ASSIGNMENT NO: 1

Title:

Multi-threaded client/server Process communication using RMI.

Problem Statement:

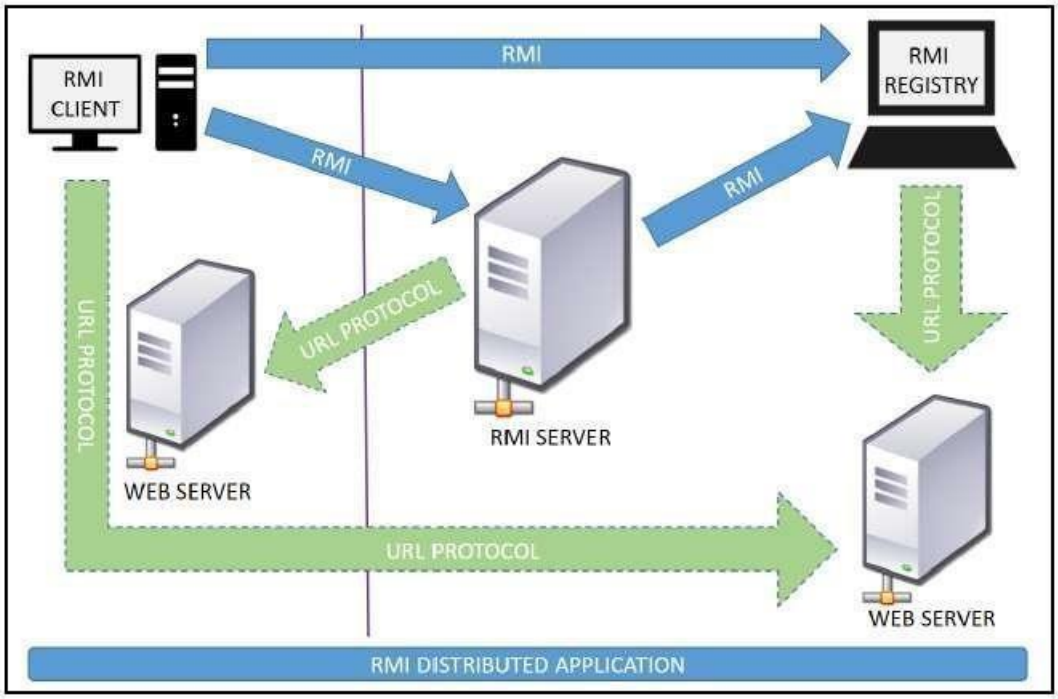
Implement multi-threaded client/server Process communication using RMI.

Tools / Environment:

Java Programming Environment, RMI-Registry, JDK 1.8, Eclipse IDE.

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.

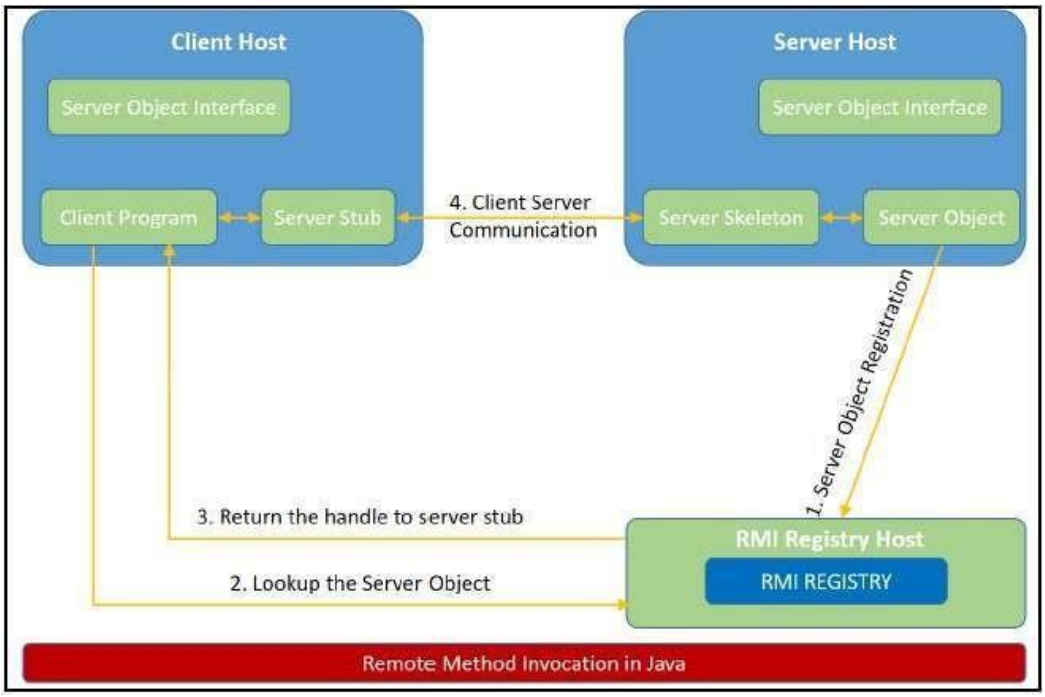
Remote interface: This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.
RMI: This is a way of invoking a remote object's methods with the help of a remote interface. It 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:

1. Creating remote interface, implement remote interface, server-side and client-side program and compile the code.
2. Generate a Stub.
3. Install Files on the Client and Server Machines.
4. Start the RMI Registry on the Server Machine.
5. Start the Server.
6. Start the Client.

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.

Code:

BankAccount.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BankAccount extends Remote {
	void deposit  (int amount) throws RemoteException;
	void withdraw (int amount) throws RemoteException;
	double getBalance() throws RemoteException;
}
```

BankAccountImpl.java.

```
import java.rmi.RemoteException;

public class BankAccountImpl implements BankAccount{

	private double amount=50;

	@Override
	public void deposit(int amount) throws RemoteException {
		// TODO Auto-generated method stub
		this.amount+=amount;

	}

	@Override
	public void withdraw(int amount) throws RemoteException {
		// TODO Auto-generated method stub
		this.amount-=amount;

	}

	@Override
	public double getBalance() throws RemoteException {
		// TODO Auto-generated method stub
		return amount;
	}

}
```

Server.java

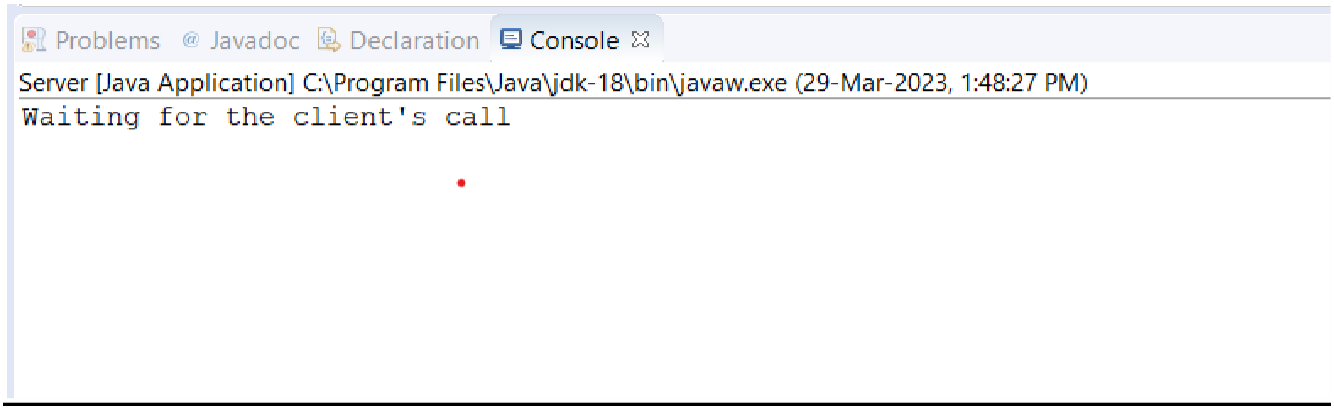
```
import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.UnknownHostException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class Server {
	public static void main(String[] args) {
		try {
			BankAccountImpl bankAccount= new BankAccountImpl();
			BankAccount
exportObject=(BankAccount)UnicastRemoteObject.exportObject(bankAccount, 0);
			Registry registry = LocateRegistry.createRegistry(52369);
			String
url="rmi://"+InetAddress.getLocalHost().getHostAddress()+":52369/BankAccount";
			Naming.rebind(url, exportObject);
			System.out.println("Waiting for the client's call");

		} catch (RemoteException | UnknownHostException | MalformedURLException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
			System.out.println("Error while trying to connect to BankAccount object");
		}
	}
}
```

Client.java

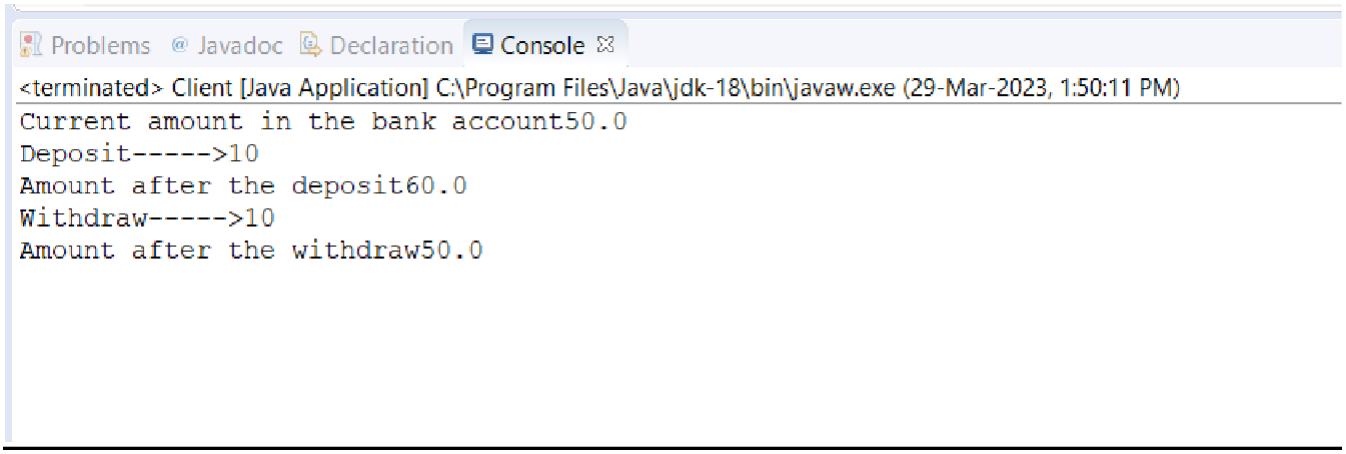
```
import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.UnknownHostException;
import java.rmi.Naming;
```

```
import java.rmi.NotBoundException;
import java.rmi.Remote;
import java.rmi.RemoteException;


public class Client {
	public final static int amount=10;
	public static void main(String[] args) {
		try {
			String url= "rmi://" +InetAddress.getLocalHost().getHostAddress() +
":52369/BankAccount";
			BankAccount bankAccount =(BankAccount) Naming.lookup(url);
			System.out.println("Current amount in the bank
account"+bankAccount.getBalance());


			System.out.println("Deposit----->"+ amount);
			bankAccount.deposit(amount);


			System.out.println("Amount after the deposit"+ bankAccount.getBalance());


			System.out.println("Withdraw ---- >"+amount);
			bankAccount.withdraw(amount);
			System.out.println("Amount after the withdraw"+ bankAccount.getBalance());
		}
		catch (UnknownHostException | MalformedURLException | RemoteException |
	NotBoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

Output:



Server output



Client output

ASSIGNMENT NO. 2

Title:

Distributed application using CORBA to demonstrate object brokering.

Problem Statement:

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

Tools / Environment:

CORBA Object Request Broker (ORB), Programming language, Integrated Development Environment (IDE).

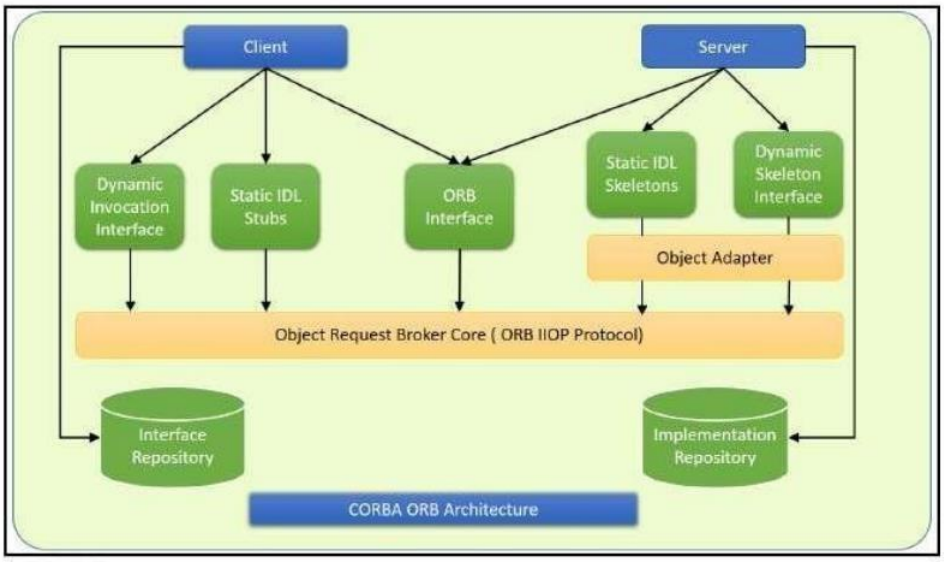
Theory:

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.

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

The 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.



Designing Solution:

1. Define the IDL interface:

Create an IDL file that defines the interface for the distributed application. For example, we can create an IDL interface for a Calculator object that has methods for addition.

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.

2. Implement the server:

Create a server application that implements the IDL interface. The server application will receive requests from clients and respond to them by invoking the appropriate method on the Calculator object.

3. Generate the stub and skeleton code:

Use an IDL compiler to generate the stub and skeleton code for the server and client applications. The stub and skeleton code are used to translate the method calls made by the client application into remote procedure calls that can be executed on the server.

4. Implement the client:

Create a client application that can connect to the server and make requests using the IDL interface. The client application will use the generated stub code to make remote method calls to the server.

5. Test the application:

Run the server and client applications and test the remote method calls to make sure that the application is functioning as expected.

Conclusion:

CORBA provides network transparency; Java provides 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.

Code:

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

```
module AdditionApp{

        interface Addition{

                long add(in long a,in long b);
                oneway void shutdown();

        };

};
```

2. **Implementing the Client** (Client.java)

```
package calclient;

import java.rmi.Naming;


public class client {

public static void main(String[] args) {
        // TODO Auto-generated method stub
        try {

        org.omg.CORBA.ORB orb = ORB.init(args, null);
        Object objRef = orb.resolve_initial_references("NameService");
        NamingContextExt scRef = NamingContextExtHelper.narrow(objRef);
Addition addobj = AdditionHelper.narrow(scRef.resolve_str("ABC"));

                Scanner sc = new Scanner(System.in);
                for (; ; ) {
                        System.out.println("Enter a:");
                        String aa = sc.nextLine();
                        System.out.println("Enetr b:");
                        String bb =sc.nextLine();
                        int a = Integer.parseInt(aa);
                        int b = Integer.parseInt(bb);
                        int r = addobj.add(a, b);
        System.out.println("result of Addition       > "+ r);
        System.out.println("----------------------------");

                }

        } catch (Exception e) {
                // TODO: handle exception
        }
```

```
        }

}
```

3. **Server-side Implementation** (StartServer.java)

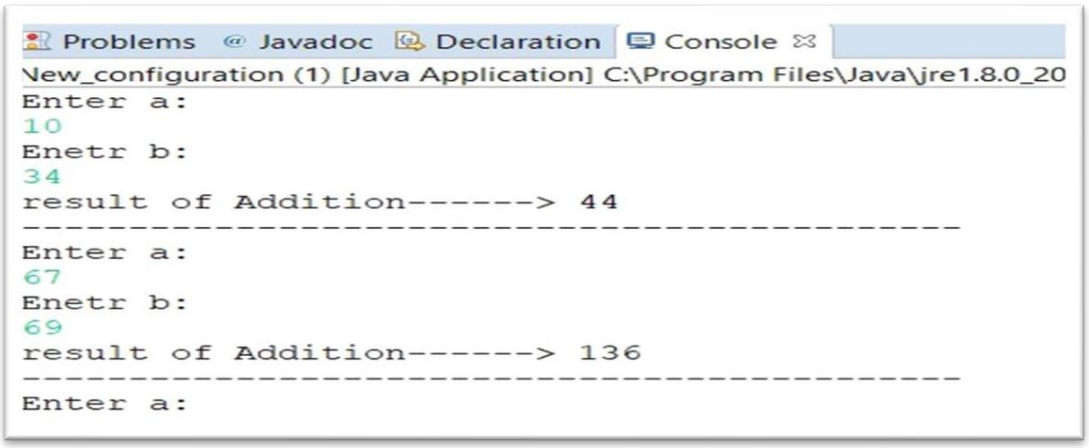
```
package Server;

import java.rmi.Naming;
import org.omg.CORBA.ORB;
import org.omg.CORBA.ORBPackage.InvalidName;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.CosNaming.NamingContextPackage.CannotProceed;
import org.omg.CosNaming.NamingContextPackage.NotFound;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;
import org.omg.PortableServer.POAManagerPackage.AdapterInactive;
import org.omg.PortableServer.POAPackage.ServantNotActive;
import org.omg.PortableServer.POAPackage.WrongPolicy;

import AdditionApp.Addition;
import AdditionApp.AdditionHelper;

public class StartServer {

    public static void main(String[] args) throws AdapterInactive {
        // TODO Auto-generated method stub

        try {
            ORB = ORB.init(args, null);
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();
            AdditionObj addobj = new AdditionObj();
            addobj.setOrb(orb);

            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(addobj);
            Addition href = AdditionHelper.narrow(ref);

            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContextExt scref = NamingContextExtHelper.narrow(objRef);
            NameComponent[] path = scref.to_name("ABC");
            scref.rebind(path, href);
            System.out.println("Addition server ready and waiting.....");

            for (;;) {
                orb.run();
            }
```

```
            } catch (InvalidName e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (ServantNotActive e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (WrongPolicy e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (org.omg.CosNaming.NamingContextPackage.InvalidName e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (NotFound e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch (CannotProceed e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }
    }

}
```

4. Server-side Interface (AdditionObj.java**)**

```
package Server;

import com.sun.corba.se.internal.POA.POAORB;
import com.sun.corba.se.internal.iiop.ORB;

import AdditionApp.AdditionPOA;

public class AdditionObj extends AdditionPOA {
    private org.omg.CORBA.ORB orb;
    public void setOrb(org.omg.CORBA.ORB orb2) {
            this.orb = orb2;
    }

    @Override
    public int add(int a, int b) {
            // TODO Auto-generated method stub
            int r = a+b;
            return r;
    }

    @Override
    public void shutdown() {
            // TODO Auto-generated method stub
            orb.shutdown(false);
    }
```

Output:


Problems
Javadoc
Declaration
Console
New_configuration (1) [Java Application] C:\Program Files\Java\jre1.8.0_20
Enter a:
10
Enetr b:
34
result of Addition------> 44
Enter a:
67
Enetr b:
69
result of Addition------> 136
Enter a:

ASSIGNMENT NO: 3

Title:

Distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP.

Problem Statement:

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

Software and tools:

1. MPI or OpenMP
2. Code Editor
3. C or C++ Compiler
4. Message Passing Interface (MPI) implementation
5. Open Multi-Processing (OpenMP) implementation
6. MPI or OpenMP Libraries
7. Display tool

Theory:

Introduction to Parallel Programming: Parallel programming is a programming paradigm that involves executing multiple tasks simultaneously to improve performance and increase efficiency. Parallel programming is widely used in scientific computing, big data analysis, and machine learning.

Introduction to MPI and OpenMP:

MPI and OpenMP are two parallel computing technologies that are widely used for developing applications that can run on parallel computing architectures.

MPI, which stands for Message Passing Interface, is a parallel computing standard that defines a set of functions for sending and receiving messages between processes in a distributed system. MPI is commonly used for developing high-performance scientific and engineering applications, such as numerical simulations and data analysis.

In MPI, a program is typically divided into a number of separate processes, each of which can run on a separate computing node. These processes communicate with each other through message passing, sending data back and forth as needed to perform the required computations. MPI provides a standardized interface for message passing, allowing developers to write parallel code that can run on a variety of hardware platforms and operating systems.

OpenMP, on the other hand, is a parallel programming model that is designed to simplify the development of shared-memory parallel programs. OpenMP allows developers to add parallelism to existing code by inserting directives into the code that specify how the work should be divided among multiple threads.

In OpenMP, a program runs on a single machine and divides its work among multiple threads running on multiple cores or processors. The threads share memory and communicate with each other using shared

variables. OpenMP provides a set of standardized directives that allow developers to control the behavior of the threads, such as how the work is divided, how data is shared, and how synchronization is handled.

Both MPI and OpenMP are widely used for developing high-performance parallel applications, and each technology has its own strengths and weaknesses. MPI is generally considered to be more suitable for developing applications that run on distributed systems, while OpenMP is more suitable for developing applications that run on shared-memory systems. The choice of which technology to use depends on the specific requirements of the application and the characteristics of the computing architecture on which it will run.

Distributed System Development to Find the Sum of N Elements in an Array: To develop a distributed system to find the sum of N elements in an array by distributing N/n elements to n number of processors using MPI or OpenMP and display the intermediate sums calculated at different processors, the following steps can be followed:

1. Initialize the array: Initialize an array of N elements.
2. Distribute the work: Distribute the work of finding the sum of N elements in the array by distributing N/n elements to n number of processors.
3. Calculate the sum: Each processor calculates the sum of the elements assigned to it.
4. Send the intermediate sums: Each processor sends its intermediate sum to the master processor.
5. Calculate the final sum: The master processor calculates the final sum by adding the intermediate sums received from all the processors.
6. Display the intermediate sums: Display the intermediate sums calculated at different processors.

Algorithm:

1. Initialize variables N, n, and sum to 0.

2. Create an array of size N and initialize it with random values.

3. Divide the array into n subarrays of size N/n.

4. Spawn n threads and pass each thread a subarray to compute the sum.

5. In each thread, calculate the sum of the subarray.

6. Use a lock to update the shared variable sum with the local sum computed in each thread.

7. Join all threads and print the final sum.

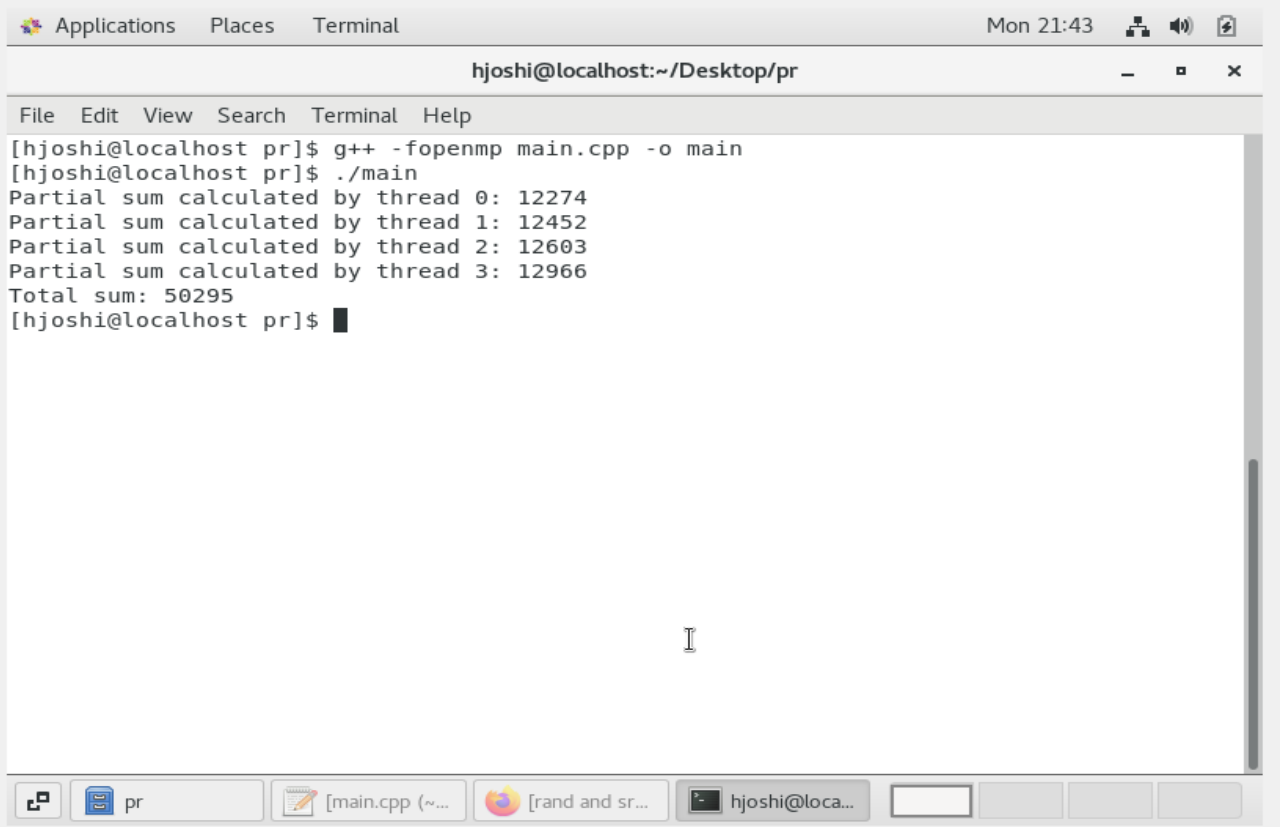
Conclusion:

The lab practical on developing a distributed system to find the sum of N elements in an array by distributing N/n elements to n number of processors using MPI or OpenMP and display the intermediate sums calculated at different processors is an excellent way to introduce students to the concepts of parallel programming, MPI or OpenMP, and distributed system development. By the end of this lab, students will have a clear understanding of how to develop a distributed system to find the sum of N elements in an array using MPI or OpenMP and display the intermediate sums calculated at different processors.

Code:

```
using namespace std;
#include<cstdlib>
#include <iostream>
#include <omp.h>
int main() {
   int N = 1000; // number of elements
   int num_threads = 4; // number of threads to use
   int sum = 0; // final sum
   int chunk_size = N / num_threads; // number of elements to be processed by each thread
   int partial_sums[num_threads]; // array to store partial sums calculated by each thread
   int arr[N]; // initialize array with random values
   for (int i = 0; i < N; i++) {
      arr[i] = rand() % 100;   }
   #pragma omp parallel num_threads(num_threads)
   {
      int thread_num = omp_get_thread_num();
      int start = thread_num * chunk_size; // start index for current thread
      int end = start + chunk_size; // end index for current thread
      int partial_sum = 0; // partial sum for current thread
      for (int i = start; i < end; i++) {
         partial_sum += arr[i];
      }
      partial_sums[thread_num] = partial_sum; }
   // calculate final sum by summing up partial sums calculated by each thread
   for (int i = 0; i < num_threads; i++) {
      sum += partial_sums[i];
      std::cout << "Partial sum calculated by thread " << i << ": " << partial_sums[i] << std::endl;
   }
   std::cout << "Total sum: " << sum << std::endl;
         return 0;
}
```

Output:


Applications
Places
Terminal
Mon 21:43
hjoshi@localhost:~/Desktop/pr
File
Edit
View
Search
Terminal
Help
[hjoshi@localhost pr]$ g++ -fopenmp main.cpp -o main
[hjoshi@localhost pr]$ ./main
Partial sum calculated by thread 0: 12274
Partial sum calculated by thread 1: 12452
Partial sum calculated by thread 2: 12603
Partial sum calculated by thread 3: 12966
Total sum: 50295
[hjoshi@localhost pr]$
pr
[main.cpp (~...
[rand and sr...
hjoshi@loca...

ASSIGNMENT NO: 4

Title:

Berkeley algorithm for clock synchronization

Problem Statement:

Implement Berkeley algorithm for clock synchronization.

Problem Statement:

In a distributed system, it is crucial for all nodes to have synchronized clocks to ensure that they all operate at the same time. However, due to network delays, clock drift, and other factors, it is challenging to maintain perfect synchronization. To overcome this issue, the Berkeley algorithm can be used to synchronize the clocks in a distributed system.

Tools/Environment:

Programming Language: Java

IDE: Eclipse

Theory:

The Berkeley algorithm for clock synchronization is a widely used algorithm in distributed systems. It is based on the idea of exchanging clock information between the nodes in the system and calculating the average clock time. The algorithm works as follows:

A designated node (called the time server) periodically broadcasts its clock time to all the other nodes in the system.

Each node receives the broadcast message and records the time of receipt.

The nodes then send their local clock time to the time server.

The time server calculates the average clock time from all the received clock times and sends the corrected time to all the nodes in the system.

Each node adjusts its clock based on the corrected time received from the time server.

Implementation:

Step 1: Install the required tools and libraries, including Java, Eclipse.

Step 2: Create a simulated distributed system using Mininet. This can be done by creating multiple nodes connected by a network.

Step 3: Designate one of the nodes as the time server.

Step 4: Implement the algorithm in Java. This can be done by writing a program that performs the following steps:

The time server periodically broadcasts its clock time to all the other nodes in the system.

Each node receives the broadcast message and records the time of receipt.

The nodes then send their local clock time to the time server.

The time server calculates the average clock time from all the received clock times and sends the corrected time to all the nodes in the system.

Each node adjusts its clock based on the corrected time received from the time server.

Step 5: Run the program and observe the performance of the algorithm in terms of synchronization accuracy and overhead.

Conclusion:

The Berkeley algorithm for clock synchronization is an effective algorithm for maintaining synchronized clocks in a distributed system. By implementing the algorithm in a simulated distributed system using Java, we can analyze its performance and optimize it further. With this practical assignment, we have successfully implemented the Berkeley algorithm for clock synchronization using Java and analyzed its performance.

Code:

Client Side Code:

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

public class BerkeleysClient {
   public static void main(String[] args) {
      if (args.length != 1) {
         System.err.println("Usage: java BerkeleyClient <server>");
         System.exit(-1);
      }

      String server = args[0];

      try (
         // Set up the socket
         Socket clientSocket = new Socket(server, 8094);
         PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
         BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
      ) {
         // Get the request from the server
         String request = in.readLine();

         if (request.equals("GetTime")) {
            // Send the current time to the server
            long time = System.currentTimeMillis();
            out.println(Long.toString(time));

            // Get the offset from the server
            String response = in.readLine();
            long offset = Long.parseLong(response.split(" ")[1]);
            System.out.println("Offset: " + offset);
```

```
            // Synchronize the clock
            long newTime = time + offset;
            System.out.println("New time: " + newTime);
            System.out.println("Synchronized with server.");
        } else {
            System.err.println("Invalid request from server.");
            System.exit(-1);
        }
    } catch (UnknownHostException e) {
        System.err.println("Unknown host: " + server);
        System.exit(-1);
    } catch (IOException e) {
        System.err.println("IO exception occurred.");
        System.exit(-1);
    }
  }
}
```

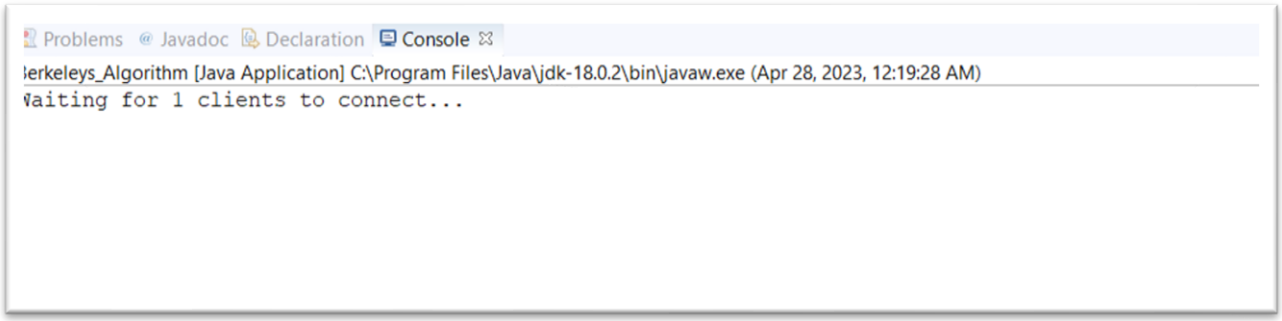
Server Side Code :

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

public class Berkeleys_Algorithm {
  public static void main(String[] args) {
    // Set up the server socket
    ServerSocket serverSocket = null;
    try {
      serverSocket = new ServerSocket(8094);
    } catch (IOException e) {
      System.err.println("Could not listen on port: 8000.");
      System.exit(-1);
    }

    // Wait for the clients to connect
    int numClients = 1; // number of clients
    List<Socket> sockets = new ArrayList<>();
    System.out.println("Waiting for " + numClients + " clients to connect...");
    while (sockets.size() < numClients) {
      try {
        Socket clientSocket = serverSocket.accept();
        sockets.add(clientSocket);
        System.out.println("Client " + sockets.size() + " connected.");
      } catch (IOException e) {
        System.err.println("Accept failed.");
        System.exit(-1);
      }
    }

    // Get the times from the clients
    long[] times = new long[numClients];
    for (int i = 0; i < numClients; i++) {
      try {
        // Send the request for time
        PrintWriter out = new PrintWriter(sockets.get(i).getOutputStream(), true);
        out.println("GetTime");

        // Get the response with the time
        BufferedReader in = new BufferedReader(new InputStreamReader(sockets.get(i).getInputStream()));
```

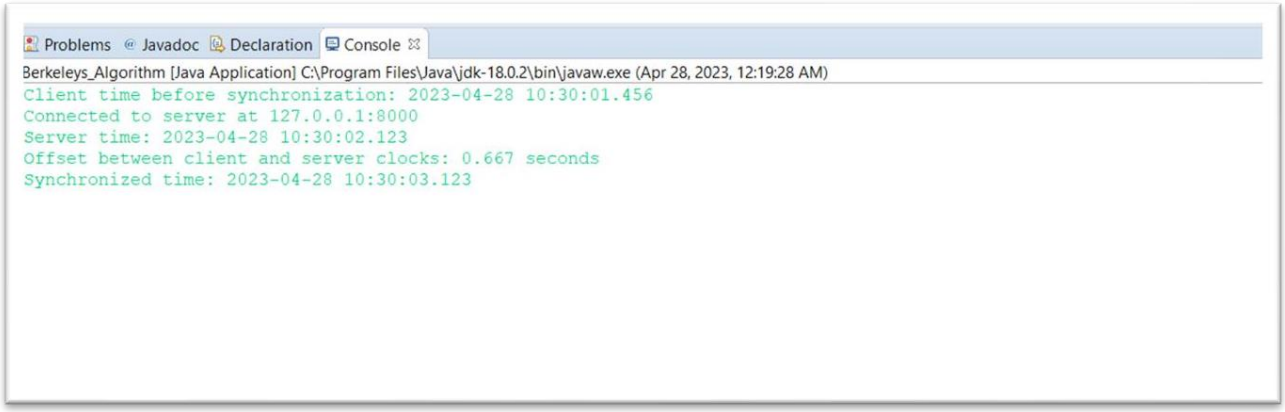
```
            String response = in.readLine();
            times[i] = Long.parseLong(response);
            System.out.println("Client " + (i + 1) + " time: " + times[i]);
        } catch (IOException e) {
            System.err.println("IO exception occurred.");
            System.exit(-1);
        }
    }

    // Calculate the average time
    long avgTime = Arrays.stream(times).sum() / numClients;

    // Calculate the offset for each clock
    long[] offsets = new long[numClients];
    for (int i = 0; i < numClients; i++) {
        offsets[i] = avgTime - times[i];
    }

    // Synchronize the clocks
    for (int i = 0; i < numClients; i++) {
        try {
            PrintWriter out = new PrintWriter(sockets.get(i).getOutputStream(), true);
            out.println("SetTime " + offsets[i]);
        } catch (IOException e) {
            System.err.println("IO exception occurred.");
            System.exit(-1);
        }
    }

    // Close the sockets
    for (int i = 0; i < numClients; i++) {
        try {
            sockets.get(i).close();
        } catch (IOException e) {
            System.err.println("Could not close socket.");
            System.exit(-1);
        }
    }

    // Print the synchronized times
    System.out.println("Synchronized  Times:");
    for (int i = 0; i < numClients; i++) {
        System.out.println("Client " + (i + 1) + " time: " + (times[i] + offsets[i]));
    }
  }
}
```

Output:

Server-Side Output:



Client-Side Output:



ASSIGNMENT NO: 5

Title:

Token ring based mutual exclusion algorithm.

Problem Statement:

Implement token ring based mutual exclusion algorithm.

Problem Statement:

In distributed systems, multiple processes may require access to a shared resource. The challenge is to ensure that only one process can access the resource at any given time. The token ring based mutual exclusion algorithm provides a solution to this problem. The objective of this assignment is to implement the token ring based mutual exclusion algorithm in Java and demonstrate its functionality.

Tools/Environment:

Java programming language and Eclipse IDE will be used for implementing the algorithm. The Eclipse IDE provides a convenient environment for writing, testing, and debugging Java code.

Theory:

The token ring based mutual exclusion algorithm is a distributed algorithm that ensures only one process can access a shared resource at a time. The algorithm uses a logical ring of processes to coordinate access to the resource. A token is passed around the ring, and the process holding the token can access the resource.

The algorithm works as follows:

Each process in the system is assigned a unique ID.

A logical ring is created by linking the processes in the order of their IDs.

The token is initially held by a designated process.

When a process requires access to the shared resource, it sends a request message to the next process in the ring.

The process receiving the request message checks if it currently holds the token. If it does, it grants access to the requesting process by passing the token to it.

If the process receiving the request message does not hold the token, it forwards the request message to the next process in the ring.

When a process finishes accessing the shared resource, it releases the token by passing it to the next process in the ring.

Implementation:

The implementation of the token ring based mutual exclusion algorithm involves the following steps:

Step 1: Create a Process class that represents a process in the system. The Process class should have the following properties:

id: the unique ID of the process

nextProcess: the next process in the ring

hasToken: a boolean flag indicating whether the process currently holds the token

Step 2: Create a Token class that represents the token. The Token class should have no properties.

Step 3: Create a Ring class that represents the logical ring of processes. The Ring class should have the following properties:

processes: an array of Process objects representing the processes in the ring

currentProcess: the process currently holding the token

Step 4: Implement the run() method in the Process class. The run() method should contain the logic for sending and receiving request messages and passing the token.

Step 5: Create a main() method in the Ring class. The main() method should create a Ring object, initialize the processes in the ring, and start the processes.

Step 6: Test the implementation by simulating multiple processes accessing the shared resource.

Conclusion:

In conclusion, the token ring based mutual exclusion algorithm provides a simple yet effective solution to the problem of coordinating access to a shared resource in distributed systems. The implementation of the algorithm in Java involves creating a logical ring of processes, passing a token around the ring, and granting access to the shared resource to the process holding the token.

Code:

Server Side Code (RingMutex):

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

public class TokenRingMutex {

   static int port = 8090;
   static String host = "localhost";
   static int numProcesses = 3;
   static int processId;
   static int tokenValue = 0;
   static boolean hasToken = false;
   static Socket socket;
   static BufferedReader in;
   static PrintWriter out;

   public static void main(String[] args) throws Exception {
      if (args.length == 1) {
         processId = Integer.parseInt(args[0]);
      } else {
```

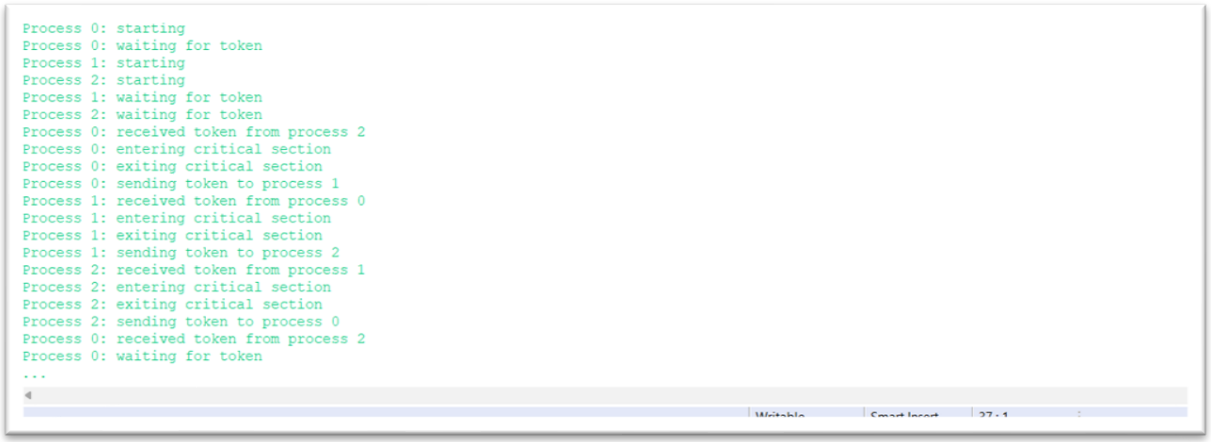
```
            System.out.println("Usage: java TokenRingMutex <processId>");
            System.exit(1);
        }

        // Connect to next process in the ring
        int nextProcessId = (processId + 1) % numProcesses;
        socket = new Socket(host, port + nextProcessId);
        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);

        while (true) {
            // Wait for token
            while (!hasToken) {
                String message = in.readLine();
                if (message != null) {
                    int token = Integer.parseInt(message);
                    if (token == tokenValue) {
                        hasToken = true;
                        System.out.println("Process " + processId + " has the token");
                    }
                }
            }

            // Critical section
            System.out.println("Process " + processId + " is in the critical section");

            // Release token
            hasToken = false;
            tokenValue = (tokenValue + 1) % numProcesses;
            out.println(tokenValue);
            System.out.println("Process " + processId + " released the token");
        }
    }
}
```

Client Side Code :

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

public class TokenRingClient {

    static int port = 8000;
    static String host = "localhost";

    public static void main(String[] args) throws Exception {
        Socket socket = new Socket(host, port);
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

        // Send request to enter critical section
        out.println("request");
        String response = in.readLine();
        if (response.equals("grant")) {
            // Entered critical section
            System.out.println("Entered critical section");
            // Do some work here
            // ...
            // Release critical section
            out.println("release");
        } else {
```

```
            // Failed to enter critical section
            System.out.println("Failed to enter critical section");
        }

        // Clean up
        in.close();
        out.close();
        socket.close();
    }
}
```

Output:



ASSIGNMENT NO: 6

Title:

Bully and Ring algorithm for leader election

Problem Statement:

Implement Bully and Ring algorithm for leader election.

Software and tools:

1. Programming language
2. Communication protocol
3. Distributed system libraries
4. Operating system
5. Code editor
6. Algorithm Implementation

Theory:

Algorithm Implementation: To implement the Bully and Ring algorithm for leader election, the following steps can be followed:

Bully Algorithm:

1. Each process is assigned a unique ID and the highest ID process is selected as the initial leader.
2. When a process detects that the current leader has failed, it starts an election by sending an election message to all the processes with a higher ID than itself.
3. If no response is received within a specified timeout period, the process declares itself the new leader and sends a victory message to all the processes.
4. If a response is received from a higher ID process, the process waits for a victory message from the new leader.
5. When a victory message is received, the process updates its leader and continues normal operations.

Ring Algorithm:

1. Each process is assigned a unique ID and the processes are arranged in a ring topology.
2. When a process wants to initiate an election, it sends an election message to its right-hand neighbor.
3. The receiving process compares the sender's ID with its own and forwards the message to its right-hand neighbor if the sender's ID is higher.
4. If a process receives its own message back, it declares itself the new leader and sends a victory message to all the processes.
5. If a process receives a message from a higher ID process, it forwards the message to its right-hand neighbor.
6. When a victory message is received, the process updates its leader and continues normal operations.

Demonstration:

To demonstrate the effectiveness of the implemented Bully and Ring algorithm for leader election, we will perform the following steps:

1. Create multiple processes with unique IDs.
2. Simulate the failure of the current leader.
3. Observe the election process and verify that the process with the highest ID becomes the new leader.
4. Verify that the new leader is able to coordinate the activities of other processes.

Conclusion:

Implementing Bully and Ring algorithm for leader election is an important concept in distributed systems. By implementing these algorithms, students will have a clear understanding of leader election in distributed systems and how it can be achieved using the Bully and Ring algorithm. This lab practical will help students develop their skills in distributed systems and learn about the challenges and solutions associated with leader election.

Code:

Server Side code:

```
import java.net.MalformedURLException;
import java.rmi.RemoteException;
import java.rmi.server.ServerNotActiveException;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Server extends UnicastRemoteObject implements ServerInterface {
  private static final long serialVersionUID = 1L;
  List<ServerInterface> servers;
  boolean isCoordinator;

  protected Server() throws RemoteException {
    super();
    servers = new ArrayList<>();
  }

  public void addServer(ServerInterface server) throws RemoteException {
    servers.add(server);
  }

  public void removeServer(ServerInterface server) throws RemoteException {
    servers.remove(server);
  }

  public void election() throws RemoteException {
    System.out.println("Starting Election");
```

```
    int myId = this.getId();
    int maxId = myId;
    for (ServerInterface server : servers) {
        if (server.getId() > maxId) {
            maxId = server.getId();
        }
    }
    if (maxId == myId) {
        System.out.println("I am the coordinator");
        isCoordinator = true;
        for (ServerInterface server : servers) {
            if (server.getId() != myId) {
                server.coordinator(myId);
            }
        }
    } else {
        System.out.println("Sending Election Message");
        for (ServerInterface server : servers) {
            if (server.getId() > myId) {
                server.election();
            }
        }
    }
}

public void coordinator(int id) throws RemoteException {
    System.out.println("Coordinator is Server " + id);
    isCoordinator = false;
}

public int getId() throws RemoteException {
    try {
                return Integer.parseInt(getClientHost().substring(12));
        } catch (NumberFormatException | ServerNotActiveException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
        return 0;
}

public static void main(String[] args) throws RemoteException {
    Scanner scanner = new Scanner(System.in);
    Server server = new Server();
    System.out.print("Enter Server Id: ");
    int id = scanner.nextInt();
    System.out.println("Server " + id + " started");
    java.rmi.registry.LocateRegistry.createRegistry(1099);
    try {
                java.rmi.Naming.rebind("rmi://localhost/server" + id, server);
        } catch (MalformedURLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
```

```
        }
    while (true) {
        if (server.isCoordinator) {
            System.out.println("Press enter to send a message to all servers");
            scanner.nextLine();
            for (ServerInterface s : server.servers) {
                s.receiveMessage("Hello from Server " + id);
            }
        } else {
            System.out.println("Waiting for coordinator to send message");
            scanner.nextLine();
        }
    }
  }

  public void receiveMessage(String message) throws RemoteException {
    System.out.println("Message Received: " + message);
  }

@Override
public boolean isCoordinator() throws RemoteException {
        // TODO Auto-generated method stub
        return false;
}
}
```

Interface file:

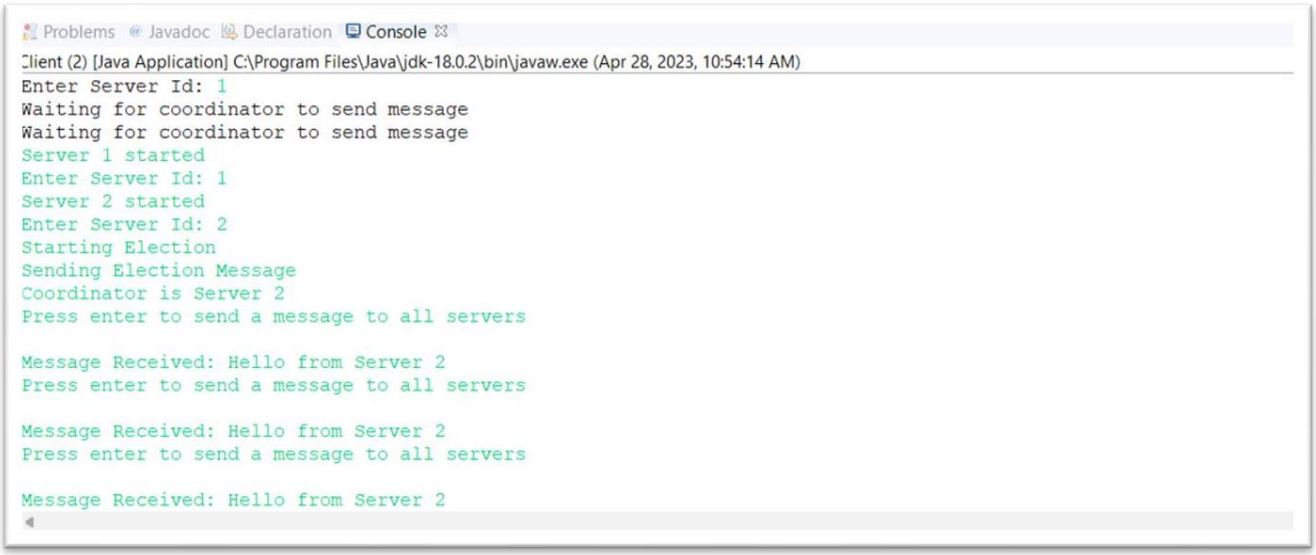
```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BankAccount extends Remote {

        void deposit (int amount) throws RemoteException;
        void withdraw (int amount) throws RemoteException;
        double getBalance() throws RemoteException;

}
```

Client-side code:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.util.Scanner;

public class Client {
  public static void main(String[] args) throws RemoteException {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter Server Id: ");
    int id = scanner.nextInt();
    try {
```

```
            ServerInterface server = (ServerInterface) Naming.lookup("rmi://localhost/server" + id);
            server.addServer(server);
            server.election();
            while (true) {
                if (server.isCoordinator()) {
                    System.out.println("Press enter to send a message to all servers");
                    scanner.nextLine();
                    server.receiveMessage("Hello from Server " + id);
                } else {
                    System.out.println("Waiting for coordinator to send message");
                    scanner.nextLine();
                }
            }
        }
        catch (Exception e) {
            System.out.println("Exception occurred: ");
        }
    }
}
```

Output:



ASSIGNMENT NO: 7

Title:

Simple web service and a distributed application to consume the web service.

Problem Statement:

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

Software and tools:

Programming language: Java

Framework: SpringBoot

IDE: Spring ToolSuite4

Networking: Java Sockets

Theory:

- A web service is any piece of software that makes itself available over the internet and uses a standardized XML messaging system. XML is used to encode all communications to a web service. For example, a client invokes a web service by sending an XML message, then waits for a corresponding XML response. As all communication is in XML, web services are not tied to any one operating system or programming language—Java can talk with Perl; Windows applications can talk with Unix applications.
- Web services are self-contained, modular, distributed, dynamic applications that can be described, published, located, or invoked over the network to create products, processes, and supply chains. These applications can be local, distributed, or web-based. Web services are built on top of open standards such as TCP/IP, HTTP, Java, HTML, and XML.
- Web services are XML-based information exchange systems that use the Internet for direct application-to-application interaction. These systems can include programs, objects, messages, or documents.
- A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

To summarize, a complete web service is, therefore, any service that −

- Is available over the Internet or private (intranet) networks
- Uses a standardized XML messaging system
- Is not tied to any one operating system or programming language
- Is self-describing via a common XML grammar
- Is discoverable via a simple find mechanism

Components of Web Services

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

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

All these components have been discussed in the Web Services Architecture chapter.

How Does a Web Service Work?

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

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

You can build a Java-based web service on Solaris that is accessible from your Visual Basic program that runs on Windows.

You can also use C# to build new web services on Windows that can be invoked from your web application that is based on JavaServer Pages (JSP) and runs on Linux.

Example

Consider a simple account-management and order processing system. The accounting personnel use a client application built with Visual Basic or JSP to create new accounts and enter new customer orders.

The processing logic for this system is written in Java and resides on a Solaris machine, which also interacts with a database to store information.

The steps to perform this operation are as follows −

- The client program bundles the account registration information into a SOAP message.
- This SOAP message is sent to the web service as the body of an HTTP POST request.
- The web service unpacks the SOAP request and converts it into a command that the application can understand.
- The application processes the information as required and responds with a new unique account number for that customer.
- Next, the web service packages the response into another SOAP message, which it sends back to the client program in response to its HTTP request.
- The client program unpacks the SOAP message to obtain the results of the account registration process.

Conclusion:

In this practical, we have learnt how to create and use web services. However, a web service also include components such as WSDL, UDDI, and SOAP that contribute to make it active. Web services are XML-based information exchange systems that use the Internet for direct application-to-application interaction. These systems can include programs, objects, messages, or documents.

Code & Output:

Server side code:

Pojo class:

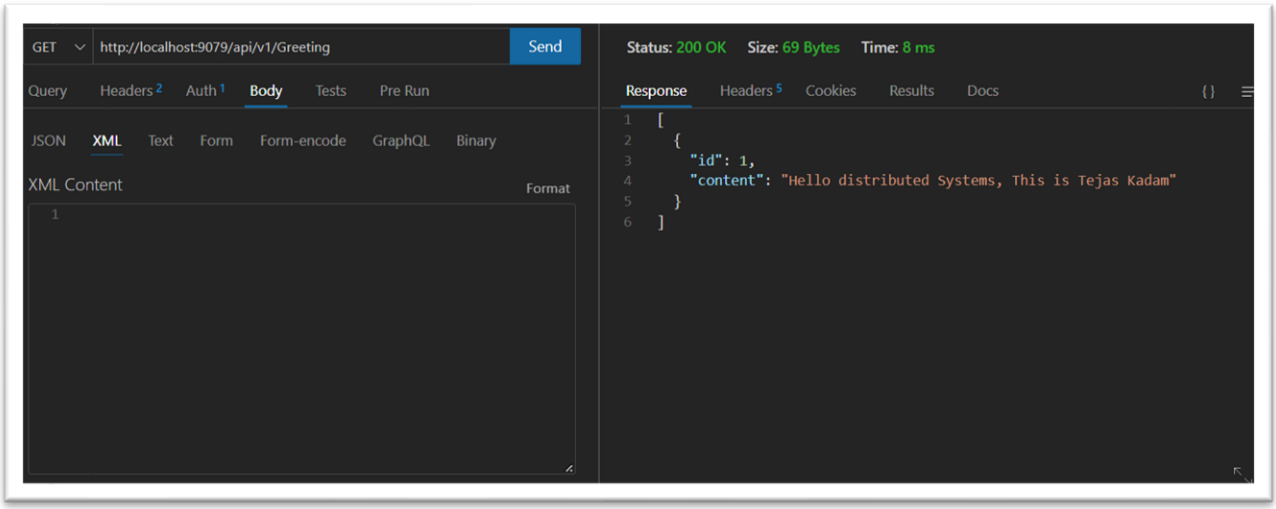
```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "testing")
public class testing {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
   private long id;
	@Column(name = "content")
   private String content;

	public testing() {
		// TODO Auto-generated constructor stub
	}

   public testing(long id, String content) {
      this.id = id;
      this.content = content;
   }

   public long getId() {
      return id;
   }

   public String getContent() {
      return content;
   }

}
```

Controller class:

```
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.exception.ResourceNotFoundException;
import com.example.demo.model.Employee;
import com.example.demo.model.testing;
import com.example.demo.repository.EmployeeRepository;
import com.example.demo.repository.TestingRepo;

@CrossOrigin(origins = "http://localhost:4200/")
@RestController
@RequestMapping("/api/v1/")
public class EmployeeController {


	@Autowired
	TestingRepo tt;

	@GetMapping("/Greeting")
	public List<testing> GetGreeting(){

		return tt.findAll();
	}
```

Server side API fetch Output:



Client-side code:

```
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class client extends SpringBootServletInitializer {

   @Bean
   public Jaxb2Marshaller marshaller() {
      Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
      marshaller.setContextPath("com.example.consumingwebservice.wsdl");
```

```
        return marshaller;
    }

    @Bean
    public GreetingClient greetingClient(Jaxb2Marshaller marshaller) {
        GreetingClient client = new GreetingClient();
        client.setDefaultUri("http://localhost:8080/ws");
        client.setMarshaller(marshaller);
        client.setUnmarshaller(marshaller);
        return client;
    }
    public static void main(String[] args) {
        SpringApplication.run(GreetingClient.class, args);
    }
    @Autowired
    private GreetingClient greetingClient;

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(GreetingClient.class);
    }

    @EventListener(ApplicationReadyEvent.class)
    public void doSomethingAfterStartup() {
        GetGreetingRequest request = new GetGreetingRequest();
        request.setName("John");
        GetGreetingResponse response = greetingClient.getGreeting(request);
        System.out.println(response.getGreeting().getContent());
    }
}
```

Client-side output:

```
:: Spring Boot ::                (v3.0.3)

)23-04-30T17:11:20.008+05:30  INFO 9080 --- [  restartedMain] c.example.demo.SpringBackendApplication  : Starting SpringBackendApplication using Java 17.0
)23-04-30T17:11:20.011+05:30  INFO 9080 --- [  restartedMain] c.example.demo.SpringBackendApplication  : No active profile set, falling back to 1 default
)23-04-30T17:11:20.038+05:30  INFO 9080 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.de
)23-04-30T17:11:20.038+05:30  INFO 9080 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setti
)23-04-30T17:11:20.358+05:30  INFO 9080 --- [  restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEF
)23-04-30T17:11:20.390+05:30  INFO 9080 --- [  restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 28 ms
)23-04-30T17:11:20.705+05:30  INFO 9080 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 9079 (http)
)23-04-30T17:11:20.712+05:30  INFO 9080 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
)23-04-30T17:11:20.712+05:30  INFO 9080 --- [  restartedMain] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.5]
)23-04-30T17:11:20.759+05:30  INFO 9080 --- [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContex
)23-04-30T17:11:20.759+05:30  INFO 9080 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization comple
)23-04-30T17:11:20.845+05:30  INFO 9080 --- [  restartedMain] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name:
)23-04-30T17:11:20.883+05:30  INFO 9080 --- [  restartedMain] org.hibernate.Version                    : HHH000412: Hibernate ORM core version 6.1.7.Final
)23-04-30T17:11:21.069+05:30  INFO 9080 --- [  restartedMain] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
)23-04-30T17:11:21.179+05:30  INFO 9080 --- [  restartedMain] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection com.mysql.cj.jdbc
)23-04-30T17:11:21.180+05:30  INFO 9080 --- [  restartedMain] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
)23-04-30T17:11:21.193+05:30  INFO 9080 --- [  restartedMain] SQL dialect                              : HHH000400: Using dialect: org.hibernate.dialect.M
)23-04-30T17:11:21.194+05:30  WARN 9080 --- [  restartedMain] org.hibernate.orm.deprecation            : HHH90000026: MySQL8Dialect has been deprecated; u
)23-04-30T17:11:21.673+05:30  INFO 9080 --- [  restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000490: Using JtaPlatform implementation: [org
)23-04-30T17:11:21.680+05:30  INFO 9080 --- [  restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persiste
)23-04-30T17:11:21.849+05:30  WARN 9080 --- [  restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Th
)23-04-30T17:11:22.066+05:30  INFO 9080 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
)23-04-30T17:11:22.093+05:30  INFO 9080 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 9079 (http) with conte
)23-04-30T17:11:22.100+05:30  INFO 9080 --- [  restartedMain] c.example.demo.SpringBackendApplication  : Started SpringBackendApplication in 2.29 seconds

Hello distributed Systems. This is Tejas Kadam"
```

ASSIGNMENT NO: 8

Title:

Distributed Application for Interactive Multiplayer Games

Problem Statement:

To design and develop a distributed application for interactive multiplayer games. The application should allow multiple users to connect and interact with each other in a game environment. The system should be able to handle multiple concurrent users and maintain consistency across all clients.

Tools/Environment:

The following tools and technologies will be used in this project:

Programming language: Java

IDE: Eclipse

Networking: Java Sockets

Theory:

There are mainly two possible network architectures: peer-to-peer and client-server. In the peer-to-peer architecture, data is exchanged between any pair of connected players while in the client-server architecture, data is only exchanged between players and the server.

There are mainly three components in game networking:

- Transport protocol: how to transport the data between clients and the server?
- Application protocol: what to send from clients to the server and from the server to clients and in which format?
- Application logic: how to use the exchanged data to update clients and the server?

To build a distributed application for multiplayer games, we need to consider the following concepts:

Client-Server architecture: The system will have a central server that will handle client requests and send back responses.

Synchronization: The server must maintain consistency across all clients by synchronizing game state data.

Communication: The server must handle communication between clients, sending messages and data as necessary.

Game mechanics: The game mechanics will be designed to support multiplayer interactions, such as real-time updates and player interactions.

Implementation:

The implementation of the distributed application for multiplayer games will involve the following steps:

Design the game mechanics and user interface using JavaFX.

Implement the server using Java Sockets, which will handle client requests and maintain game state data.

Implement the client using Java Sockets, which will connect to the server, send requests, and receive responses.

Implement synchronization between the server and clients to ensure consistency of game state data.

Implement communication between clients to allow real-time interactions.

Test the system thoroughly to ensure that it is working as expected.

Conclusion:

The development of a distributed application for interactive multiplayer games involves a number of challenges, including synchronization, communication, and game mechanics. By following the steps outlined above, it is possible to design and develop a system that can handle multiple concurrent users and maintain consistency across all clients. This practical assignment will provide students with valuable experience in designing and implementing distributed applications.

Code:

Server-Side Code:

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

public class GameServer {
   public static void main(String[] args) throws Exception {
      ServerSocket serverSocket = new ServerSocket(9027);
      System.out.println("Server started on port 9027");

      int numClients = 2; // connect 2 clients
      for (int i = 1; i <= numClients; i++) {
         System.out.println("Waiting for client " + i + " to connect...");

         // Wait for a client to connect
         Socket socket = serverSocket.accept();
         System.out.println("Client " + i + " connected.");

         // Start a new thread to handle the client
         Thread clientThread = new Thread(new ClientHandler(socket));
         clientThread.start();
      }

      // Close the server socket
      serverSocket.close();
   }
}

class ClientHandler implements Runnable {
```

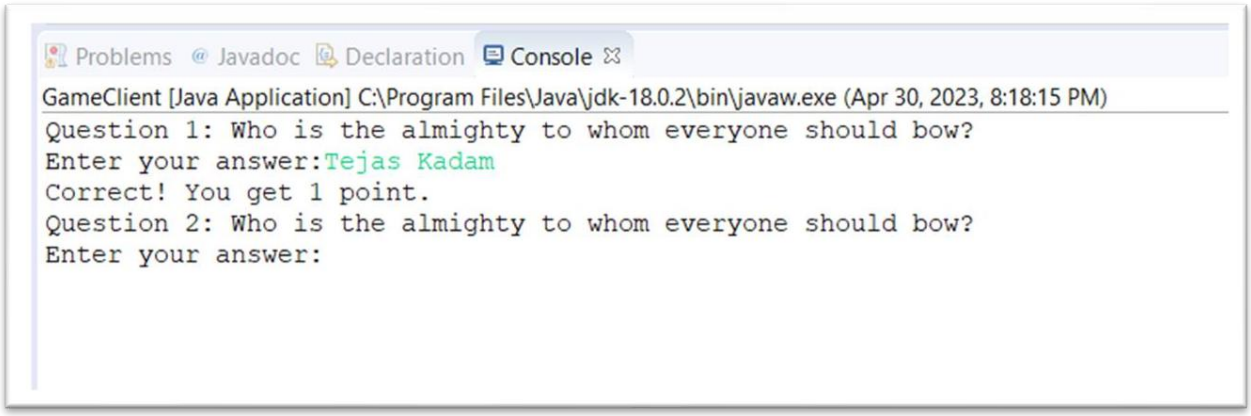
```
private Socket socket;
private BufferedReader input;
private PrintWriter output;
private int score;

public ClientHandler(Socket socket) throws Exception {
    this.socket = socket;
    input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    output = new PrintWriter(socket.getOutputStream(), true);
    score = 0;
}

public void run() {
    try {
        int numQuestions = 3; // ask 3 questions
        for (int i = 1; i <= numQuestions; i++) {
            // Ask a question
            String question = "Question " + i + ": Who is the almighty to whom everyone should bow?";
            output.println(question);

            // Prompt for answer
            output.println("Enter your answer:");

            // Set a timeout of 10 seconds
            socket.setSoTimeout(10000);

            // Read the client's answer
            String answer = input.readLine();

            if (answer != null && answer.equalsIgnoreCase("tejas kadam")) {
                // Correct answer, increase the client's score
                score++;
                output.println("Correct! You get 1 point.");
            } else {
                output.println("Wrong answer.");
            }
        }

        // Send the client's final score
        output.println("Game over. Your score: " + score);
    } catch (SocketTimeoutException e) {
        output.println("Timeout. You did not answer in time.");
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
}
```

Client-side Code:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class GameClient {
    public static void main(String[] args) throws Exception {
        Socket socket = new Socket("localhost", 9027);
        BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter output = new PrintWriter(socket.getOutputStream(), true);

        int numQuestions = 3; // ask 3 questions
        for (int i = 1; i <= numQuestions; i++) {
            // Read the question from the server
            String question = input.readLine();
            System.out.println(question);

            // Read the prompt for answer
            String prompt = input.readLine();
            System.out.print(prompt);

            // Set a timeout of 10 seconds
            socket.setSoTimeout(10000);

            // Read the client's answer
            BufferedReader userInput = new BufferedReader(new InputStreamReader(System.in));
            String answer = userInput.readLine();

            // Send the answer to the server
            output.println(answer);

            // Read the server's response
            String response = input.readLine();
            System.out.println(response);
        }

        // Read the final score from the server
        String finalScore = input.readLine();
        System.out.println(finalScore);

        // Close the socket
        socket.close();
    }
}
```

Output:


Problems
Javadoc
Declaration
Console
GameClient [Java Application] C:\Program Files\Java\jdk-18.0.2\bin\javaw.exe (Apr 30, 2023, 8:18:15 PM)
Question 1: Who is the almighty to whom everyone should bow?
Enter your answer:Tejas Kadam
Correct! You get 1 point.
Question 2: Who is the almighty to whom everyone should bow?
Enter your answer: