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

RMI (Remote Method Invocation) is a Java API that provides a mechanism to create distributed applications in Java. It allows communication between two Java Virtual Machines (JVMs) running on different machines over the network. In this example, we will implement a multi-threaded client/server process communication using RMI in Java.

Step 1: Define the Interface First, we need to define the remote interface that the client will use to communicate with the server. In this example, we will define a simple interface called **"ProcessManager"** with a method called "executeProcess". This method takes a String parameter representing the command to execute on the server, and returns a String representing the output of the command.

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

public interface ProcessManager extends Remote {
   public String executeProcess(String command) throws RemoteException;
}
```

Step 2: Implement the Server Next, we need to implement the server that will provide the remote service to the client. In this example, we will create a class called **"ProcessManagerImpl"** that implements the "ProcessManager" interface. This class will use Java's ProcessBuilder API to execute the command received from the client and return the output to the client.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ProcessManagerImpl extends UnicastRemoteObject implements
ProcessManager {
   public ProcessManagerImpl() throws RemoteException {
      super();
   }

   public String executeProcess(String command) throws RemoteException {
      try {
         ProcessBuilder builder = new ProcessBuilder(command.split(" "));
         builder.redirectErrorStream(true);
         Process process = builder.start();

BufferedReader reader = new BufferedReader(new
InputStreamReader(process.getInputStream()));
         StringBuilder output = new StringBuilder();
         String line;
```

```
            while ((line = reader.readLine()) != null) {
                output.append(line).append("\n");
            }
             return output.toString();
        } catch (Exception e) {
            throw new RemoteException("Error executing command: " + e.getMessage());
        }}}
```

Step 3: Implement the Client Finally, we need to implement the client that will use the remote service provided by the server. In this example, we will create a simple class called **"ProcessClient"** that takes the command to execute as a command-line argument and sends it to the server using RMI.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ProcessClient {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java ProcessClient <command>");
            System.exit(1);
        }

        try {
            Registry registry = LocateRegistry.getRegistry("localhost");
      ProcessManager processManager = (ProcessManager) registry.lookup("ProcessManager");

            String output = processManager.executeProcess(args[0]);
            System.out.println(output);
        } catch (Exception e) {
            e.printStackTrace();
        }}}
```

Step 4: Start the Server To start the server, we need to create a Registry and bind the remote service to a name. In this example, we will use the default port for RMI (1099) and bind the service to the name "ProcessManager".

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ProcessServer {
    public static void main(String[] args) {
        try {
            ProcessManager processManager = new ProcessManagerImpl();
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.bind("ProcessManager", processManager);
            System.out.println("ProcessManager service started.");
        } catch (Exception e) {
            e.printStackTrace
```

2. Develop distributed application for calculator using CORBA to demonstrate object brokering.

CORBA (Common Object Request Broker Architecture) is a middleware technology that allows communication between objects in a distributed system. In this example, we will develop a distributed application for a calculator using CORBA to demonstrate object brokering.

Step 1: Define the Interface First, we need to define the interface that the client will use to communicate with the server. In this example, we will define a simple interface called **"Calculator"** with **methods for addition, subtraction, multiplication, and division.**

```
module CalculatorApp {
   interface Calculator {
      float add(in float a, in float b);
      float subtract(in float a, in float b);
      float multiply(in float a, in float b);
      float divide(in float a, in float b) raises (CalculatorException);
   };

   exception CalculatorException {
      string reason;
   };
};
```

Step 2: Implement the Server Next, we need to implement the server that will provide the remote service to the client. In this example, we will create a class called **"CalculatorImpl"** that implements the "Calculator" interface. This class will perform the requested calculations and return the result to the client.

```
import CalculatorApp.CalculatorPOA;

public class CalculatorImpl extends CalculatorPOA {
   public float add(float a, float b) {
      return a + b;
   }

   public float subtract(float a, float b) {
      return a - b;
   }

   public float multiply(float a, float b) {
      return a * b;
   }

   public float divide(float a, float b) throws CalculatorException {
      if (b == 0) {
            CalculatorException ex = new CalculatorException();
            ex.reason = "Division by zero.";
            throw ex;
        }

        return a / b;
    }
}
```

Step 3: Implement the Client Finally, we need to implement the client that will use the remote service provided by the server. In this example, we will create a simple class called **"CalculatorClient"** that takes the operation to perform and the operands as command-line arguments and sends it to the server using CORBA.

```
import CalculatorApp.Calculator;
import CalculatorApp.CalculatorException;
import org.omg.CORBA.ORB;

public class CalculatorClient {
    public static void main(String[] args) {
        if (args.length != 3) {
                         System.out.println("Usage:  java  CalculatorClient  <operation>  <operand1>
<operand2>");
            System.exit(1);
        }

        try {
            ORB orb = ORB.init(args, null);
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            Calculator calculator = CalculatorHelper.narrow(ncRef.resolve_str("Calculator"));

            float a = Float.parseFloat(args[1]);
            float b = Float.parseFloat(args[2]);
            float result;

            switch (args[0]) {
                case "add":
                    result = calculator.add(a, b);
                    System.out.println("Result: " + result);
                    break;
                case "subtract":
                    result = calculator.subtract(a, b);
                    System.out.println("Result: " + result);
                    break;
                case "multiply":
                    result = calculator.multiply(a, b);
                    System.out.println("Result: " + result);
                    break;
                case "divide":
                    try {
                      result = calculator.divide(a, b);
                      System.out.println("Result: " + result);
                    } catch (CalculatorException e) {
                      System.out.println("Error: " + e.reason);
                    }
                    break;
                default:
                    System.out.println("Invalid operation.");
```

general instructions on how to install MPI (Message Passing Interface) on Ubuntu:

sudo apt install mpi-default-dev

This will install the default MPI implementation on Ubuntu.

If you prefer a specific MPI implementation, you can install it by specifying the package name in the `apt-get` command. For example, to install OpenMPI, run:

sudo apt install libopenmpi-dev

After the installation is complete, you can test the MPI installation by running the following command:

mpiexec --version

Once you have installed an MPI implementation on your Ubuntu system, you can use the following commands to execute MPI programs:

Compile your MPI program using the MPI compiler wrapper. For example, if your program is named `my_mpi_program.c`, you can compile it using the following command:

mpicc my_mpi_program.c -o my_mpi_program

This will compile your program and generate an executable named **`my_mpi_program`**.

Run your MPI program using the **`mpiexec`** command. For example, to run **`my_mpi_program`** with 4 processes, use the following command:

mpiexec -n 4 ./my_mpi_program
This will run your program with 4 processes.

The `-n` option of `mpiexec` specifies the number of processes to run. The `./` before the program name is necessary to specify the current directory.

Note that when you run your MPI program using `mpiexec`, it will create multiple processes that communicate with each other using MPI. Therefore, you should make sure that your program is designed to work with MPI and uses the appropriate MPI functions to communicate between processes.

MPI (Message Passing Interface) provides a rich set of functions for communication and synchronization between processes in a distributed memory parallel program. Here are some commonly used MPI functions:

`MPI_Init()`: This function initializes MPI environment and should be called before any other MPI function.

1. `MPI_Comm_size()`: This function returns the total number of processes in a specified communicator.
2. `MPI_Comm_rank()`: This function returns the rank of the calling process within a specified communicator.
3. `MPI_Send()`: This function sends a message from one process to another.
4. `MPI_Recv()`: This function receives a message from another process.
5. `MPI_Bcast()`: This function broadcasts a message from one process to all other processes in a communicator.
6. `MPI_Reduce()`: This function performs a reduction operation (such as sum, product, maximum, or minimum) across all processes in a communicator and returns the result to a designated process.
7. `MPI_Barrier()`: This function synchronizes all processes in a communicator, forcing them to wait until all processes have reached the barrier before continuing.
8. `MPI_Finalize()`: This function cleans up MPI environment and should be called after all other MPI functions have been called.

These are just a few of the many MPI functions available. The MPI library provides many more functions for more advanced communication patterns, such as scatter-gather, all-to-all communication, and non-blocking communication, among others. It's important to consult the MPI documentation to find the right function for your specific communication needs.

To execute an MPI program written in Java, you need to follow these steps:

1.Install an MPI implementation that supports Java.
OpenMPI is a popular MPI implementation that supports Java.

2.Write your MPI program in Java using the MPI Java bindings.
The MPI Java bindings provide a set of classes and methods that mirror the MPI C functions.

Here is an example MPI program in Java that sends a message from one process to another:

```
import mpi.*;

public class MPITest {
  public static void main(String[] args) throws MPIException {
    MPI.Init(args);
    int rank = MPI.COMM_WORLD.Rank();
    if (rank == 0) {
      int[] data = {1, 2, 3, 4, 5};
      MPI.COMM_WORLD.Send(data, 0, data.length, MPI.INT, 1, 0);
    } else if (rank == 1) {
      int[] data = new int[5];
      MPI.COMM_WORLD.Recv(data, 0, data.length, MPI.INT, 0, 0);
      System.out.println("Received data: ");
      for (int i = 0; i < data.length; i++) {
        System.out.println(data[i]);
      }
    }
    MPI.Finalize();
  }
}
```

This program sends an integer array from process 0 to process 1.

3.Compile your Java MPI program using the `mpijavac` command provided by your MPI implementation. For example, if you're using OpenMPI, you can compile the program using the following command:

mpijavac MPITest.java

This will compile the Java source code and generate a class file

4.Run your Java MPI program using the `mpirun` command provided by your MPI implementation. For example, to run the `MPITest` program with 2 processes, use the following command:

mpirun -n 2 java MPITest

This will run the Java program with 2 processes.

The -n option of mpirun specifies the number of processes to run. The java command is used to specify that the program is written in Java. Note that when you run your MPI program using mpirun, it will create multiple Java processes that communicate with each other using MPI. Therefore, you should make sure that your program is designed to work with MPI and uses the appropriate MPI functions to communicate between processes.

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

In this example, we will develop a distributed system using MPI to find the sum of N elements in an array by distributing N/n elements to n number of processors. We will also display the intermediate sums calculated at different processors.

Step 1: Initialize MPI Environment.

We will start by initializing the MPI environment and getting the rank and size of the processors.

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

int main(int argc, char** argv) {
   int rank, size, n = 10, sum = 0;
   int *arr = (int*)malloc(sizeof(int) * n);
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Step 2: Distribute Array Elements.
Next, we will distribute the array elements to different processors.
We will use the "MPI_Scatter" function to distribute the array elements.

```
   if (rank == 0) {
      printf("Enter %d array elements:\n", n);
      for (int i = 0; i < n; i++) {
         scanf("%d", &arr[i]);
      }
   }

   int *localArr = (int*)malloc(sizeof(int) * (n / size));
   MPI_Scatter(arr, n / size, MPI_INT, localArr, n / size, MPI_INT, 0, MPI_COMM_WORLD);
```

Step 3: Calculate Local Sum.
Each processor will calculate its local sum by iterating through its local array elements.

```
   int localSum = 0;
   for (int i = 0; i < n / size; i++) {
      localSum += localArr[i];
```

```
    }
```

Step 4: Reduce Sums.
We will use the "MPI_Reduce" function to reduce the local sums at different processors to the final sum at processor 0.

```
    MPI_Reduce(&localSum, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Step 5: Display Intermediate Sums.
We will display the intermediate sums calculated at each processor using the "MPI_Gather" function.

```
    int *intermediateSums = NULL;
    if (rank == 0) {
        intermediateSums = (int*)malloc(sizeof(int) * size);
    }

    MPI_Gather(&localSum, 1, MPI_INT, intermediateSums, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Intermediate Sums:\n");
        for (int i = 0; i < size; i++) {
            printf("Processor %d: %d\n", i, intermediateSums[i]);
        }
    }
```

Step 6: Finalize MPI Environment.
We will finalize the MPI environment and free the memory.

```
    if (rank == 0) {
        printf("Total Sum: %d\n", sum);
    }

    MPI_Finalize();
    free(arr);
    free(localArr);
    free(intermediateSums);

    return 0;
}
```

4.Implement Berkeley algorithm for clock synchronisation in java.

The Berkeley algorithm is a clock synchronization algorithm that is used to synchronize the clocks of different computers in a distributed system. The algorithm works by having a coordinator node called the "time server" that collects the clock values from all the nodes, computes the average time, and then sends the time differences back to each node to adjust their local clocks accordingly.

Here is the implementation of the Berkeley algorithm in Java:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Scanner;

public class BerkeleyAlgorithm {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int n;
      System.out.print("Enter the number of nodes: ");
      n = sc.nextInt();
      List<Integer> timeValues = new ArrayList<>();
      for (int i = 0; i < n; i++) {
         System.out.print("Enter the time value of node " + i + ": ");
         timeValues.add(sc.nextInt());
      }

      int averageTime = calculateAverage(timeValues);

      System.out.println("Average time: " + averageTime);

      List<Integer> timeDifferences = new ArrayList<>();
      for (int i = 0; i < n; i++) {
         int timeDifference = averageTime - timeValues.get(i);
         timeDifferences.add(timeDifference);
      }

      System.out.println("Time differences: " + timeDifferences);

      System.out.println("Enter the time offset for the time server: ");
      int timeOffset = sc.nextInt();

      for (int i = 0; i < n; i++) {
         int adjustedTime = timeValues.get(i) + timeDifferences.get(i) + timeOffset;
         System.out.println("Adjusted time for node " + i + ": " + adjustedTime);
      }
   }

   private static int calculateAverage(List<Integer> timeValues) {
      int sum = 0;
      for (int value : timeValues) {
         sum += value;
      }
      return sum / timeValues.size();
   }
}
```

In this implementation, we first take the input of the number of nodes and their respective time values. Then we calculate the average time value and the time differences of each node from the average time. After that, we take the input of the time offset for the time server and adjust the local clocks of each node based on the time differences and time offset.

Note that this is a simple implementation of the algorithm, and in a real distributed system, the time server would be a separate process or machine that is responsible for coordinating the clock synchronization.

5.Implement token ring based mutual exclusion algorithm in java.

The Token Ring algorithm is a distributed mutual exclusion algorithm that allows a group of nodes in a network to coordinate access to a shared resource by passing a token around the network. Only the node that holds the token can access the shared resource, while other nodes wait for the token to be passed to them.

Here's an implementation of the Token Ring algorithm in Java:

```
import java.util.concurrent.Semaphore;

public class TokenRingAlgorithm {
   private static final int NUM_NODES = 5;
   private static final int TOKEN_VALUE = 1;
   private static int token = 0;
   private static Semaphore mutex = new Semaphore(1);
   private static Semaphore tokenSemaphore = new Semaphore(0);

   public static void main(String[] args) {
      Node[] nodes = new Node[NUM_NODES];

      // Create nodes and start threads
      for (int i = 0; i < NUM_NODES; i++) {
         nodes[i] = new Node(i);
         nodes[i].start();
      }

      // Start the token
      tokenSemaphore.release();
   }

   private static class Node extends Thread {
      private int nodeId;

      public Node(int nodeId) {
         this.nodeId = nodeId;
      }
```

```
        @Override
        public void run() {
            while (true) {
                try {
                    // Wait for the token
                    tokenSemaphore.acquire();

                    // Access the critical section
                    mutex.acquire();
                    System.out.println("Node " + nodeId + " is in the critical section");


                    // Release the token
                    token = TOKEN_VALUE;
                    System.out.println("Node " + nodeId + " is passing the token");
                    tokenSemaphore.release();


                    // Exit the critical section
                    mutex.release();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

In this implementation, we create Node objects, which represent the nodes in the network. Each node runs on a separate thread and waits for the token to be passed to it.

When a node receives the token, it enters the critical section, accesses the shared resource, and then passes the token to the next node in the ring. Once a node has passed the token, it releases the mutex and waits for the next token to be passed to it.

Note that this implementation does not handle failures or timeouts. In a real implementation, additional logic would be required to handle these cases.

6.Implement Bully and Ring algorithm for leader election in java.

Leader election is a process by which a group of processes in a distributed system elect a leader from among themselves. The leader is typically responsible for coordinating activities and making decisions on behalf of the group.

Two commonly used algorithms for leader election are the Bully algorithm and the Ring algorithm. Here's an implementation of both algorithms in Java:

```
import java.util.ArrayList;
import java.util.List;

public class BullyAlgorithm {
    private static final int NUM_NODES = 5;

    public static void main(String[] args) {
        List<Node> nodes = new ArrayList<>();
        for (int i = 0; i < NUM_NODES; i++) {
            nodes.add(new Node(i, NUM_NODES));
        }

        // Node 0 initiates the election
        nodes.get(0).startElection();
    }

    private static class Node {
        private int nodeId;
        private int numNodes;
        private boolean coordinator;
        private boolean active;

        public Node(int nodeId, int numNodes) {
            this.nodeId = nodeId;
            this.numNodes = numNodes;
            this.coordinator = false;
            this.active = true;
        }

        public void startElection() {
            System.out.println("Node " + nodeId + " starts the election");
            int highestId = nodeId;
            for (int i = nodeId + 1; i < numNodes; i++) {
                if (nodes.get(i).isActive()) {
                    if (nodes.get(i).getNodeId() > highestId) {
                        highestId = nodes.get(i).getNodeId();
                    }
                    nodes.get(i).sendElectionMessage(nodeId);
                }
            }
            if (highestId == nodeId) {
                // Node is the coordinator
                coordinator = true;
                System.out.println("Node " + nodeId + " is the coordinator");
                sendCoordinatorMessage();
            } else {
```

```
            // Node is not the coordinator
            coordinator = false;
            System.out.println("Node " + nodeId + " is not the coordinator");
        }
    }

    public void sendElectionMessage(int senderId) {
        if (!active) {
            return;
        }
         System.out.println("Node " + nodeId + " receives an election message from Node " +
senderId);
        if (nodeId > senderId) {
            // Node has higher priority than sender, so start a new election
            startElection();
        } else {
            // Node has lower priority than sender, so ignore message
        }
    }

    public void sendCoordinatorMessage() {
        for (int i = nodeId + 1; i < numNodes; i++) {
            if (nodes.get(i).isActive()) {
                nodes.get(i).receiveCoordinatorMessage();
            }
        }
    }

    public void receiveCoordinatorMessage() {
        if (!active) {
            return;
        }
        System.out.println("Node " + nodeId + " receives a coordinator message");
        coordinator = true;
        System.out.println("Node " + nodeId + " is the coordinator");
    }

    public int getNodeId() {
        return nodeId;
    }

    public boolean isActive() {
        return active;
    }

    public boolean isCoordinator() {
        return coordinator;
    }
```

```
        public void setActive(boolean active) {
            this.active = active;
        }
    }
}
```

In this implementation of the Bully algorithm, each node maintains a list of all other nodes in the network, along with their active/inactive status. When a node starts the election, it sends an election message to all higher-priority nodes in the network. If a node receives an election message from a lower-priority node, it ignores the message. If it receives an election message from a higher-priority node.

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

Here's an example of creating a simple web service in Java using the JAX-WS .
JAX-WS (Java API for XML Web Services) technology:

```
import javax.jws.WebService;

@WebService
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
}
```

In this example, we have created a simple web service called Calculator that exposes an add operation.

Now, let's write a distributed application to consume this web service. We'll use the Apache CXF framework to generate the client-side code for our web service:

First, we need to generate the client-side code using the wsdl2java tool provided by CXF. The tool takes the URL of the WSDL file for the web service as input and generates Java code for the client-side stubs and skeletons.

```
$ wsdl2java -d src -p com.example.client http://localhost:8080/calculator?wsdl
```

This command generates Java code for the client-side stubs and skeletons in the src/com/example/client directory.

Next, we can use the generated client-side code to invoke the add operation of the web service:

```
import com.example.client.Calculator;
import com.example.client.CalculatorService;

public class Client {
    public static void main(String[] args) {
        CalculatorService service = new CalculatorService();
        Calculator calculator = service.getCalculatorPort();

        int result = calculator.add(2, 3);
        System.out.println("Result: " + result);
    }
}
```

In this example, we create an instance of the CalculatorService class, which represents the web service endpoint. We then get a reference to the Calculator object, which represents the port through which we can invoke the operations of the web service. Finally, we invoke the add operation of the web service, passing in the arguments 2 and 3, and print the result.

Note that the getCalculatorPort method is generated by the wsdl2java tool, and returns an instance of the Calculator interface, which represents the operations of the web service.