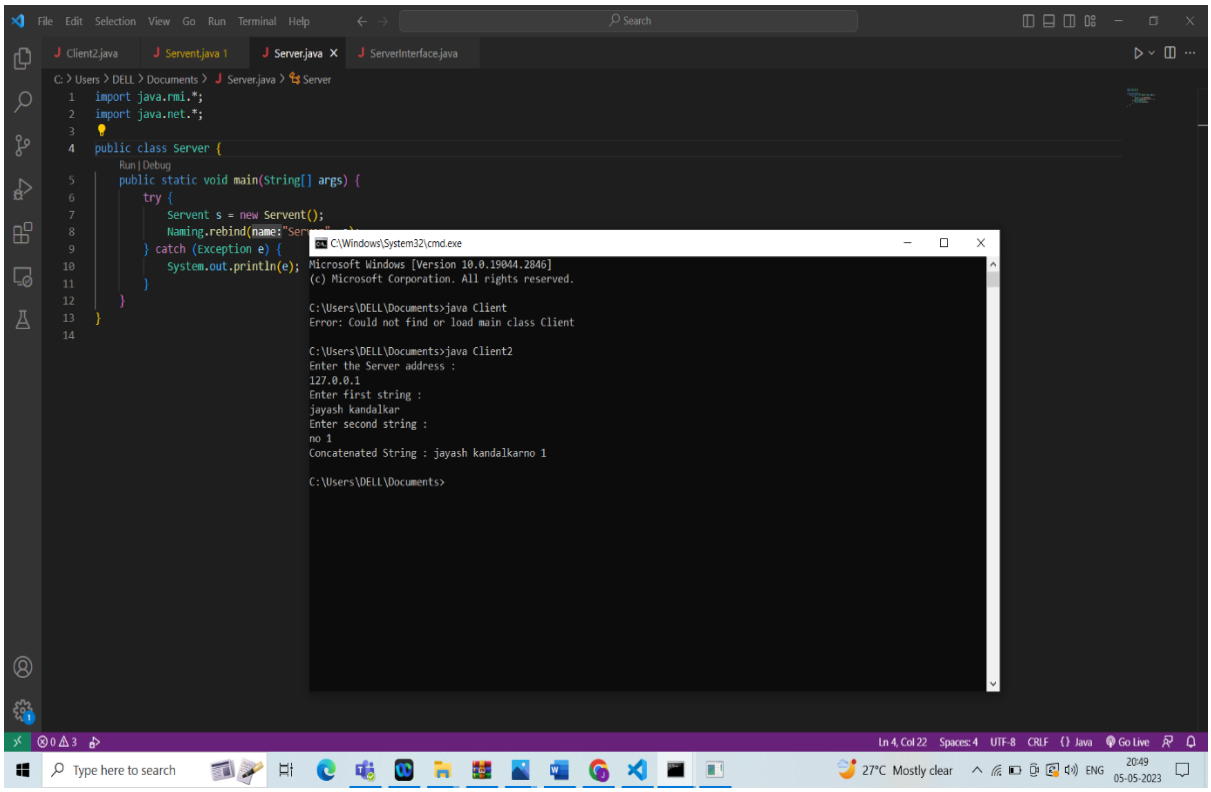
```
            String first = s.nextLine();
            System.out.println("Enter second string : ");
            String second = s.nextLine();
            System.out.println("Concatenated String : " + si.concat(first, second));
            s.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

OUTPUT:

Java Server is Running:



String Concanate using Multithreaded client server model:



Assignment 2

Q1. What is Heterogeneity?

Ans: Heterogeneity refers to the state of being diverse or different in composition or nature. In computer science, heterogeneity can refer to different types of hardware or software systems, different programming languages or interfaces, or different data formats and protocols.

Q2. Example of is marshaling and unmarshalling?

Ans: Marshaling and unmarshaling are techniques used to convert data between different representations, such as between binary data and a structured data format such as XML or JSON. An example of marshaling would be converting a data structure in memory to a binary stream that can be sent over a network, while an example of unmarshaling would be converting a binary stream back into a data structure in memory.

Q3. Explain RMI with diagram.

Ans: RMI stands for Remote Method Invocation, which is a Java-based technology for enabling distributed computing. RMI allows Java objects to invoke methods on remote objects running on different JVMs (Java Virtual Machines) over a network. The client and server run on different JVMs and communicate over a network. The client uses a remote stub, which is a local proxy object that represents the remote object running on the server. When the client invokes a method on the remote stub, the stub serializes the method call and sends it over the network to the remote object, which executes the method and returns the result to the stub, which deserializes the result and returns it to the client.

Q4. What is binding?

Ans: Binding is the process of associating a name or identifier with a resource, such as an object, a file, or a network address. In RMI, binding refers to the process of associating a name with a remote object, so that clients can look up the object by name and invoke its methods remotely.

Q5. What is role of RMI registry? why we start RMI registry first.

Ans: The RMI registry is a service that provides a centralized directory of named objects in an RMI system. The registry is started first because it needs to be running before other RMI services can be registered or looked up. The role of the RMI registry is to act as a naming service, allowing clients to look up remote objects by name and obtain their remote stubs.

Q6. What is use of "UnicastRemoteObject","lookup(),rebind().

Ans: 'UnicastRemoteObject' is a base class for implementing remote objects that can be accessed over a network using RMI. lookup() is a method provided by the RMI registry for looking up a remote object by name. rebind() is a method provided by the RMI registry for binding a remote object to a name, replacing any existing binding with the same name.

Q7. What is stub and skeleton?

Ans: In RMI, a stub is a local object that represents a remote object, providing a transparent interface for invoking remote methods. A skeleton is a server-side object that receives incoming requests from clients and dispatches them to the appropriate remote object.

Q8. What is difference between Exception and RemoteException

Ans: An Exception is a type of error that can occur during program execution, indicating an abnormal or unexpected condition. A 'RemoteException' is a type of 'Exception' that specifically indicates an error that occurred during an RMI method invocation. A 'RemoteException' can occur when a communication failure or other error prevents a remote method call from completing successfully.

ASSIGNMENT NO. 3

Title: Common Object Request Broker Architecture (CORBA)

Aim: To develop any distributed application with CORBA program using JAVA IDL.

Objective:

1. To understand the basics steps for implementation of CORBA.
2. To develop any distributed application using CORBA to demonstrate object brokering.

Tools / Environment:

Java Programming Environment, JDK 1.8

Related Theory:

Common Object Request Broker Architecture (CORBA):

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

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

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

Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR&Benefits" maintain an object

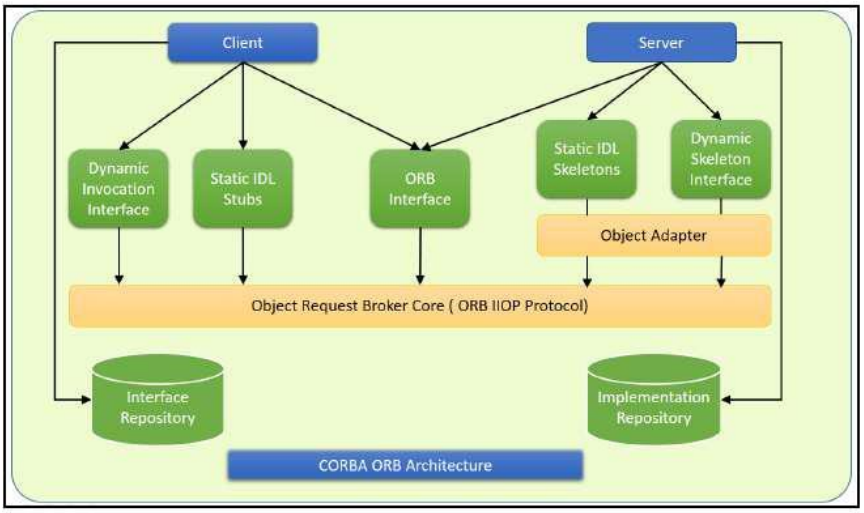
model with details of the organization, employees with demographic information, benefits, payroll, and deductions. They are only different in the way they handle the details, based on the country and region they are operating for. For each object type, similar to the HR&Benefits systems, we can define an interface using the **Interface Definition Language** (**OMG IDL**).

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

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

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

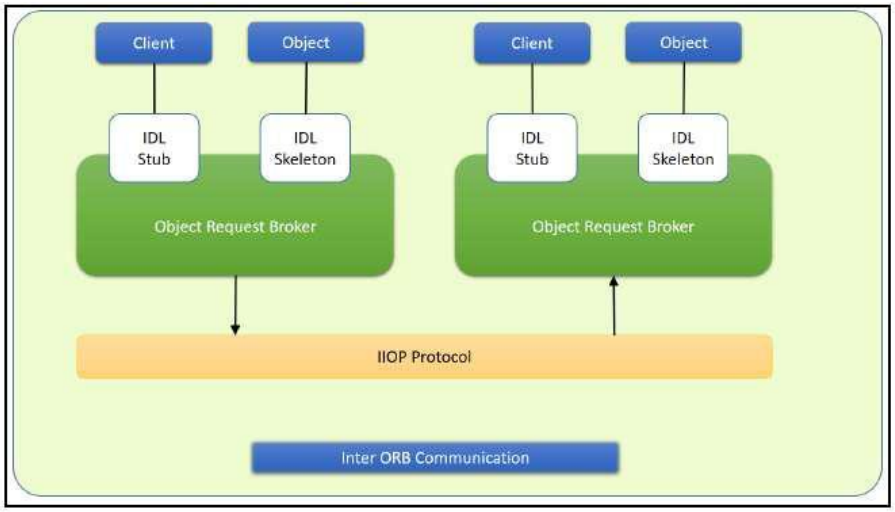
The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons, The objects are written (on the right) and a client for it (on the left), as represented in the diagram. The client and server use stubs and skeletons as proxies, respectively.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



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

Inter-ORB communication

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



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

Java Support for CORBA

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

CORBA standards provide the proven, interoperable infrastructure to the Java platform. IIOP (Internet Inter-ORB Protocol) manages the communication between the object components that power the system. The Java platform provides a portable object infrastructure that works on every major operating system. CORBA provides the network transparency, Java provides the implementation transparency. **An *Object Request Broker (ORB)* is part of the Java Platform.**

The ORB is a runtime component that can be used for distributed computing using IIOP communication. Java IDL is a Java API for interoperability and integration with CORBA. Java IDL included both a Java-based ORB, which supported IIOP, and the **IDL-to-Java**

compiler, for generating client-side stubs and server-side code skeletons. J2SE v.1.4 includes an

Object Request Broker Daemon (ORBD), which is used to enable clients to transparently locate and invoke persistent objects on servers in the CORBA environment.

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

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

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

The IDL Programming Model:

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

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

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

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

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

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

Designing the solution:

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

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

1. Creating CORBA Objects using Java IDL:

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

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

Interfaces in IDL are declared much like interfaces in Java.

1.2. Modules

Modules are declared in IDL using the module keyword, followed by a name for the module and an opening brace that starts the module scope. Everything defined within the scope of this module

(interfaces, constants, other modules) falls within the module and is referenced in other IDL modules using the syntax *modulename*::x. e.g.

```
//  IDL
module jen {

   module corba {

      interface NeatExample ...

   };

};
```

1.3. Interfaces

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

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

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

1.4 Data members and methods

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

```
readonly attribute string myString;
```

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

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

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

1.5 A complete IDL example

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

```
module OS {
   module services {
      interface Server {
         readonly attribute string serverName;
         boolean init(in string sName);
      };

      interface Printable {
         boolean print(in string header);
      };

      interface PrintServer : Server {

         boolean printThis(in Printable p);
      };
   };
};
```

The first interface, Server, has a single read-only 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 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 server-side implementation of the interface. The base class can accept requests for the object from the ORB and channel return values back through the ORB to the remote client. The skeleton class for an interface named Server is called _ServerImplBase.

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

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

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.

Outcome:

1. Students understand architecture and basics steps for implementation of CORBA
2. Students develop distributed application using CORBA to demonstrate object brokering for string and arithmetic operations.

FAQ:

1. What is CORBA?
2. How CORBA works?
3. Does it synchronous/Asynchronous application?
4. What is ORB?
5. What is IDL interface?
6. What is Object Request Broker Daemon (ORBD).
7. What is middleware.
8. List the examples of middleware.
9. List the use of middleware.
10. List the applications of CORBA.

Assignment 3

Title: Common Object Request Broker Architecture (CORBA)

Code:

StartServer.java

```
import Calculator.Calc;
import Calculator.CalcHelper;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

public class StartServer {

   public static void main(String args[]) {
      try{
         // create and initialize the ORB
         ORB orb = ORB.init(args, null);

         // get reference to rootpoa & activate the POAManager
         POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         rootpoa.the_POAManager().activate();

         // create servant and register it with the ORB
         CalcObject calcObj = new CalcObject();
         calcObj.setORB(orb);
```

```
        // get object reference from the servant
        org.omg.CORBA.Object ref = rootpoa.servant_to_reference(calcObj);
        Calc href = CalcHelper.narrow(ref);

        // get the root naming context
        // NameService invokes the name service
        org.omg.CORBA.Object nsRef =  orb.resolve_initial_references("NameService");

        // Use NamingContextExt which is part of the Interoperable
        // Naming Service (INS) specification.
        NamingContextExt ncRef = NamingContextExtHelper.narrow(nsRef);

        // bind the Object Reference in Naming
        NameComponent path[] = ncRef.to_name("Calculator");
        ncRef.rebind(path, href);

        System.out.println("CalculatorServer is listening...");

        // wait for invocations from clients
        orb.run();
        System.out.println("I am out");
      }
      catch (Exception e) {
        System.err.println("Server Error: " + e.getMessage());
        e.printStackTrace(System.out);
      }
    }
}
```

StartClient.java

```
import Calculator.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import java.util.*;

public class StartClient {
   private static Calc calcObj;

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      try {
         // create and initialize the ORB
      ORB orb = ORB.init(args, null);

         // get the root naming context
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");

         // Use NamingContextExt instead of NamingContext. This is
         // part of the Interoperable naming Service.
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

         // resolve the Object Reference in Naming
      calcObj = (Calc) CalcHelper.narrow(ncRef.resolve_str("Calculator"));

         while(true) {
            // asking for input and read it
            System.out.println("----------------------------------------");
```

```
            System.out.println("Enter the parameters in this format
[operator][sp][operand1][sp][operand2]."
                    + "\nFor example: + 1 2");
            Scanner c=new Scanner(System.in);
        String input = c.nextLine();


            // if the command is exit, request the server to shutdown
            if (input.toLowerCase().equals("exit")) {
               calcObj.exit();
               break;
            }


            // test the input
            String[] inputParams = input.split(" ");
            if (inputParams.length != 3) {
               System.out.println("Client Exception: Wrong number of parameters. Try again...");
               continue;
            }
            int operatorCode;
            int operand1;
            int operand2;


            // set calculation type
            if (inputParams[0].equals("+")) {
               operatorCode = 1;
            }
            else if (inputParams[0].equals("-")) {
               operatorCode = 2;
            }
            else if (inputParams[0].equals("*")) {
               operatorCode = 3;
```

```
        }
        else if (inputParams[0].equals("/")) {
            operatorCode = 4;
        }
        else {
            System.out.println("Client Exception: Un-recognized operation code. Try again...");
            continue;
        }

        // test input operands are integers
        try {
            operand1 = Integer.parseInt(inputParams[1]);
            operand2 = Integer.parseInt(inputParams[2]);
        }
        catch (NumberFormatException e) {
            System.out.println("Client Exception: Wrong number format. Try again...");
            continue;
        }

        // check if it is divided by zero
        if (operatorCode == 4 && operand2 == 0) {
            System.out.println("Client Exception: Can't be divided by zero. Try again...");
            continue;
        }

        // do the calculation and return result
    int result = calcObj.calculate(operatorCode, operand1, operand2);
        String resultDisplay = "";
        if (result == Integer.MAX_VALUE) {
            resultDisplay = "There might be an Integer Overflow. Please try again...";
        }
```

```
            else if (result == Integer.MIN_VALUE) {
               resultDisplay = "There might be an Integer Underflow. Please try again...";
            }
            else {
               resultDisplay = String.valueOf(result);
            }
      System.out.println("The result is: " + resultDisplay);
         }
      }
      catch (Exception e) {
         System.out.println("Client exception: " + e.getMessage());
         e.printStackTrace();
      }
   }
}
```

Calculator.idl

```
module Calculator {
   interface Calc {
      long calculate (in long opcode, in long op1, in long op2);
      oneway void exit();
   };
};
```

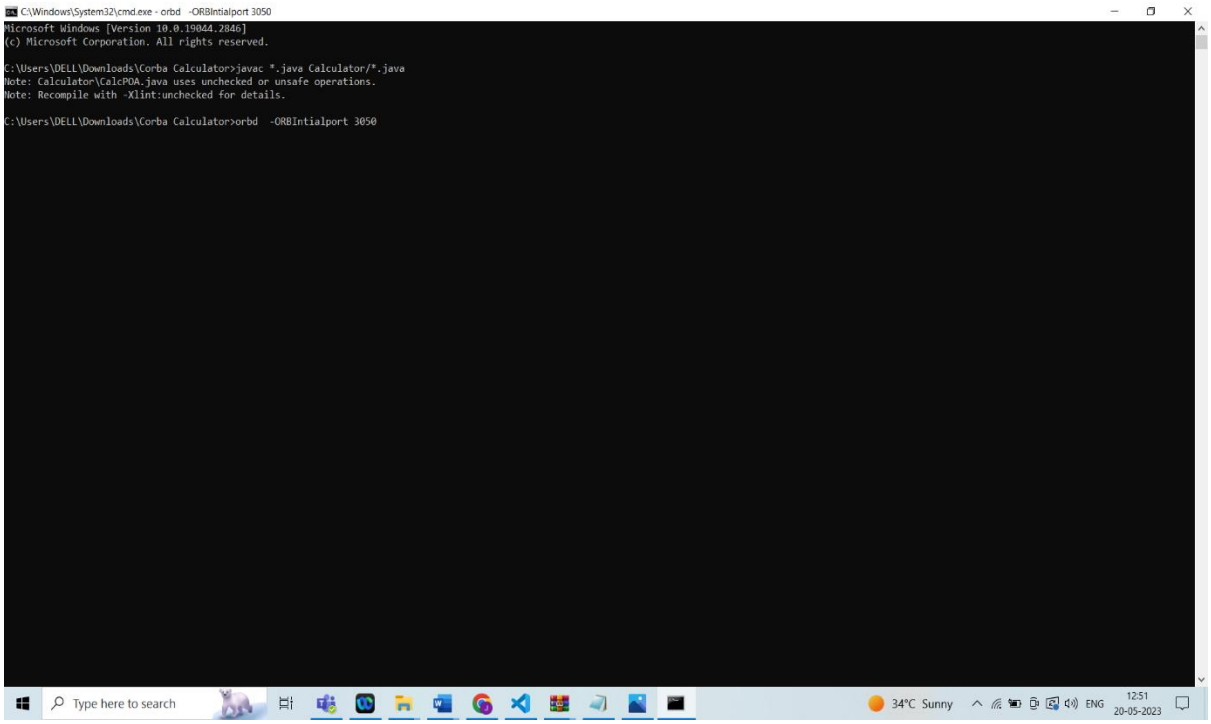
Calcobject.java

```
public class CalcObject extends CalcPOA{
   private ORB orb;

   public void setORB(ORB orb) {
      this.orb = orb;
```

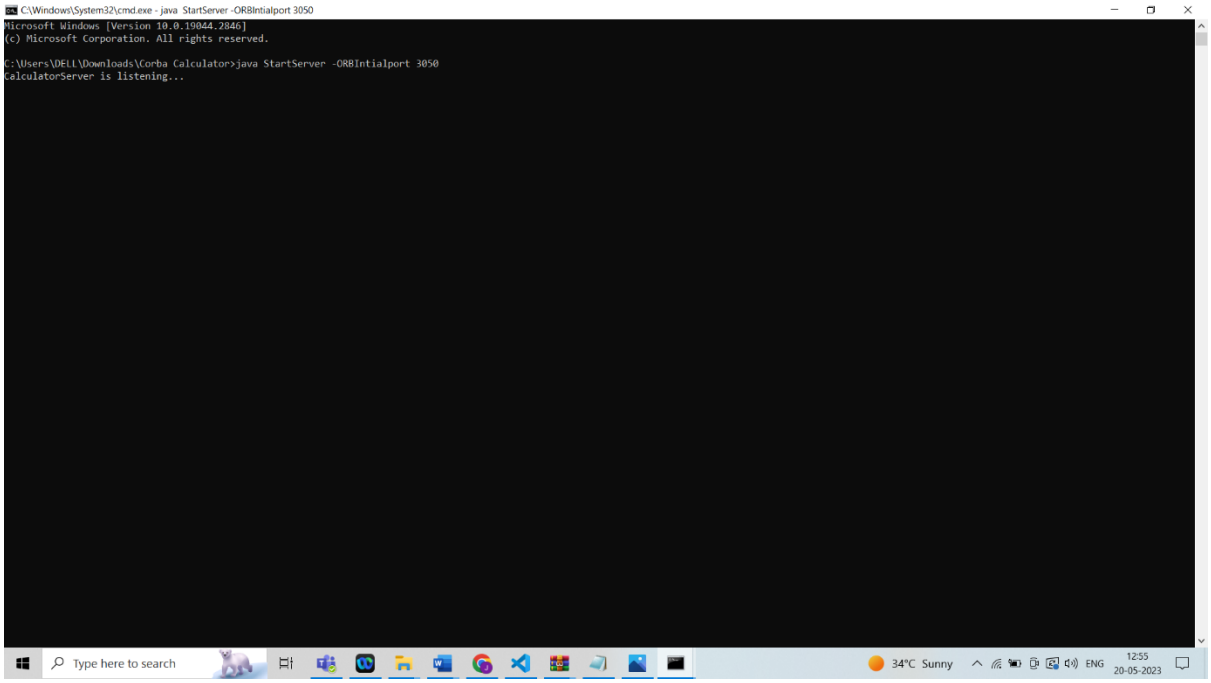
```
}

/** Calculate
   * @param type the type of the operation, 1 -> +, 2 -> -, 3 -> *, 4 -> /
* @param a first number
* @param b second number
* @return calculation result
*/
@Override
public int calculate(int type, int a, int b) {
   long result;

   if (type == 1) {
      result = (long) a + b;
   }
   else if (type == 2) {
      result = (long) a - b;
   }
   else if (type == 3) {
      result = (long) a * b;
   }
   else{
      result = (long) a / b;
   }

   if (result >= Integer.MAX_VALUE) {
      return Integer.MAX_VALUE;
   }
   else if (result <= Integer.MIN_VALUE) {
      return Integer.MIN_VALUE;
   }
```

```
        else {
            return (int) result;
        }
    }

    @Override
    public void exit() {
        orb.shutdown(false);
    }
}
```

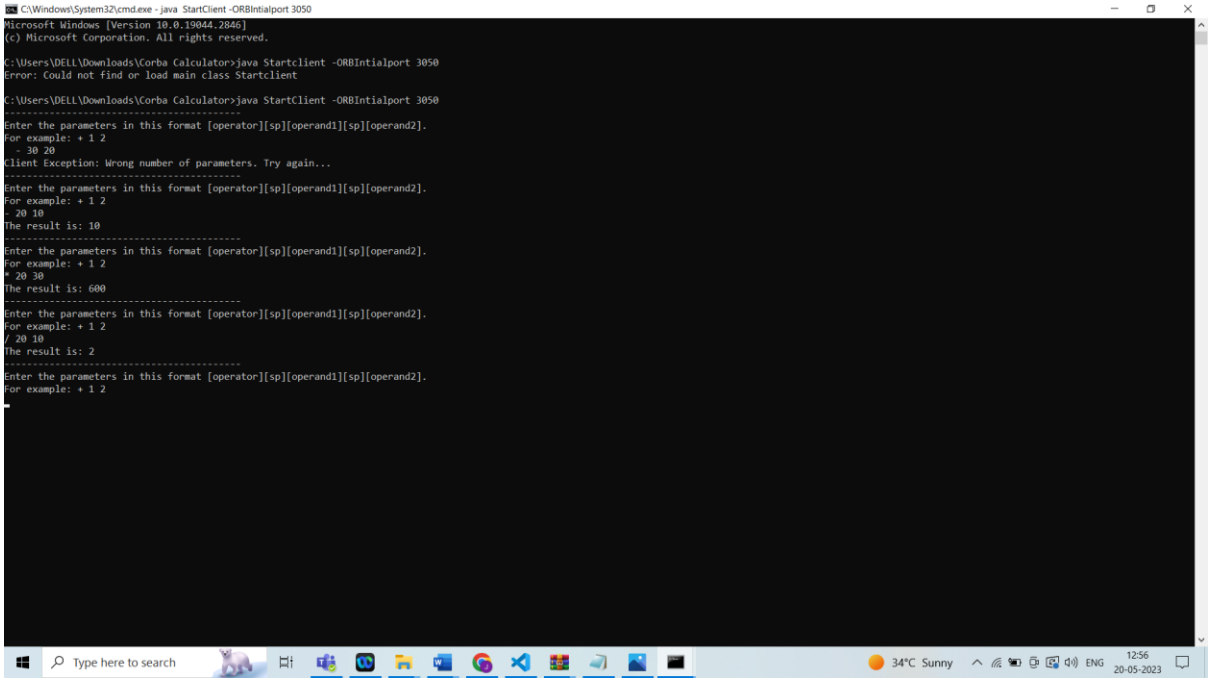
Output:



Server is Running:



Calculator output:



Assignment 3

Q1. What is CORBA?

Ans: CORBA stands for Common Object Request Broker Architecture. It is a standard defined by the Object Management Group (OMG) for enabling distributed computing across heterogeneous systems, including different hardware, operating systems, programming languages, and networks.

Q2. How CORBA works?

Ans: CORBA works by using an Object Request Broker (ORB) to facilitate communication between distributed objects. A client sends a request to the ORB, which marshals the request and sends it over the network to a remote object. The remote object receives the request, processes it, and sends a response back to the ORB, which marshals the response and sends it back to the client. The ORB provides services such as object activation, object location, and transaction support.

Q3. Does it synchronous/Asynchronous application?

Ans: CORBA can support both synchronous and asynchronous communication between objects.

Q4. What is ORB?

Ans: ORB stands for Object Request Broker, which is the core component of the CORBA architecture. The ORB is responsible for locating objects, marshalling and unmarshalling data, and providing other services such as security and transaction management.

Q5. What is IDL interface?

Ans: IDL stands for Interface Definition Language, which is a language used to describe the interface of a CORBA object. IDL is used to specify the methods and properties of a remote object, as well as the data types used by those methods.

Q6. What is Object Request Broker Daemon (ORBD).

Ans: Object Request Broker Daemon (ORBD) is a utility program that provides a central registry for CORBA objects. ORBD runs in the background and provides services such as object activation, object location, and security management.

Q7. What is middleware?

Ans: Middleware is software that acts as a bridge between different applications or systems, enabling them to communicate and share data. Middleware can provide services such as message passing, remote procedure calls, and data persistence.

Q8. List the examples of middleware

Ans: Examples of middleware include CORBA, Message Queuing, Web Services, Remote Procedure Calls (RPC), and Distributed Object Technologies.

Q9. List the use of middleware.

Ans: Middleware is used to simplify the process of developing distributed systems by providing a common set of services and abstractions that can be used by different applications. Middleware can provide benefits such as improved interoperability, increased scalability, and better reliability.

Q10. List the applications of CORBA

Ans: Applications of CORBA include distributed systems such as telecommunications networks, financial services, healthcare systems, and manufacturing systems. CORBA is also used in embedded systems and real-time systems.

ASSIGNMENT NO. 4

Title: Message Passing Interface (MPI)

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

Objective:

1. To introduce you to the fundamentals of MPI.
2. To understand widely used standard for writing message passing programs.

Tools / Environment:

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

Related Theory:

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

Currently developers can produce more efficient and effective parallel applications using message passing.

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

Message-Passing Interface Basics:

Every MPI program must contain the preprocessor directive:

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

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

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

The principal weakness of this limited form of processing is that the processes on different nodes run entirely independent of each other. It cannot enable capability or coordinated computing. **To get the different processes to interact, the concept of communicators is needed.** MPI programs are made up of concurrent processes executing at the same time that in almost all cases

are also communicating with each other. To do this, an object called the "communicator" is provided by MPI. Thus the user may specify any number of communicators within an MPI program, each with its own set of processes. "**MPI_COMM_WORLD**" communicator contains all the concurrent processes making up an MPI program.

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

int MPI_Comm_size(MPI_Comm comm, int _size).

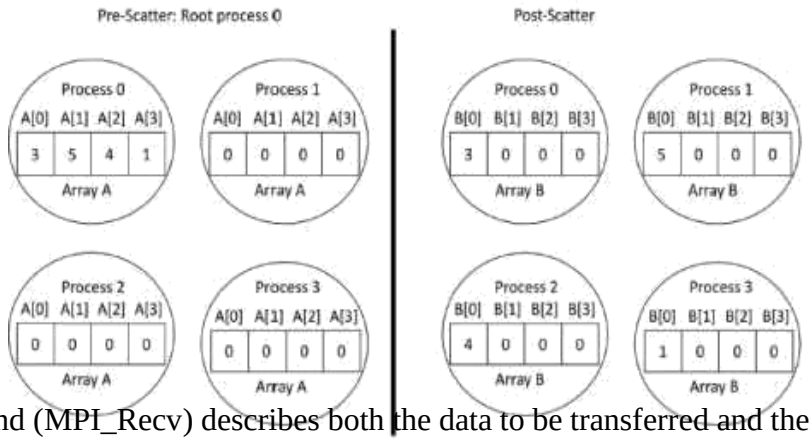
The function "MPI_Comm_size" required to return the number of processes; int size. MPI_Comm_size(MPI_COMM_WORLD,&size); This will put the total number of processes in the MPI_COMM_WORLD communicator in the variable size of the process data context. Every process within the communicator has a unique ID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

int MPI_Comm_rank (MPI_Comm comm, int _rank).

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

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

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



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

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

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

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

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

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

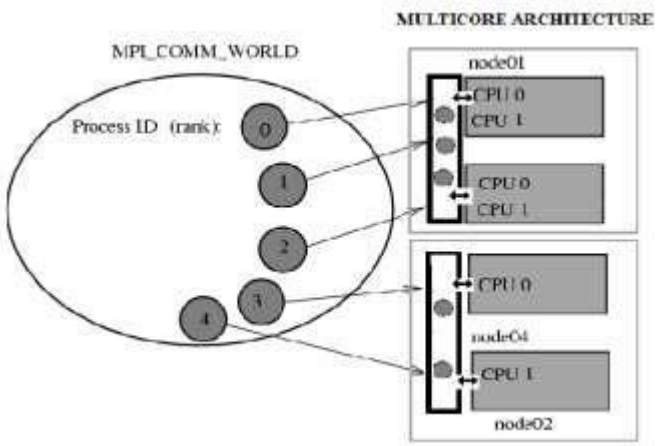
MPJ Express is essentially a middleware that supports communication between individual processors of clusters. **The programming model followed by MPJ Express is Single Program**

Multiple Data (SPMD).

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

Designing the solution:

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



Implementing the solution:

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

Conclusion:

Thus Student develop a distributed system application, 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.

Outcome:

1. Students learn fundamentals of parallel programming and MPI.
2. Students design parallel algorithms and write parallel programs using the MPI library.

FAQ:

1. What is use of MPI?
2. Application in which we are using MPI?
3. Why we are providing rank to process in MPI.
4. Explain MPI operations.
5. Explain Different data types of MPI.
6. Draw MPI architecture.
7. What is MPI_ABORT?
8. What is MPI_FINALIZE
9. What is difference MPI_ABORT and MPI_FINALIZE

Assignment 4

Title: Message Passing Interface (MPI)

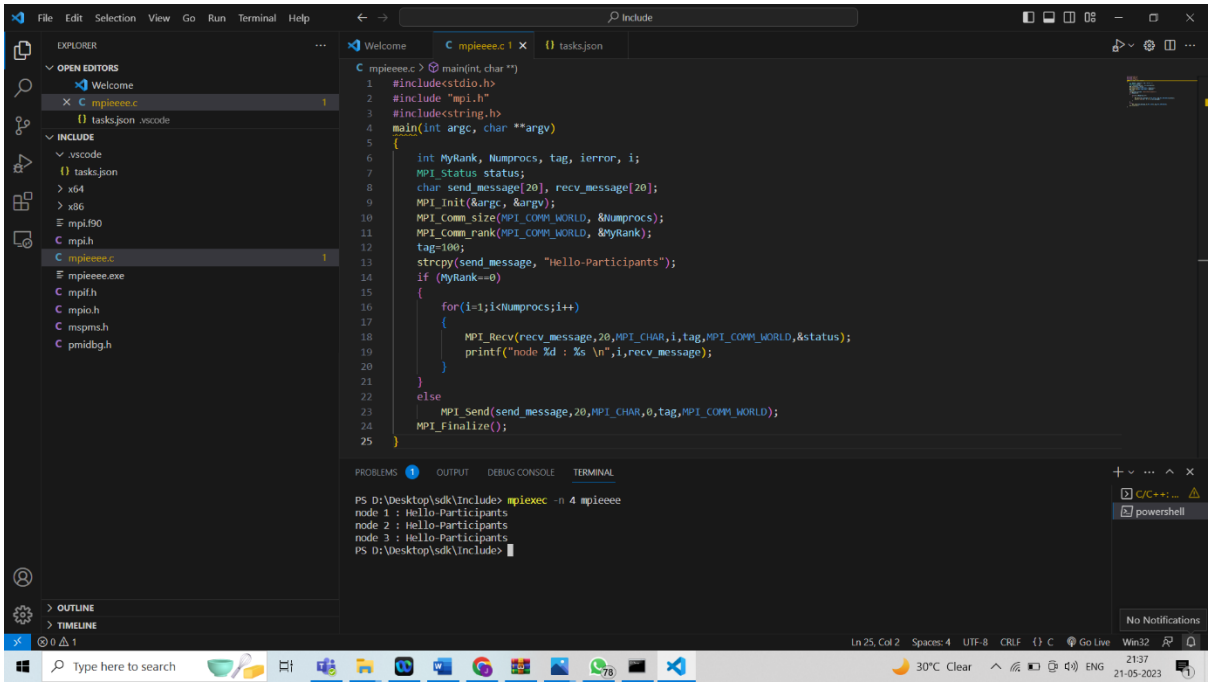
Code:

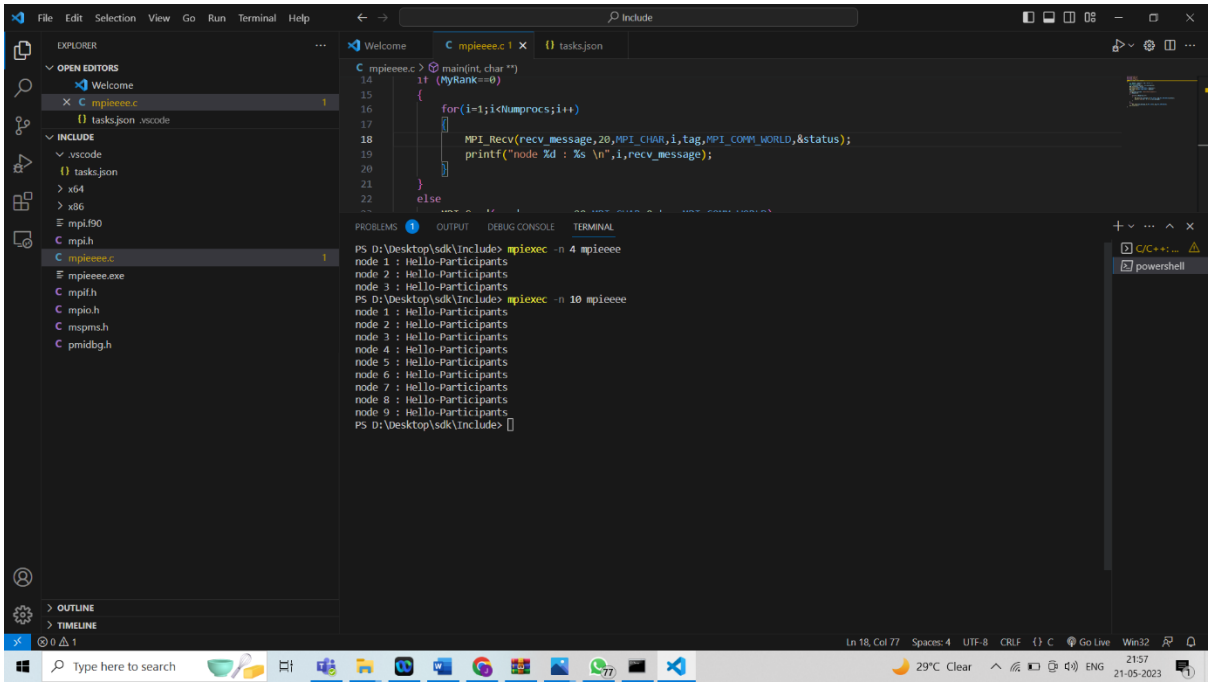
Hello world program

```
#include<stdio.h>
#include "mpi.h"
#include<string.h>
main(int argc, char **argv)
{
   int MyRank, Numprocs, tag, ierror, i;
   MPI_Status status;
   char send_message[20], recv_message[20];
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &Numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
   tag=100;
   strcpy(send_message, "Hello-Participants");
   if (MyRank==0)
   {
      for(i=1;i<Numprocs;i++)
      {
         MPI_Recv(recv_message,20,MPI_CHAR,i,tag,MPI_COMM_WORLD,&status);
         printf("node %d : %s \n",i,recv_message);
      }
   }
   else
      MPI_Send(send_message,20,MPI_CHAR,0,tag,MPI_COMM_WORLD);
   MPI_Finalize();
```

}

Output:


mpieeee.c
#include<stdio.h>
#include "mpi.h"
#include<string.h>
main(int argc, char **argv)
{
int MyRank, Numprocs, tag, ierror, i;
MPI_Status status;
char send_message[20], recv_message[20];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &Numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
tag=100;
strcpy(send_message, "Hello-Participants");
if (MyRank==0)
{
for(i=1;i<Numprocs;i++)
{
MPI_Recv(recv_message,20,MPI_CHAR,i,tag,MPI_COMM_WORLD,&status);
printf("node %d : %s \n",i,recv_message);
}
}
else
MPI_Send(send_message,20,MPI_CHAR,0,tag,MPI_COMM_WORLD);
MPI_Finalize();
}
PS D:\Desktop\sdk\Include> mpiexec -n 4 mpieeee
node 1 : Hello-Participants
node 2 : Hello-Participants
node 3 : Hello-Participants
PS D:\Desktop\sdk\Include>


PS D:\Desktop\sdk\Include> mpiexec -n 4 mpieeee
node 1 : Hello-Participants
node 2 : Hello-Participants
node 3 : Hello-Participants
PS D:\Desktop\sdk\Include> mpiexec -n 10 mpieeee
node 1 : Hello-Participants
node 2 : Hello-Participants
node 3 : Hello-Participants
node 4 : Hello-Participants
node 5 : Hello-Participants
node 6 : Hello-Participants
node 7 : Hello-Participants
node 8 : Hello-Participants
node 9 : Hello-Participants
PS D:\Desktop\sdk\Include>

Assignment 4

Q1. What is use of MPI?

Ans: MPI (Message Passing Interface) is a library specification for parallel computing using distributed memory systems. It provides a standardized way for multiple processes running on different nodes to communicate and coordinate their actions.

Q2. Application in which we are using MPI?

Ans: MPI is used in a wide range of scientific and engineering applications, including computational fluid dynamics, molecular dynamics simulations, weather forecasting, and seismic analysis.

Q3. Why we are providing rank to process in MPI.

Ans: The rank is a unique identifier assigned to each process in an MPI program. It is used to differentiate between different processes and to specify the source and destination of messages. Without a rank, it would be difficult to determine which process is sending or receiving a message.

Q4. Explain MPI operations.

Ans: MPI operations are functions that perform common parallel computing tasks such as sending and receiving messages, synchronizing processes, and performing collective operations such as broadcasting and reducing data. Some of the most commonly used MPI operations include MPI_Send, MPI_Recv, MPI_Barrier, MPI_Bcast, and MPI_Reduce.

Q5. Explain Different data types of MPI.

Ans: MPI supports a variety of data types, including integer, float, double, character, and complex data types. It also supports derived data types such as structures and arrays.

Q6. Draw MPI architecture.

Ans: The MPI architecture consists of a group of processes running on different nodes, connected by a communication network. Each process has its own local memory, and processes communicate with each other by passing messages over the network. The MPI library provides functions that enable processes to send and receive messages, synchronize their actions, and perform collective operations.

Q7. What is MPI_ABORT?

Ans: MPI_ABORT is a function that immediately terminates the MPI program. It can be used to handle fatal errors or exceptional conditions that require the program to exit immediately.

Q8. What is MPI_FINALIZE

Ans: MPI_FINALIZE is a function that cleans up resources used by the MPI library and terminates the MPI program. It should be called at the end of every MPI program.

Q9. What is difference MPI_ABORT and MPI_FINALIZE

Ans: The main difference between MPI_ABORT and MPI_FINALIZE is that MPI_ABORT immediately terminates the program without any cleanup, while MPI_FINALIZE cleans up resources and ensures that all processes have completed their tasks before exiting the program. MPI_ABORT is typically used to handle fatal errors or exceptional conditions, while MPI_FINALIZE is used to gracefully exit the program.

ASSIGNMENT NO. 5

Title: Clock Synchronization

Aim: To implement Berkeley algorithm for clock synchronization.

Objective:

1. To understand the basics of physical and Logical clock in DS.
2. To develop an n-node distributed system that implements Berkeley's time synchronization algorithm.

Related Theory:

Berkeley's Algorithm is a clock synchronization technique used in distributed systems. The algorithm assumes that each machine node in the network either doesn't have an accurate time source or doesn't possess an UTC server.

Algorithm:

- An individual node is chosen as the master node from a pool nodes in the network. This node is the main node in the network which acts as a master and rest of the nodes act as slaves. Master node is chosen using a election process/leader election algorithm.

- Master node periodically pings slaves nodes and fetches clock time at them using Cristian's algorithm.

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

- Improvision in accuracy of cristian's algorithm.

- Ignoring significant outliers in calculation of average time difference

- In case 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 master's unavailability.

- Instead of sending the synchronized time, master broadcasts relative inverse time difference, which leads to decrease in latency induced by traversal time in the network while time of calculation at slave node.

Scope of Improvement

- Improvision in accuracy of cristian's algorithm.
- Ignoring significant outliers in calculation of average time difference
- In case 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 master's unavailability.
- Instead of sending the synchronized time, master broadcasts relative inverse time difference, which leads to decrease in latency induced by traversal time in the network while time of calculation at slave node.

Conclusion:

The Berkeley algorithm is a simple yet effective solution for clock synchronization in distributed systems. Its decentralized approach allows for resilience against failures, and it remains a relevant and widely used tool in ensuring accurate timekeeping between machines.

Outcome:

1. Students learn fundamental of clock synchronization in DS.
2. Students implemented Berkeley algorithm for clock synchronization.

FAQ:

1. What is difference between logical clock and physical clock?
2. Why is it necessary to synchronize the clocks in distributed real time system?
3. How the principle of Berkeley algorithm is used to synchronize time in distributed system?
4. What are other algorithms for clock synchronization in DS?

ASSIGNMENT NO. 5

Title: Clock Synchronization

Code:

Server.java

```
# Python3 program imitating a clock server

from functools import reduce
from dateutil import parser
import threading
import datetime
import socket
import time

# datastructure used to store client address and clock data
client_data = {}

''' nested thread function used to receive
    clock time from a connected client '''
def startReceivingClockTime(connector, address):

    while True:
        # receive clock time
        clock_time_string = connector.recv(1024).decode()
        clock_time = parser.parse(clock_time_string)
        clock_time_diff = datetime.datetime.now() - \
                                    clock_time

        client_data[address] = {
```

```
                "clock_time"     : clock_time,
                "time_difference" : clock_time_diff,
                "connector"  : connector
                }

        print("Client Data updated with: "+ str(address),
                                    end = "\n\n")
        time.sleep(5)

''' master thread function used to open portal for
    accepting clients over given port '''
def startConnecting(master_server):

    # fetch clock time at slaves / clients
    while True:
        # accepting a client / slave clock client
        master_slave_connector, addr = master_server.accept()
        slave_address = str(addr[0]) + ":" + str(addr[1])

        print(slave_address + " got connected successfully")

        current_thread = threading.Thread(
                    target = startReceivingClockTime,
                    args = (master_slave_connector,
                                slave_address, ))
        current_thread.start()

# subroutine function used to fetch average clock difference
def getAverageClockDiff():

    current_client_data = client_data.copy()
```

```
    time_difference_list = list(client['time_difference']
                                for client_addr, client
                                   in client_data.items())


    sum_of_clock_difference = sum(time_difference_list, \
                                datetime.timedelta(0, 0))

    average_clock_difference = sum_of_clock_difference \
                                    / len(client_data)

    return average_clock_difference

''' master sync thread function used to generate
    cycles of clock synchronization in the network '''
def synchronizeAllClocks():

    while True:

        print("New synchronization cycle started.")
        print("Number of clients to be synchronized: " + \
                                str(len(client_data)))

        if len(client_data) > 0:

            average_clock_difference = getAverageClockDiff()

            for client_addr, client in client_data.items():
                try:
                    synchronized_time = \
```

```
                datetime.datetime.now() + \
                        average_clock_difference

            client['connector'].send(str(
                synchronized_time).encode())

        except Exception as e:
            print("Something went wrong while " + \
                "sending synchronized time " + \
                "through " + str(client_addr))

    else :
        print("No client data." + \
                " Synchronization not applicable.")

    print("\n\n")

    time.sleep(5)

# function used to initiate the Clock Server / Master Node
def initiateClockServer(port = 8080):

    master_server = socket.socket()
    master_server.setsockopt(socket.SOL_SOCKET,
                        socket.SO_REUSEADDR, 1)

    print("Socket at master node created successfully\n")

    master_server.bind(('', port))

    # Start listening to requests
```

```
    master_server.listen(10)
    print("Clock server started...\n")

    # start making connections
    print("Starting to make connections...\n")
    master_thread = threading.Thread(
                    target = startConnecting,
                    args = (master_server, ))
    master_thread.start()

    # start synchronization
    print("Starting synchronization parallelly...\n")
    sync_thread = threading.Thread(
                    target = synchronizeAllClocks,
                    args = ())
    sync_thread.start()


# Driver function
if __name__ == '__main__':

    # Trigger the Clock Server
    initiateClockServer(port = 2050)
```

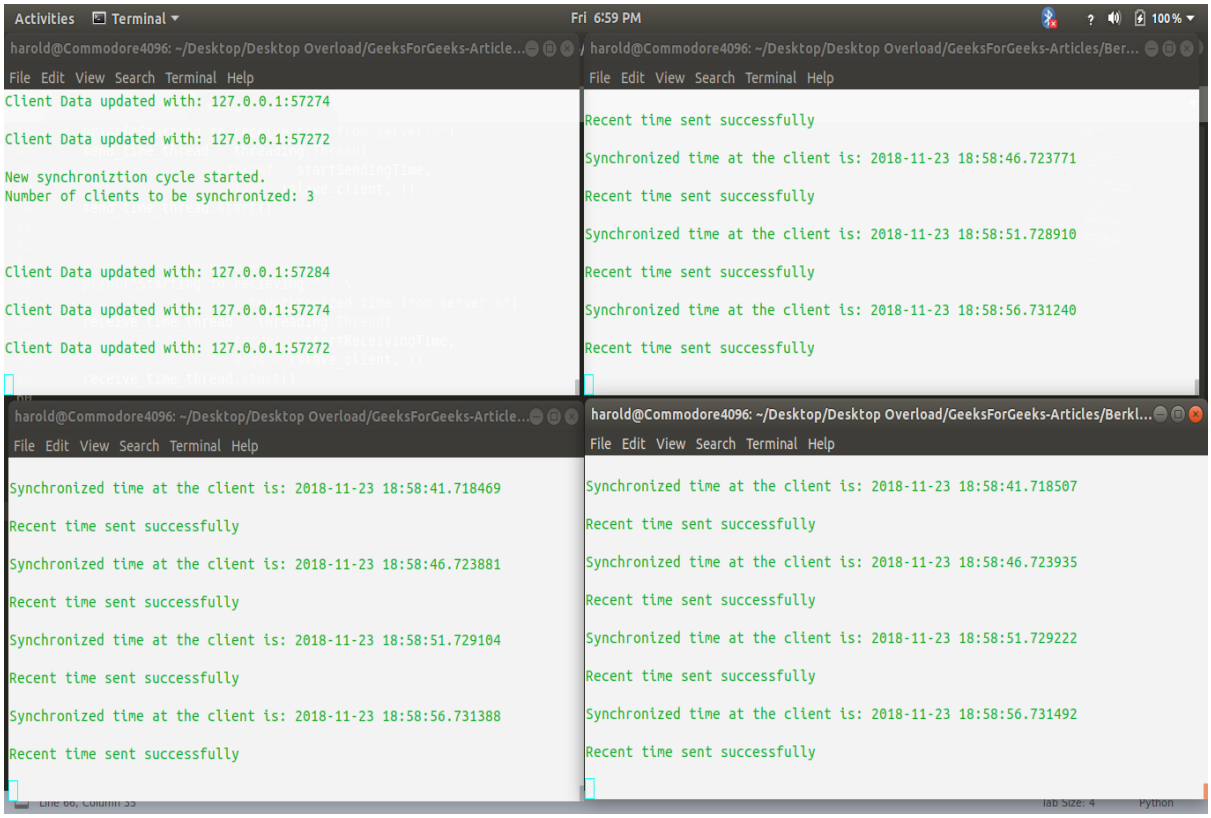
Client.py

```
# Python3 program imitating a client process

from timeit import default_timer as timer
from dateutil import parser
import threading
import datetime
import socket
import time

# client thread function used to send time at client side
def startSendingTime(slave_client):

    while True:
        # provide server with clock time at the client
        slave_client.send(str(
                datetime.datetime.now()).encode())

        print("Recent time sent successfully",
                                end = "\n\n")
        time.sleep(5)

# client thread function used to receive synchronized time
def startReceivingTime(slave_client):

    while True:
        # receive data from the server
        Synchronized_time = parser.parse(
                    slave_client.recv(1024).decode())
```

```
        print("Synchronized time at the client is: " + \
                              str(Synchronized_time),
                              end = "\n\n")

# function used to Synchronize client process time
def initiateSlaveClient(port = 8080):

    slave_client = socket.socket()

    # connect to the clock server on local computer
    slave_client.connect(('127.0.0.1', port))

    # start sending time to server
    print("Starting to receive time from server\n")
    send_time_thread = threading.Thread(
                target = startSendingTime,
                args = (slave_client, ))
    send_time_thread.start()

    # start receiving synchronized from server
    print("Starting to receiving " + \
                    "synchronized time from server\n")
    receive_time_thread = threading.Thread(
                target = startReceivingTime,
                args = (slave_client, ))
    receive_time_thread.start()

# Driver function
if __name__ == '__main__':

    # initialize the Slave / Client
```

initiateSlaveClient(port = 2050)

output:



Assignment 5

Q1. What is difference between logical clock and physical clock?

Ans: The main difference between logical clocks and physical clocks is that physical clocks measure time based on the passage of time in the real world, while logical clocks measure time based on events in the system. Physical clocks are typically based on hardware such as a quartz crystal oscillator and provide accurate time measurements, while logical clocks are typically implemented in software and may not provide accurate time measurements.

Q2. Why is it necessary to synchronize the clocks in distributed real time system?

Ans: In a distributed real-time system, it is necessary to synchronize the clocks to ensure that all nodes in the system are working with the same time reference. This is important for ensuring consistency and correctness in the system, especially in applications that require coordinated action or require events to occur in a specific sequence.

Q3. How the principle of Berkeley algorithm is used to synchronize time in distributed system?

Ans: The Berkeley algorithm is a popular algorithm for clock synchronization in distributed systems. The basic principle of the algorithm is to periodically synchronize the clocks of all nodes in the system with a reference clock maintained by a time server. The time server periodically broadcasts its current time to all nodes in the system, and each node adjusts its clock to match the time server's clock. This approach helps to ensure that all nodes in the system are working with a common time reference.

Q4. What are other algorithms for clock synchronization in DS.

Ans: Other algorithms for clock synchronization in distributed systems include the Cristian's algorithm, which is similar to the Berkeley algorithm but uses a different approach for calculating clock adjustments, and the Network Time Protocol (NTP), which is a widely used protocol for clock synchronization on the internet. NTP uses a hierarchical approach with multiple time servers and algorithms to account for network delays and minimize clock skew.

ASSIGNMENT NO. 6

Title: Mutual Exclusion

Aim: Implement token ring based mutual exclusion algorithm.

Objective:

1. To understand the concept of starvation, Mutual Exclusion in DS.
2. To Achieve Mutual Exclusion In Distributed System based on Token Exchange.

Related Theory:

Token Ring algorithm achieves mutual exclusion in a distributed system by creating a bus network of processes. A logical ring is constructed with these processes and each process is assigned a position in the ring. Each process knows who is next in line after itself. When the ring is initialized, process 0 is given a token. The token circulates around the ring. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token to the next process in the ring. It is not allowed to enter the critical region again using the same token. If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes the token along to the next process.

Advantages:

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

Disadvantages

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

Conclusion:

The token ring-based mutual exclusion algorithm is a well-known solution for coordinating access to shared resources in distributed systems. Its simple and efficient design ensures that only one process can access a shared resource at a time, thus preventing conflicts and ensuring consistency. While the algorithm can suffer from potential delays and network congestion, it remains a widely used and effective solution for achieving mutual exclusion in distributed systems.

Outcome:

1. Students learn concept of Mutual exclusion to prevent Race conditions.
2. Students have implemented token ring based mutual exclusion algorithm.

FAQ:

1. What is race condition?
2. What is deadlock and starvation?
3. What is Mutual Exclusion?
4. How to avoid mutual exclusion using

ASSIGNMENT NO. 6

Title: Mutual Exclusion

Code:

TokenServer1.java

```
import java.io.*;
import java.net.*;
public class TokenServer1
{
   public static void main(String agrs[])throws Exception
      {

         while(true)
         {
         Server sr=new Server();
         sr.recPort(8000);
         sr.recData();
         }
      }
}
class Server
{

   boolean hasToken=false;
   boolean sendData=false;
   int recport;

   void recPort(int recport)
   {
```

```
        this.recport=recport;
    }
    void recData()throws Exception
    {
        byte buff[]=new byte[256];
        DatagramSocket ds;
        DatagramPacket dp;
        String str;

        ds=new DatagramSocket(recport);
        dp=new DatagramPacket(buff,buff.length);
        ds.receive(dp);
        ds.close();

        str=new String(dp.getData(),0,dp.getLength());
        System.out.println("The message is "+str);
    }
}
```

TokenClient1.java

```
import java.io.*;
import java.net.*;
public class TokenClient1
{
    public static void main(String arg[]) throws Exception
        {
            InetAddress lclhost;
            BufferedReader br;
            String str="";
```

```
TokenClient12 tkcl,tkser;
boolean hasToken;
boolean setSendData;

while(true)
{
lclhost=InetAddress.getLocalHost();
tkcl = new TokenClient12(lclhost);
tkser = new TokenClient12(lclhost);
//tkcl.setSendPort(9001);
tkcl.setSendPort(9004);
tkcl.setRecPort(8002);
lclhost=InetAddress.getLocalHost();
tkser.setSendPort(9000);
if(tkcl.hasToken == true)
{

System.out.println("Do you want to enter the Data –> YES/NO");
   br=new BufferedReader(new InputStreamReader(System.in));
   str=br.readLine();
   if(str.equalsIgnoreCase("yes"))
   {
      System.out.println("ready to send");
      tkser.setSendData = true;
      tkser.sendData();
      tkser.setSendData = false;
   }
   else if(str.equalsIgnoreCase("no"))
   {
      System.out.println("i m in else");
      //tkcl.hasToken=false;
```

```
                tkcl.sendData();
                tkcl.recData();
            System.out.println("i m leaving else");
            }
        }
        else
        {
        System.out.println("ENTERING RECEIVING MODE…");
            tkcl.recData();
        }
    }
}
}

class TokenClient12
{
    InetAddress lclhost;
    int sendport,recport;
    boolean hasToken = true;
    boolean setSendData = false;
    TokenClient12 tkcl,tkser;
    TokenClient12(InetAddress lclhost)
    {

        this.lclhost = lclhost;
    }

    void setSendPort(int sendport)
    {
        this.sendport = sendport;
    }
```

```
    void setRecPort(int recport)
    {
       this.recport = recport;
    }

void sendData() throws Exception
       {
       BufferedReader br;
       String str="Token";
       DatagramSocket ds;
       DatagramPacket dp;


       if(setSendData == true)
       {
          System.out.println("sending ");
          System.out.println("Enter the Data");
          br=new BufferedReader(new InputStreamReader(System.in));
          str = "ClientOne....." + br.readLine();
          System.out.println("now sending");

       }
          ds = new DatagramSocket(sendport);
          dp = new DatagramPacket(str.getBytes(),str.length(),lclhost,sendport-1000);
          ds.send(dp);
          ds.close();
          setSendData = false;
          hasToken = false;
    }

    void recData()throws Exception
```

```
    {
        String msgstr;
        byte buffer[] = new byte[256];
        DatagramSocket ds;
        DatagramPacket dp;
        ds = new DatagramSocket(recport);
        dp = new DatagramPacket(buffer,buffer.length);
        ds.receive(dp);
        ds.close();
        msgstr = new String(dp.getData(),0,dp.getLength());
        System.out.println("The data is "+msgstr);

        if(msgstr.equals("Token"))
          {
             hasToken = true;
          }
    }

}
```

TokenClient2.java

```
import java.io.*;
import java.net.*;
public class TokenClient2
{
    static boolean setSendData ;
    static boolean hasToken ;
    public static void main(String arg[]) throws Exception
        {
```

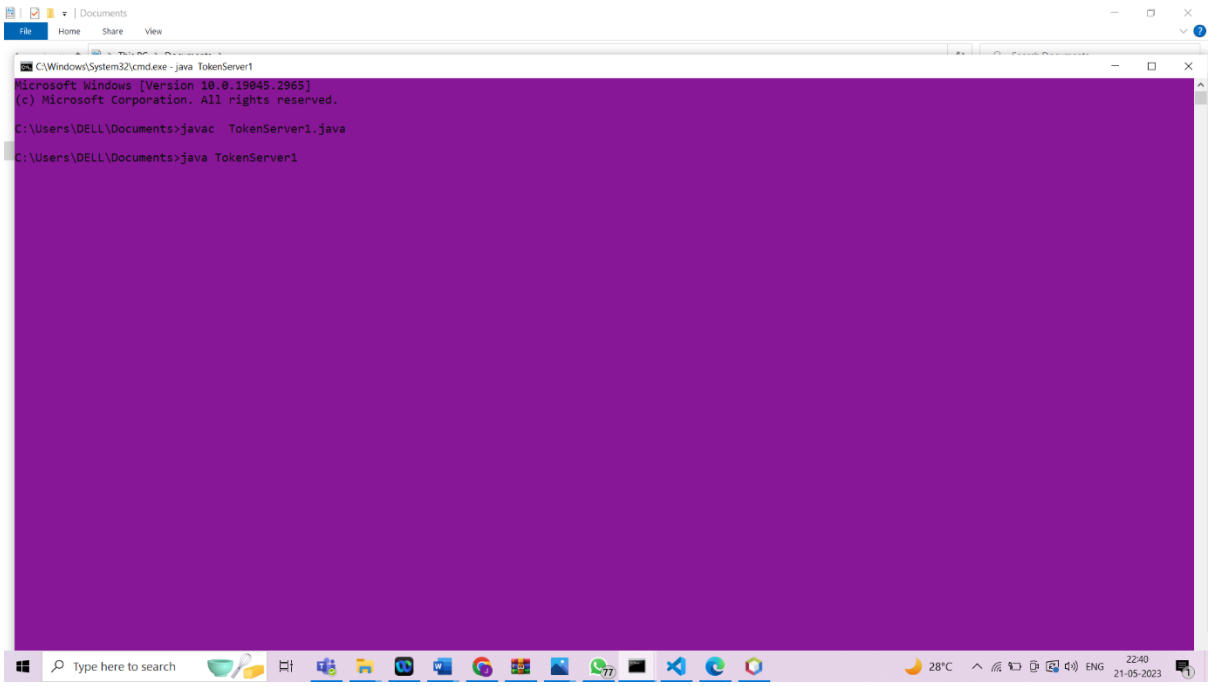
```
InetAddress lclhost;
BufferedReader br;
String str1;
TokenClient21 tkcl;
TokenClient21 ser;
while(true)
{
lclhost=InetAddress.getLocalHost();
tkcl = new TokenClient21(lclhost);
tkcl.setRecPort(8004);
tkcl.setSendPort(9002);
lclhost=InetAddress.getLocalHost();
ser = new TokenClient21(lclhost);
ser.setSendPort(9000);
System.out.println("entering if");
if(hasToken == true)
{

System.out.println("Do you want to enter the Data –> YES/NO");
   br=new BufferedReader(new InputStreamReader(System.in));
   str1=br.readLine();
   if(str1.equalsIgnoreCase("yes"))
   {
      System.out.println("ignorecase");
      ser.setSendData = true;
      ser.sendData();
      }
   else if(str1.equalsIgnoreCase("no"))
   {
      tkcl.sendData();
      hasToken=false;
```

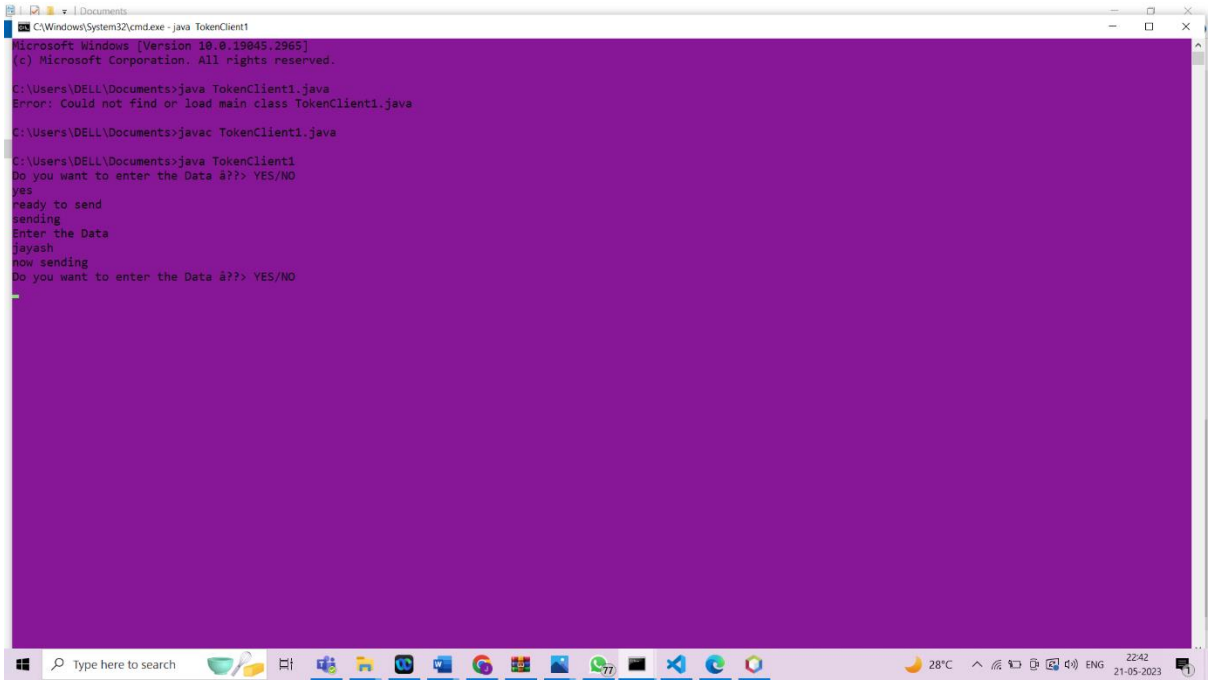
```
            }
        }
        else
        {
        System.out.println("entering recieving mode");
            tkcl.recData();
            hasToken=true;
        }
      }
   }
}
class TokenClient21
{
   InetAddress lclhost;
   int sendport,recport;
   boolean setSendData = false;
   boolean hasToken = false;
   TokenClient21 tkcl;
   TokenClient21 ser;

   TokenClient21(InetAddress lclhost)
   {

      this.lclhost = lclhost;
   }

   void setSendPort(int sendport)
   {
      this.sendport = sendport;
   }
   void setRecPort(int recport)
```

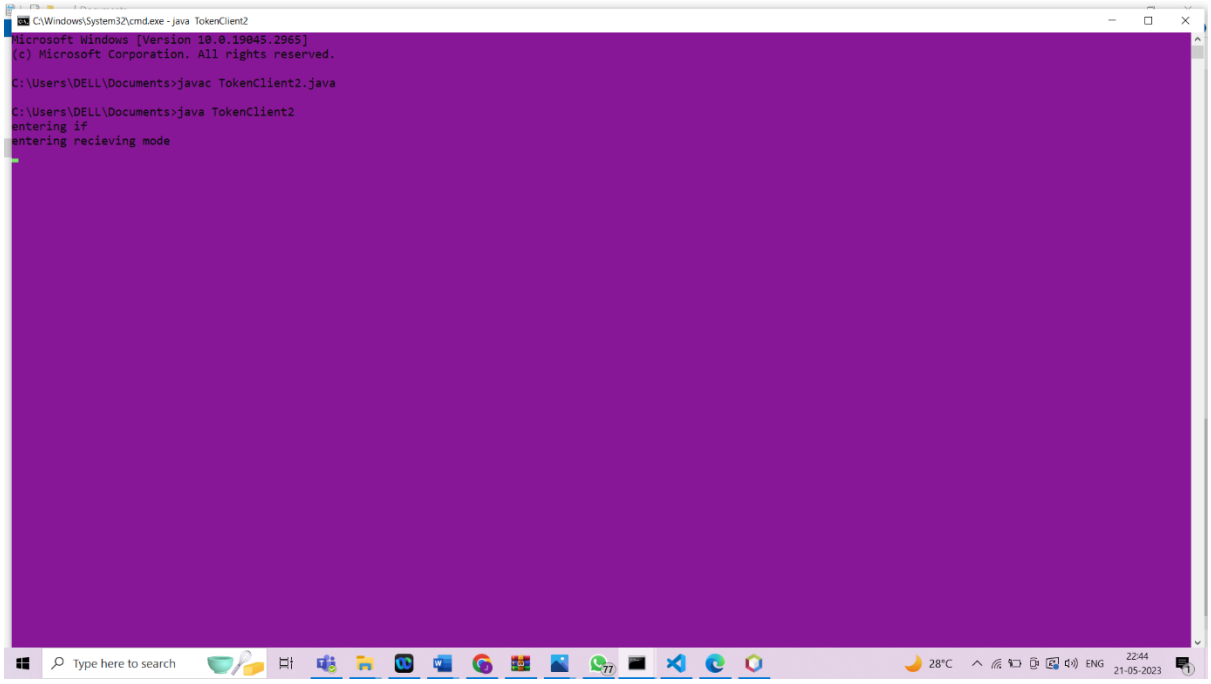
```
  {
     this.recport = recport;
  }
  void sendData() throws Exception
  {
     System.out.println("case");
     BufferedReader br;
     String str="Token";
     DatagramSocket ds;
     DatagramPacket dp;

     if(setSendData == true)
     {
        System.out.println("Enter the Data");
        br=new BufferedReader(new InputStreamReader(System.in));
        str = "ClientTwo....." + br.readLine();
     }
        ds = new DatagramSocket(sendport);
        dp = new DatagramPacket(str.getBytes(),str.length(),lclhost,sendport-1000);
        ds.send(dp);
        ds.close();
        System.out.println("Data Sent");
        setSendData = false;
        hasToken = false;

  }


void recData()throws Exception
  {
     String msgstr;
```

```
        byte buffer[] = new byte[256];
        DatagramSocket ds;
        DatagramPacket dp;
ds = new DatagramSocket(recport);
        //ds = new DatagramSocket(4000);
        dp = new DatagramPacket(buffer,buffer.length);
        ds.receive(dp);
        ds.close();
msgstr = new String(dp.getData(),0,dp.getLength());
        System.out.println("The data is "+msgstr);
        if(msgstr.equals("Token"))
          {
             hasToken = true;
          }
    }

}
```

Output:




C:\Windows\System32\cmd.exe - java TokenClient1
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.
C:\Users\DELL\Documents>java TokenClient1.java
Error: Could not find or load main class TokenClient1.java
C:\Users\DELL\Documents>javac TokenClient1.java
C:\Users\DELL\Documents>java TokenClient1
Do you want to enter the Data â??> YES/NO
yes
ready to send
sending
Enter the Data
jayash
now sending
Do you want to enter the Data â??> YES/NO
Type here to search
28°C
ENG
22:42
21-05-2023


C:\Windows\System32\cmd.exe - java TokenClient2
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.
C:\Users\DELL\Documents>javac TokenClient2.java
C:\Users\DELL\Documents>java TokenClient2
entering if
entering recieving mode
Type here to search
28°C
ENG
22:44
21-05-2023

Assignment 6

Q1. What is race condition?

Ans: A race condition is a situation that can occur in concurrent programming when two or more processes or threads access a shared resource in an unpredictable order, leading to unexpected or incorrect behavior. For example, if two processes try to update the same variable at the same time, it may result in a race condition where the final value of the variable depends on the order in which the processes execute.

Q2. What is deadlock and starvation?

Ans: Deadlock and starvation are two common problems that can occur in concurrent programming. Deadlock occurs when two or more processes are blocked waiting for each other to release a resource, resulting in a state where no progress can be made. Starvation occurs when a process is unable to access a resource it needs to continue executing, often due to other processes monopolizing the resource.

Q3. What is Mutual Exclusion?

Ans: Mutual exclusion is a technique used in concurrent programming to ensure that only one process or thread can access a shared resource at a time. This is typically achieved using locks or other synchronization primitives to ensure that only one process holds a lock on a resource at a time. Mutual exclusion is important to prevent race conditions and ensure correct behavior in concurrent programs.

ASSIGNMENT NO. 7

Title: Election Algorithms

Aim: To Implement Bully and Ring algorithm for leader election.

Objective:

1. To enable distributed systems to select a leader in a decentralized manner, without requiring a centralized control mechanism.
2. To ensure that the leader selection process is reliable and efficient, even in the presence of failures, network delays, and other forms of uncertainty.
3. To provide a fair and deterministic method for selecting the leader, such that all nodes in the system have an equal chance of being chosen.

Tools / Environment:

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

Related Theory: Election Algorithm:

1. Many distributed algorithms require a process to act as a coordinator.
2. The coordinator can be any process that organizes actions of other processes.
3. *A* coordinator may fail.
4. How is a new coordinator chosen or elected?

Assumptions:

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

There are two types of Distributed Algorithms:

1. Bully Algorithm
2. Ring Algorithm

Bully Algorithm:

A. When a process, P, notices that the coordinator is no longer responding to requests, it initiates an election.

1. P sends an ELECTION message to all processes with higher numbers.
2. If no one responds, P wins the election and becomes a coordinator.
3. If one of the higher-ups answers, it takes over. P’s job is done.

B. When a process gets an ELECTION message from one of its lower-numbered colleagues:

1. Receiver sends an OK message back to the sender to indicate that he is alive and will take over.

2. Eventually, all processes give up apart of one, and that one is the new coordinator.

3. The new coordinator announces *its* victory by sending all processes a **CO-ORDINATOR** message telling them that it is the new coordinator.

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

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

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

Ring Algorithm

Initiation:

1. When a process notices that coordinator is not functioning:

2. Another process (initiator) initiates the election by sending "ELECTION" message (containing its own process number)

Leader Election:

3. Initiator sends the message to it's successor (if successor is down, sender skips over it and goes to the next member along the ring, or the one after that, until a running process is located).

4. At each step, sender adds its own process number to the list in the message.

5. When the message gets back to the process that started it all: Message comes back to initiator. In the queue the **process with maximum ID Number wins**.

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

Designing the solution: A. For Ring Algorithm

Initiation:

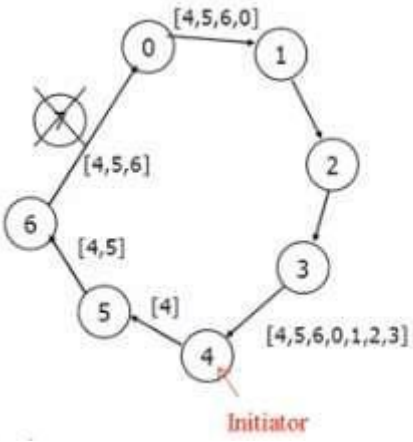
1. Consider the Process 4 understands that Process 7 is not responding.

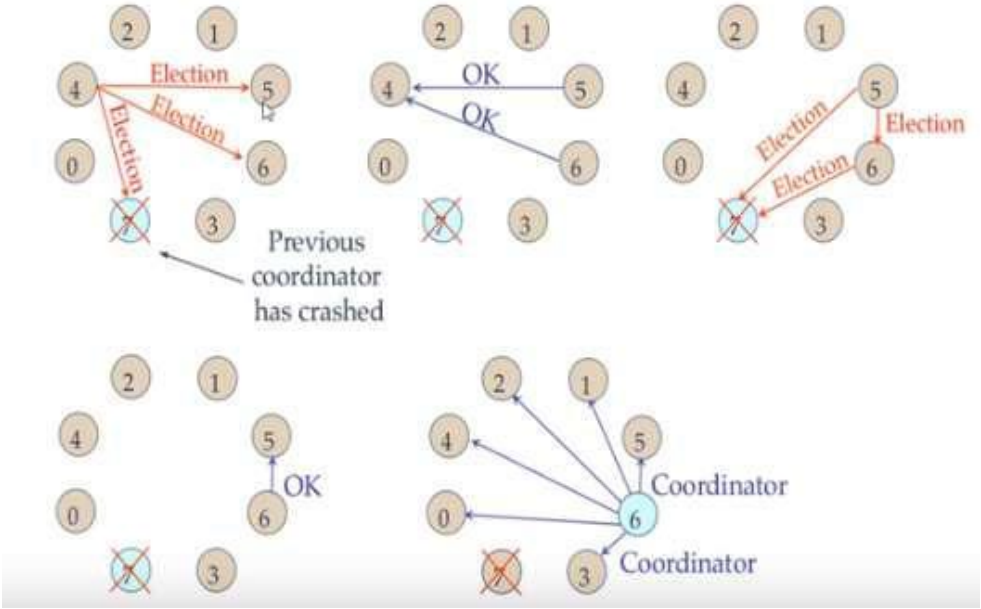
2.Process 4 initiates the Election by sending "ELECTION" message to it's successor (or next alive process) with it's ID.

Leader Election:

3.Messages comes back to initiator. Here the initiator is 4.

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





Implementing the solution:

For Ring Algorithm:

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

   ii) Index: Stores index of process.
   iii) ID: Process ID

2. Import Scanner Class for getting input from Console
3. Getting input from User for number of Processes and store them into object of classes.
4. Sort these objects on the basis of process id.
5. Make the last process id as "inactive".
6. Ask for menu 1.Election 2.Exit
7. Ask for initializing election process.

8.These inputs will be used by Ring Algorithm.

Conclusion:

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

Outcome:

1. Students learned the fundamentals of process coordinator election algorithms in DS
2. Students developed Bully and Ring algorithm for leader election

FAQ:

1. Who is process coordinator? What are its responsibilities?
2. Need of Election Algorithm?
3. What is centralized and decentralized algorithm?
4. Explain Election working of algorithm for Ring & Bully?
5. What is ‘Token’?
6. Why algorithm is known as “Bully”?

ASSIGNMENT NO. 7

Title: Election Algorithms

Bully.java

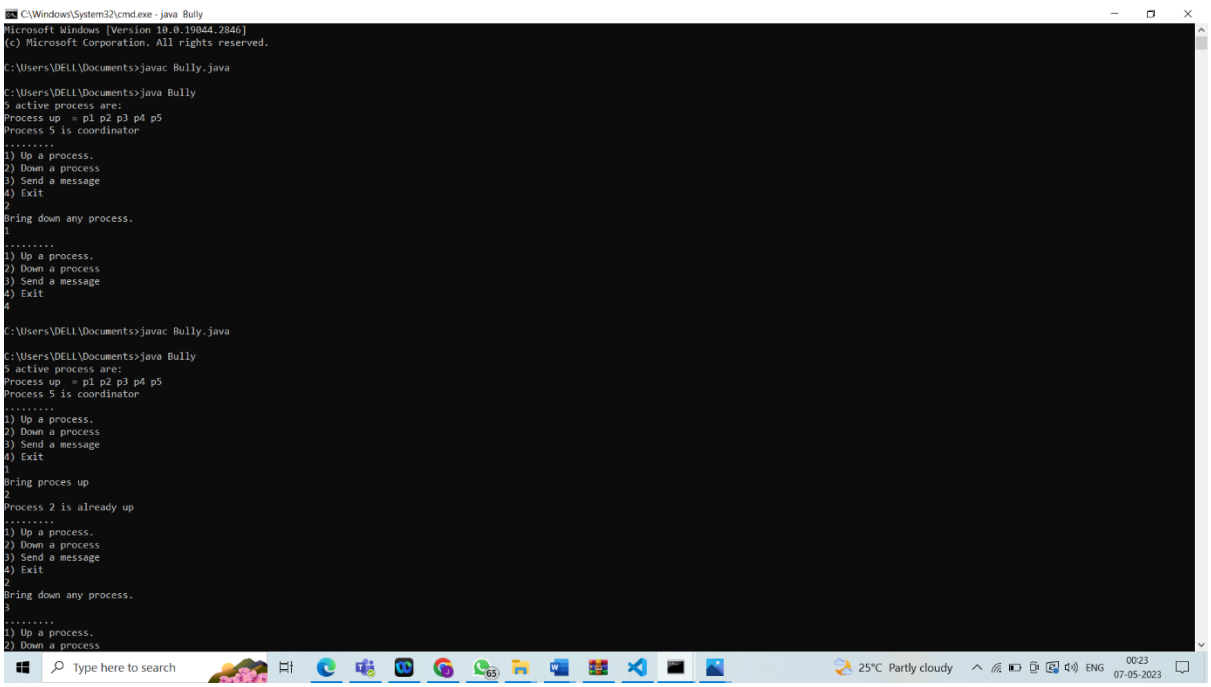
```
import java.io.InputStream;
import java.io.PrintStream;
import java.util.Scanner;

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

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

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

public static void mess(int mess) {
   if (state[mess - 1]) {
      if (state[4]) {
         System.out.println("0K");
      } else if (!state[4]) {
         int i;
         System.out.println("Process " + mess + " election");
         for (i = mess; i < 5; ++i) {
            System.out.println("Election send from process " + mess + " to process " + (i + 1));
         }
         for (i = 5; i >= mess; --i) {
            if (!state[i - 1]) continue;
            System.out.println("Coordinator message send from process " + i + " to all");
            break;
         }
      }
   } else {
      System.out.println("Process " + mess + " is down");
   }
}

public static void main(String[] args) {
```

```
int choice;
Scanner sc = new Scanner(System.in);
for (int i = 0; i < 5; ++i) {
   Bully.state[i] = true;
}
System.out.println("5 active process are:");
System.out.println("Process up  = p1 p2 p3 p4 p5");
System.out.println("Process 5 is coordinator");
do {
   System.out.println(".....");
   System.out.println("1) Up a process.");
   System.out.println("2) Down a process");
   System.out.println("3) Send a message");
   System.out.println("4) Exit");
   choice = sc.nextInt();
   switch (choice) {
      case 1: {
         System.out.println("Bring proces up");
         int up = sc.nextInt();
         if (up == 5) {
            System.out.println("Process 5 is co-ordinator");
            Bully.state[4] = true;
            break;
         }
         Bully.up(up);
         break;
      }
      case 2: {
         System.out.println("Bring down any process.");
         int down = sc.nextInt();
         Bully.down(down);
```

```
                break;
            }
            case 3: {
                System.out.println("Which process will send message");
                int mess = sc.nextInt();
                Bully.mess(mess);
            }
          }
      } while (choice != 4);
      sc.close();

   }
}
```

Output:



Ring.java

Code:

```
import java.util.Scanner;


public class Ring1 {


   public static void main(String[] args) {


      // TODO Auto-generated method stub


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


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


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


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

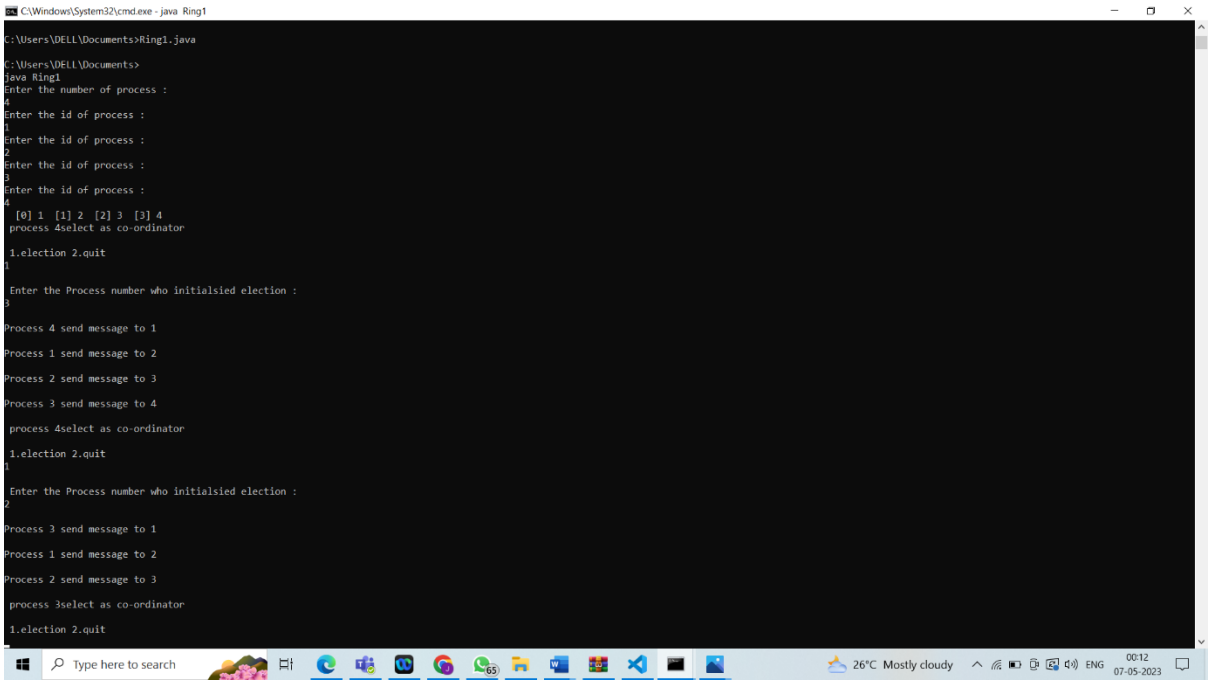
```
// sorting the processes from on the basis of id
      for (i = 0; i < num - 1; i++) {
         for (j = 0; j < num - 1; j++) {
            if (proc[j].id > proc[j + 1].id) {
               temp = proc[j].id;
               proc[j].id = proc[j + 1].id;
               proc[j + 1].id = temp;
     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

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



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

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

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

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

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

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

   i = 0;

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

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

   System.out.println("\nProcess " + proc[init].id + " send message to " + proc[temp1].id);
```

```
            arr[i] = proc[temp1].id;
            i++;
            int max = -1;

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

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

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

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

      }
   }
```

```
}

class Rr {

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

}
```

Output:



Assignment 7

Q1. Who is process coordinator? What are its responsibilities?

Ans: In a distributed system, a process coordinator is responsible for managing the processes running on multiple nodes and ensuring that they work together to achieve a common goal. The process coordinator may be a separate process or component that runs on a dedicated node and is responsible for tasks such as process scheduling, load balancing, data management, and communication coordination.

Q2. Need of Election Algorithm?

Ans: The need for an election algorithm arises in distributed systems when there is a need to elect a leader or coordinator node among a group of nodes. This may be necessary for tasks such as resource allocation, process coordination, or handling failures in the system. The election algorithm ensures that only one node is elected as the leader at any given time, and that the election process is fair and reliable.

Q3. What is centralized and decentralized algorithm?

Ans: In a centralized algorithm, all nodes in the system communicate with a central node or coordinator to perform tasks such as resource allocation or decision-making. In a decentralized algorithm, nodes communicate with each other directly and make decisions based on local information and consensus. Decentralized algorithms are often more robust and scalable than centralized algorithms, but may be more complex to implement.

Q4. Explain Election working of algorithm for Ring & Bully?

Ans: Election algorithms are used to elect a leader among a group of nodes in a distributed system. In the ring algorithm, nodes are arranged in a ring topology, and each node sends an election message to its successor in the ring. The node with the highest ID responds to the election message, and the process continues until only one node remains as the leader. In the bully algorithm, each node with a higher ID sends an election message to all lower-ID nodes. The lowest-ID node responds and becomes the leader. If a higher-ID node does not receive a response, it assumes that the lower-ID node has failed and takes over as the leader.

Q5. What is ‘Token’?

Ans: In some election algorithms, such as the token ring algorithm, a special message called a token is passed between nodes to determine the leader. The token is passed around the ring, and the node that holds the token at a given time becomes the leader. The token ensures that only one node can become the leader at any given time.

Q6. Why algorithm is known as “Bully”?

Ans: The bully algorithm is called the "bully" algorithm because nodes with higher IDs "bully" lower-ID nodes to determine the leader. The algorithm ensures that the node with the highest ID always becomes the leader, even if other nodes do not respond or are unreachable.

ASSIGNMENT NO. 8

Title: Web Services

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

Objective:

1. To understands the fundaments of web services, architecture and its types.
2. To create a simple web service.

Tools / Environment:

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

Related Theory:

Web Service:

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

A service can be treated as a web service if:

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

Types of Web Services:

There are two types of web services:

1. **SOAP**: SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services. Since it's XML based, it's

platform and language independent. So, our server can be based on JAVA and client can be on NET, PHP etc. and vice versa.

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

Web service architectures:

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

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

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

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

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

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

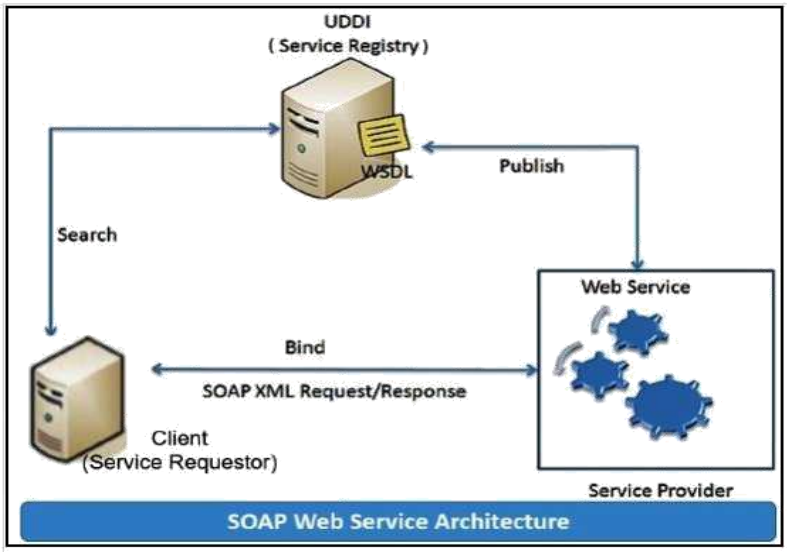
SOAP web services

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

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

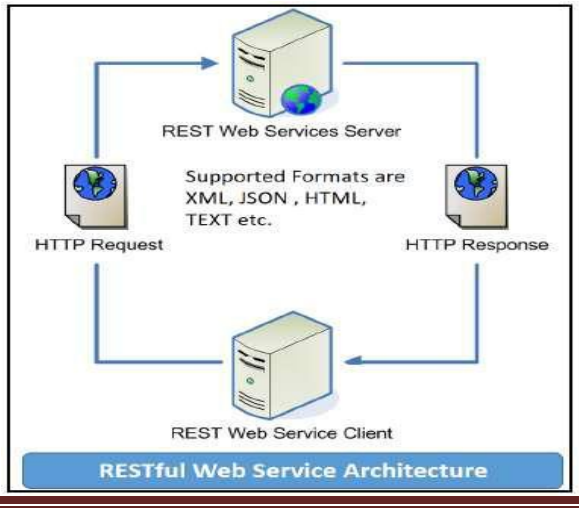
Universal Description, Discovery, and Integration (**UDDI**)*:* UDDI is an XMLbased framework for describing, discovering, and integrating web services. It acts as a directory of web service interfaces described in the WSDL language.

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 a protocol. Refer to the following diagram:



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

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

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.

Outcome:

1. Students understand the concept of web services.
2. Students developed web service and write any distributed application to consume the web service.

FAQ

1. What Is a Web Service?
2. Explain Architecture of web services w. r. to Provider, Requestor, Service registry and Broker?
3. What is WSDL?
4. List types of Web services?
5. Differentiate between SOAP and REST?
6. List the examples of web services?
7. List applications of web services?

ASSIGNMENT NO. 8

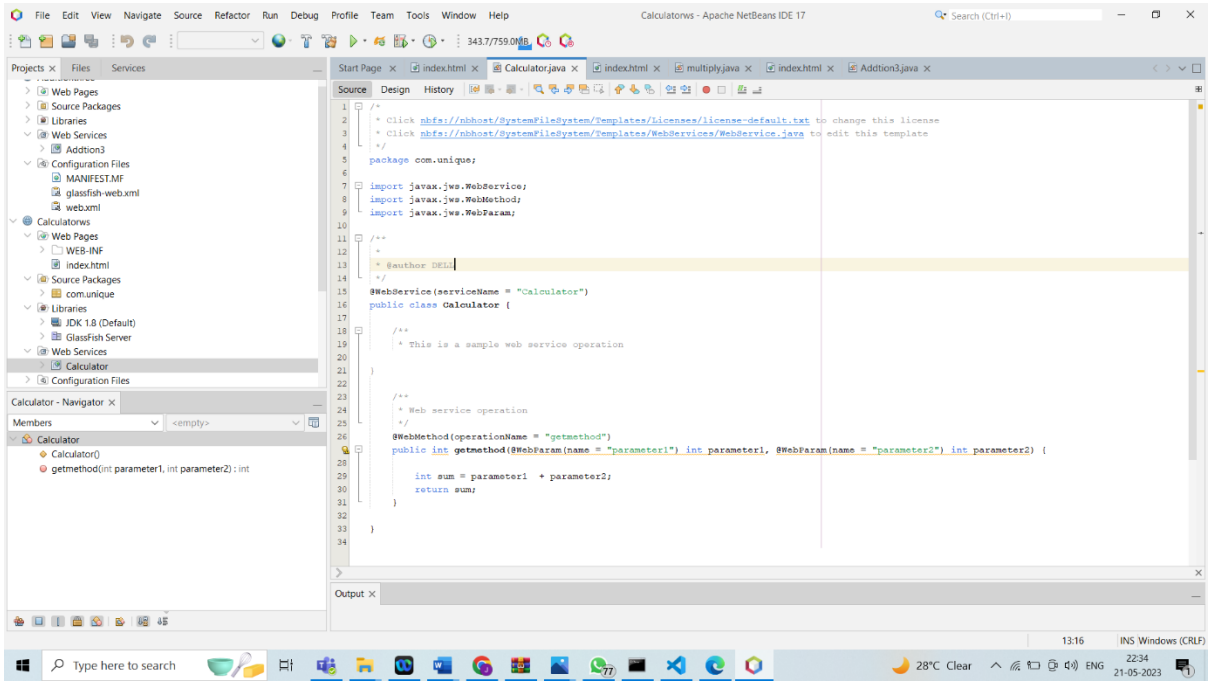
Title: Web Services

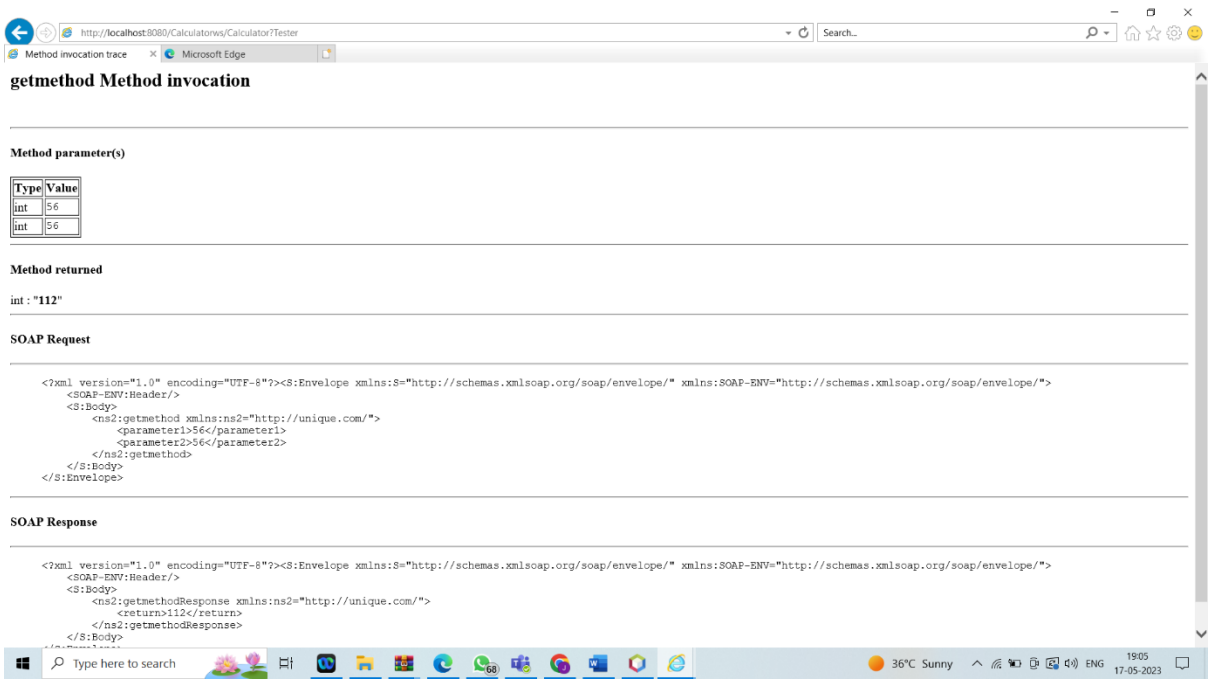
Calculator.java

package com.unique;

```
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

/**
 *
 * @author DELL
 */
@WebService(serviceName = "Calculator")
public class Calculator {

    /**
     * This is a sample web service operation

}

    /**
     * Web service operation
     */
    @WebMethod(operationName = "getmethod")
    public int getmethod(@WebParam(name = "parameter1") int parameter1, @WebParam(name =
"parameter2") int parameter2) {

        int sum = parameter1  + parameter2;
        return sum;
    }
    }
```

Output:


File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Calculatorws - Apache NetBeans IDE 17
Projects
Files
Services
Web Pages
Source Packages
Libraries
Web Services
Addtion3
Configuration Files
MANIFEST.MF
glassfish-web.xml
web.xml
Calculatorws
Web Pages
WEB-INF
index.html
Source Packages
com.unique
Libraries
JDK 1.8 (Default)
GlassFish Server
Web Services
Calculator
Configuration Files
Calculator - Navigator
Members
Calculator()
getmethod(int parameter1, int parameter2) : int
Start Page
index.html
Calculator.java
index.html
multiply.java
index.html
Addtion3.java
Source
Design
History
package com.unique;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
@WebService(serviceName = "Calculator")
public class Calculator {
/**
* This is a sample web service operation
*/
@WebMethod(operationName = "getmethod")
public int getmethod(@WebParam(name = "parameter1") int parameter1, @WebParam(name = "parameter2") int parameter2) {
int sum = parameter1 + parameter2;
return sum;
}
}
Output
13:16
INS Windows (CRLF)
Type here to search
28°C Clear
ENG
22:34
21-05-2023


http://localhost:8080/Calculatorws/Calculator?Tester
Search...
Method invocation trace
Microsoft Edge
getmethod Method invocation
Method parameter(s)
Type
Value
int
56
int
56
Method returned
int : "112"
SOAP Request
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<S:Body>
<ns2:getmethod xmlns:ns2="http://unique.com/">
<parameter1>56</parameter1>
<parameter2>56</parameter2>
</ns2:getmethod>
</S:Body>
</S:Envelope>
SOAP Response
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<S:Body>
<ns2:getmethodResponse xmlns:ns2="http://unique.com/">
<return>112</return>
</ns2:getmethodResponse>
</S:Body>
Type here to search
36°C Sunny
ENG
19:05
17-05-2023

••

Assignment 8

Q1. What Is a Web Service?

Ans: A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It provides a standardized way for different applications to exchange data and communicate with each other over the internet using open protocols such as HTTP, XML, and SOAP.

Q2. Explain Architecture of web services w. r. to Provider, Requestor, Service registry and Broker?

Ans: The architecture of web services involves several key components, including the service provider, the service requestor, the service registry, and the service broker. The service provider is responsible for publishing the service and making it available to potential requestors. The service requestor is the client application that consumes the service. The service registry is a directory of available services that provides metadata about the services, such as their endpoints, WSDL, and other information. The service broker acts as a mediator between the service provider and the requestor, enabling the two to communicate with each other in a standardized way.

Q3. What is WSDL?

Ans: WSDL (Web Services Description Language) is an XML-based language used to describe the interface of a web service. It specifies the methods that the service exposes, the parameters that are required for each method, and the format of the data that is exchanged between the service provider and the requestor.

Q4. List types of Web services?

Ans: There are two main types of web services: SOAP (Simple Object Access Protocol) and REST (Representational State Transfer). SOAP web services use a standardized XML-based messaging protocol to exchange data between the service provider and the requestor, while REST web services use a more lightweight architecture based on HTTP and other web standards.

Q5. Differentiate between SOAP and REST?

Ans: The main differences between SOAP and REST web services are that SOAP is more complex and heavyweight, while REST is simpler and more lightweight. SOAP requires a dedicated messaging protocol and XML-based message formats, while REST uses standard HTTP methods and lightweight data formats such as JSON.

Q6. List the examples of web services?

Ans: Examples of web services include weather APIs, financial APIs, social media APIs, and many others. Some popular web services include Amazon Web Services, Google Maps API, and Twitter API.

Q7. List applications of web services?

Ans: Web services are used in a wide range of applications, including e-commerce, social media, financial services, healthcare, and many others. They provide a standardized way for different applications to communicate with each other, enabling businesses to share data and services across different platforms and technologies.