LP-V Lab Practical Handbook

by Shiva Mahipal

1. Socket Programming:

Sure! Let's go through the first practical step by step. The first practical is about Socket Programming in Java, where you will create a simple client-server application using TCP sockets. Here's how you can perform it:

Practical 1: Socket Programming (TCP Client-Server Communication)

Tools/Environment Required:

1. Java Development Kit (JDK 1.8 or higher)

2. Eclipse IDE (or any text editor like Notepad++)

3. Basic knowledge of Java programming

Step-by-Step Instructions:

Step 1: Set Up Your Environment

1. Install JDK: If you don't have JDK installed, download and install it from the official Oracle website.

2. Install Eclipse IDE: Download and install Eclipse IDE for Java Developers.

3. Create a New Java Project:

   - Open Eclipse.

   - Go to `File > New > Java Project`.

   - Name your project (e.g., `SocketProgramming`).

   - Click Finish.

Step 2: Create the Server Program

1. Create a new Java class for the server:

- Right-click on the `src` folder in your project.
- Go to `New > Class`.
- Name the class `Server` and click `Finish`.

2. Write the Server Code:

- Copy and paste the following code into the `Server.java` file:

```
import java.io.*;

import java.net.*;


public class Server {

    public static void main(String[] args) {

        try {

            // Step 1: Create a ServerSocket object and bind it to a port (e.g., 5000)

            ServerSocket serverSocket = new ServerSocket(5000);

            System.out.println("Server is waiting for client connection...");


            // Step 2: Wait for a client to connect

            Socket socket = serverSocket.accept();

            System.out.println("Client connected!");


            // Step 3: Create input and output streams for communication

            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
```

```
        // Step 4: Read data from the client
        String message = in.readLine();
        System.out.println("Client says: " + message);

        // Step 5: Send a response back to the client
        out.println("Hello from Server!");

        // Step 6: Close the connection
        socket.close();
        serverSocket.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
}
```

Step 3: Create the Client Program

1. Create a new Java class for the client:

   - Right-click on the `src` folder in your project.

   - Go to `New > Class`.

   - Name the class `Client` and click `Finish`.

2. Write the Client Code:

   - Copy and paste the following code into the `Client.java` file:

```
import java.io.*;

import java.net.*;

public class Client {

    public static void main(String[] args) {

        try {

            // Step 1: Create a Socket object and connect to the server (localhost, port 5000)

            Socket socket = new Socket("localhost", 5000);

            System.out.println("Connected to server!");



            // Step 2: Create input and output streams for communication

            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);



            // Step 3: Send a message to the server

            out.println("Hello from Client!");



            // Step 4: Read the server's response

            String response = in.readLine();

            System.out.println("Server says: " + response);



            // Step 5: Close the connection

            socket.close();

        } catch (IOException e) {

            e.printStackTrace();

        }
```

    }

}

Step 4: Compile and Run the Programs

1. Run the Server Program:

   - Right-click on the `Server.java` file.

   - Select `Run As > Java Application`.

   - The server will start and wait for a client to connect.

2. Run the Client Program:

   - Right-click on the `Client.java` file.

   - Select `Run As > Java Application`.

   - The client will connect to the server, send a message, and receive a response.

Step 5: Observe the Output

- Server Output:

  Server is waiting for client connection...

  Client connected!

  Client says: Hello from Client!

- Client Output:

  Connected to server!

  Server says: Hello from Server!

Conclusion:

You have successfully created a simple client-server application using TCP sockets in Java. The server listens for client connections, receives a message, and sends a response back to the client.

2. Remote Method Invocation (RMI):

Objective:

To implement a multi-threaded client/server process communication using RMI.

Steps:

1. Set Up RMI Environment:

   - Ensure you have JDK 1.8 or higher installed.

   - Open Eclipse and create a new Java project (e.g., `RMIPractical`).

2. Create the Remote Interface:

   - Right-click on the `src` folder and create a new package (e.g., `com.rmi`).

   - Inside the package, create a new Java interface named `AddServerIntf.java`:

```
package com.rmi;

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

public interface AddServerIntf extends Remote {
    double add(double d1, double d2) throws RemoteException;
}
```

3. Implement the Remote Interface:

- Create a new class named `AddServerImpl.java` in the same package:

```
package com.rmi;

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

public class AddServerImpl extends UnicastRemoteObject implements AddServerIntf {
    protected AddServerImpl() throws RemoteException {
        super();
    }

    public double add(double d1, double d2) throws RemoteException {
        return d1 + d2;
    }
}
```

4. Create the Server Program:

- Create a new class named `AddServer.java` in the same package:

```
package com.rmi;
```

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


public class AddServer {
   public static void main(String[] args) {
      try {
         AddServerImpl server = new AddServerImpl();
         Registry registry = LocateRegistry.createRegistry(1099);
         registry.rebind("AddServer", server);
         System.out.println("Server is ready.");
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

5. Create the Client Program:
   - Create a new class named `AddClient.java` in the same package:

```
package com.rmi;


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


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

```
        try {
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);
            AddServerIntf server = (AddServerIntf) registry.lookup("AddServer");
            double result = server.add(5.5, 10.5);
            System.out.println("Result: " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

6. Run the Programs:

- First, run the `AddServer.java` program.
- Then, run the `AddClient.java` program.
- You should see the server print `"Server is ready."` and the client print `"Result: 16.0"`.

3. Common Object Request Broker Architecture (CORBA):

Great! Now that Practical 2: RMI is complete, let's move on to Practical 3: Common Object Request Broker Architecture (CORBA). This practical involves creating a distributed application using CORBA and Java IDL.

Practical 3: CORBA (Common Object Request Broker Architecture)

Objective:

To develop a distributed application using CORBA to demonstrate object brokering.

Tools/Environment:

1. Java Development Kit (JDK 1.8 or higher)

2. Eclipse IDE

3. Java IDL (included in JDK)

Step-by-Step Instructions

Step 1: Set Up the Project

1. Create a New Java Project:

   - Open Eclipse.

   - Go to `File > New > Java Project`.

   - Name the project (e.g., `CORBAPractical`).

   - Click `Finish`.

2. Create a Package:

   - Right-click on the `src` folder and create a new package (e.g., `com.corba`).

Step 2: Define the IDL Interface

1. Create an IDL File:

   - Right-click on the `src` folder and go to `New > File`.

   - Name the file `Calculator.idl`.

   - Add the following code to define the interface:

```
module com {
    module corba {
        interface Calculator {
            double add(in double x, in double y);
            double subtract(in double x, in double y);
```

```
            double multiply(in double x, in double y);

            double divide(in double x, in double y);

          };

        };

      };
```

2. Save the File:

   - Save the `Calculator.idl` file in the `src` folder.

Step 3: Generate Java Stubs and Skeletons

1. Open Command Prompt/Terminal:

   - Navigate to the `src` folder of your project. For example:

```
cd C:\Users\YourName\eclipse-workspace\CORBAPractical\src
```

2. Run the `idlj` Compiler:

   - Run the following command to generate Java stubs and skeletons:

idlj -fall Calculator.idl

   - This will generate the following files in the `com/corba` package:

     - `Calculator.java` (interface)

     - `CalculatorHelper.java`

     - `CalculatorHolder.java`

     - `CalculatorOperations.java`

- `_CalculatorStub.java`
- `CalculatorPOA.java` (server skeleton)

Step 4: Implement the Server

1. Create the Server Implementation:

- In the `com.corba` package, create a new Java class named `CalculatorImpl.java`:

```
package com.corba;

import org.omg.CORBA.ORB;

class CalculatorImpl extends CalculatorPOA {
    private ORB orb;

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

    public double add(double x, double y) {
        return x + y;
    }

    public double subtract(double x, double y) {
        return x - y;
```

```
    }

    public double multiply(double x, double y) {
        return x * y;
    }

    public double divide(double x, double y) {
        if (y == 0) throw new ArithmeticException("Division by zero");
        return x / y;
    }
}
```

2. Create the Server Program:

- In the `com.corba` package, create a new Java class named `CalculatorServer.java`:

```
package com.corba;

import org.omg.CORBA.ORB;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;

public class CalculatorServer {
```

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

    // Get reference to the root POA and activate the POAManager
    POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
    rootPOA.the_POAManager().activate();

    // Create the servant (implementation) and register it with the ORB
    CalculatorImpl calculatorImpl = new CalculatorImpl();
    calculatorImpl.setORB(orb);

    // Get the object reference from the servant
    org.omg.CORBA.Object ref = rootPOA.servant_to_reference(calculatorImpl);
    Calculator calculator = CalculatorHelper.narrow(ref);

    // Get the root naming context
    org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
    NamingContextExt namingContext = NamingContextExtHelper.narrow(objRef);

    // Bind the object reference in the naming context
    NameComponent[] path = namingContext.to_name("Calculator");
    namingContext.rebind(path, calculator);

    System.out.println("CalculatorServer is ready and waiting for requests...");
```

```
            // Wait for invocations from clients
            orb.run();
        } catch (Exception e) {
            System.err.println("Error: " + e);
            e.printStackTrace();
        }
    }
}
```

Step 5: Implement the Client

1. Create the Client Program:

- In the `com.corba` package, create a new Java class named `CalculatorClient.java`:

```
package com.corba;

import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;

public class CalculatorClient {
    public static void main(String[] args) {
        try {
            // Initialize the ORB
            ORB orb = ORB.init(args, null);

            // Get the root naming context
```

```java
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContextExt namingContext = NamingContextExtHelper.narrow(objRef);

            // Resolve the object reference in the naming context
            Calculator calculator =
CalculatorHelper.narrow(namingContext.resolve_str("Calculator"));

            // Perform calculations
            System.out.println("5 + 3 = " + calculator.add(5, 3));
            System.out.println("5 - 3 = " + calculator.subtract(5, 3));
            System.out.println("5 * 3 = " + calculator.multiply(5, 3));
            System.out.println("5 / 3 = " + calculator.divide(5, 3));
        } catch (Exception e) {
            System.err.println("Error: " + e);
            e.printStackTrace();
        }
    }
}
```

Step 6: Run the CORBA Application

1. Start the ORBD (Object Request Broker Daemon):

- Open a command prompt/terminal and run:

```
orbd -ORBInitialPort 1050 -ORBInitialHost localhost
```

- This starts the CORBA naming service on port `1050`.

2. Run the Server:

- In Eclipse, run `CalculatorServer.java`.

- The server should print:

```
CalculatorServer is ready and waiting for requests...
```

3. Run the Client:

- In Eclipse, run `CalculatorClient.java`.

- The client should print:

```
5 + 3 = 8.0

5 - 3 = 2.0

5 * 3 = 15.0

5 / 3 = 1.6666666666666667
```

Summary

- We defined an IDL interface (`Calculator.idl`).

- Generated Java stubs and skeletons using the `idlj` compiler.

- Implemented the server (`CalculatorImpl.java` and `CalculatorServer.java`).

- Implemented the client (`CalculatorClient.java`).

- Ran the CORBA application using `orbd`.

4. Message Passing Interface (MPI):

(Not to be asked in our practical exams)

Practical 4: Message Passing Interface (MPI). This practical involves developing a distributed system to find the sum of elements in an array by distributing the work across multiple processors using MPI or OpenMP.

Practical 4: Message Passing Interface (MPI)

Objective:

To develop a distributed system that calculates the sum of `N` elements in an array by distributing `N/n` elements to `n` processors using MPI. The intermediate sums calculated at different processors will be displayed.

Tools/Environment:

1. Java Development Kit (JDK 1.8 or higher)

2. MPJ Express (a Java-based MPI library)

3. Eclipse IDE

Step-by-Step Instructions

Step 1: Set Up MPJ Express

1. Download MPJ Express:

   - Go to the [MPJ Express website](https://mpj-express.org/) and download the latest version of MPJ Express (e.g., `mpj-v0_44.zip`).

2. Extract MPJ Express:

   - Extract the downloaded ZIP file to a directory (e.g., `C:\mpj`).

3. Set Environment Variables:

   - Open the `Environment Variables` settings on your system.

   - Add the `MPJ_HOME` variable pointing to the MPJ Express directory (e.g., `C:\mpj`).

- Add the `bin` directory of MPJ Express to the `PATH` variable (e.g., `C:\mpj\bin`).

Step 2: Create the MPI Project in Eclipse

1. Create a New Java Project:

   - Open Eclipse and create a new Java project (e.g., `MPIPractical`).

2. Add MPJ Express Library:

   - Right-click on the project and select `Properties`.

   - Go to `Java Build Path > Libraries` and click `Add External JARs`.

   - Browse to the `lib` folder in the MPJ Express directory and add `mpj.jar`.

Step 3: Write the MPI Program

1. Create a New Java Class:

   - Create a new Java class named `MPISum.java` in the `src` folder.

2. Write the MPI Code:

   - Copy and paste the following code into `MPISum.java`:

```
import mpi.*;

public class MPISum {
    public static void main(String[] args) {
        // Initialize MPI
        MPI.Init(args);
```

```java
// Get the rank and size of the MPI world
int rank = MPI.COMM_WORLD.Rank();
int size = MPI.COMM_WORLD.Size();

// Define the array and its size
int N = 100; // Total number of elements
int[] array = new int[N];
int[] localArray = new int[N / size];
int[] localSum = new int[1];
int[] globalSum = new int[1];

// Initialize the array on the root process (rank 0)
if (rank == 0) {
    for (int i = 0; i < N; i++) {
        array[i] = i + 1; // Fill the array with values 1 to N
    }
}

// Scatter the array to all processes
MPI.COMM_WORLD.Scatter(array, 0, N / size, MPI.INT, localArray, 0, N / size, MPI.INT, 0);

// Calculate the local sum
localSum[0] = 0;
for (int i = 0; i < localArray.length; i++) {
    localSum[0] += localArray[i];
```

```java
        }

        // Gather the local sums to the root process
        MPI.COMM_WORLD.Gather(localSum, 0, 1, MPI.INT, globalSum, 0, 1, MPI.INT, 0);

        // Calculate and display the global sum on the root process
        if (rank == 0) {
            int totalSum = 0;
            for (int i = 0; i < size; i++) {
                totalSum += globalSum[i];
            }
            System.out.println("Total sum: " + totalSum);
        }

        // Finalize MPI
        MPI.Finalize();
    }
}
```

Step 4: Run the MPI Program

1. Compile the Program:

   - Right-click on the project and select `Build Project`.

2. Run the Program:

   - Open a command prompt and navigate to the `bin` folder of your project (e.g., `C:\Users\YourName\eclipse-workspace\MPIPractical\bin`).

- Run the following command to execute the MPI program:

```
mpjrun.sh -np 4 MPISum
```

- Replace `4` with the number of processors you want to use.

3. Observe the Output:

   - The program will calculate the sum of the array elements and display the result.

Step 5: Understand the Code

1. MPI Initialization:

   - `MPI.Init(args)` initializes the MPI environment.

   - `MPI.COMM_WORLD.Rank()` returns the rank (ID) of the current process.

   - `MPI.COMM_WORLD.Size()` returns the total number of processes.

2. Scatter the Array:

   - The `MPI.COMM_WORLD.Scatter()` method distributes the array to all processes.

3. Calculate Local Sum:

   - Each process calculates the sum of its portion of the array.

4. Gather the Results:

   - The `MPI.COMM_WORLD.Gather()` method collects the local sums from all processes.

5. Display the Global Sum:

   - The root process (rank 0) calculates and displays the global sum.

Summary

- We set up MPJ Express and created an MPI program to calculate the sum of array elements.

- The program distributes the work across multiple processors and gathers the results.

- The final sum is displayed by the root process.

5. Clock Synchronization (Berkeley Algorithm):

Great! Let's move on to Practical 5: Clock Synchronization (Berkeley Algorithm). This practical involves implementing the Berkeley Algorithm to synchronize clocks in a distributed system.

Objective

To develop an n-node distributed system that implements the Berkeley Algorithm for clock synchronization.

Key Concepts

1. Berkeley Algorithm:

   - A clock synchronization algorithm used in distributed systems.

   - A master node collects clock values from all slave nodes, calculates the average, and sends the adjustment to all nodes.

2. Logical Clocks:

   - Each node maintains its own logical clock.

   - The goal is to synchronize these clocks to a common value.

Step-by-Step Implementation

Step 1: Set Up the Project

1. Create a New Java Project:

   - Open Eclipse → `File → New → Java Project`.

   - Name: `ClockSynchronization`.

2. Create a Package:

   - Right-click `src` → `New → Package`.

   - Name: `com.berkeley`.

Step 2: Implement the Berkeley Algorithm

1. Create the `Node` Class:

   - This class represents a node in the distributed system.

   - Create a file named `Node.java` in the `com.berkeley` package:

```
package com.berkeley;

public class Node {
    private int id;
    private int clock;

    public Node(int id, int clock) {
        this.id = id;
        this.clock = clock;
    }
```

```
    public int getId() {
        return id;
    }

    public int getClock() {
        return clock;
    }

    public void setClock(int clock) {
        this.clock = clock;
    }

    public void adjustClock(int adjustment) {
        this.clock += adjustment;
    }
}
```

2. Create the `MasterNode` Class:

   - This class represents the master node that coordinates clock synchronization.

   - Create a file named `MasterNode.java` in the `com.berkeley` package:

```
package com.berkeley;

import java.util.ArrayList;
```

```java
import java.util.List;


public class MasterNode {
    private List<Node> nodes;


    public MasterNode() {
        nodes = new ArrayList<>();
    }


    public void addNode(Node node) {
        nodes.add(node);
    }


    public void synchronizeClocks() {
        // Step 1: Collect clock values from all nodes
        int sum = 0;
        for (Node node : nodes) {
            sum += node.getClock();
        }


        // Step 2: Calculate the average clock value
        int average = sum / nodes.size();


        // Step 3: Send the adjustment to each node
        for (Node node : nodes) {
            int adjustment = average - node.getClock();
```

```
            node.adjustClock(adjustment);
            System.out.println("Node " + node.getId() + " adjusted by " + adjustment);
        }
    }
}
```

3. Create the `Main` Class:

   - This class simulates the distributed system and runs the Berkeley Algorithm.

   - Create a file named `Main.java` in the `com.berkeley` package:

```
package com.berkeley;

public class Main {
    public static void main(String[] args) {
        // Create nodes with initial clock values
        Node node1 = new Node(1, 10);
        Node node2 = new Node(2, 15);
        Node node3 = new Node(3, 20);

        // Create the master node
        MasterNode masterNode = new MasterNode();
        masterNode.addNode(node1);
        masterNode.addNode(node2);
        masterNode.addNode(node3);
```

```java
        // Synchronize clocks
        System.out.println("Before synchronization:");
        System.out.println("Node 1: " + node1.getClock());
        System.out.println("Node 2: " + node2.getClock());
        System.out.println("Node 3: " + node3.getClock());

        masterNode.synchronizeClocks();

        System.out.println("After synchronization:");
        System.out.println("Node 1: " + node1.getClock());
        System.out.println("Node 2: " + node2.getClock());
        System.out.println("Node 3: " + node3.getClock());
    }
}
```

Step 3: Run the Program

1. Compile the Code:

   - Right-click the project → `Build Project`.

2. Run the Program:

   - Right-click `Main.java` → `Run As → Java Application`.

Expected Output

```
Before synchronization:

Node 1: 10

Node 2: 15

Node 3: 20

Node 1 adjusted by 5

Node 2 adjusted by 0

Node 3 adjusted by -5

After synchronization:

Node 1: 15

Node 2: 15

Node 3: 15
```

How It Works

1. Initialization:

   - Each node starts with its own clock value.

   - The master node collects clock values from all nodes.

2. Synchronization:

   - The master calculates the average clock value.

   - It sends the adjustment (difference between average and node's clock) to each node.

3. Adjustment:

  - Each node adjusts its clock based on the master’s instruction.

Key Points

- Master Node: Coordinates the synchronization process.
- Slave Nodes: Adjust their clocks based on the master’s instructions.
- Logical Clocks: Represent the time at each node.

6. Mutual Exclusion (Token Ring Algorithm):

Let’s move on to Practical 6: Mutual Exclusion (Token Ring Algorithm). This practical involves implementing the Token Ring Algorithm to achieve mutual exclusion in a distributed system.

Objective

To implement a token-based mutual exclusion algorithm where processes in a ring topology pass a token to coordinate access to a shared resource.

Key Concepts

1. Mutual Exclusion:

  - Ensures that only one process accesses a shared resource at a time.

  - Prevents race conditions and ensures consistency.

2. Token Ring Algorithm:

  - Processes are arranged in a logical ring.

  - A token is passed around the ring.

  - Only the process holding the token can access the shared resource.

Step-by-Step Implementation

Step 1: Set Up the Project

1. Create a New Java Project:

   - Open Eclipse → `File → New → Java Project`.

   - Name: `TokenRingMutualExclusion`.

2. Create a Package:

   - Right-click `src` → `New → Package`.

   - Name: `com.tokenring`.

Step 2: Implement the Token Ring Algorithm

1. Create the `Process` Class:

   - This class represents a process in the ring.

   - Create a file named `Process.java` in the `com.tokenring` package:

```
package com.tokenring;

public class Process extends Thread {
    private int id;
    private Process nextProcess;
    private boolean hasToken;
    private boolean inCriticalSection;
```

```
public Process(int id) {
    this.id = id;
    this.hasToken = false;
    this.inCriticalSection = false;
}

public void setNextProcess(Process nextProcess) {
    this.nextProcess = nextProcess;
}

public void setToken(boolean hasToken) {
    this.hasToken = hasToken;
}

@Override
public void run() {
    while (true) {
        if (hasToken) {
            enterCriticalSection();
            passToken();
        }
        try {
            Thread.sleep(1000); // Simulate processing time
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```java
        }
    }

    private void enterCriticalSection() {
        inCriticalSection = true;
        System.out.println("Process " + id + " is in the critical section.");
        try {
            Thread.sleep(2000); // Simulate critical section work
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        inCriticalSection = false;
        System.out.println("Process " + id + " exited the critical section.");
    }

    private void passToken() {
        hasToken = false;
        nextProcess.setToken(true);
        System.out.println("Process " + id + " passed the token to Process " + nextProcess.id);
    }
}
```

2. Create the `Main` Class:

- This class sets up the ring of processes and starts the simulation.
- Create a file named `Main.java` in the `com.tokenring` package:

```
package com.tokenring;


public class Main {
    public static void main(String[] args) {
        // Create processes
        Process process1 = new Process(1);
        Process process2 = new Process(2);
        Process process3 = new Process(3);


        // Set up the ring
        process1.setNextProcess(process2);
        process2.setNextProcess(process3);
        process3.setNextProcess(process1);


        // Start with the token at Process 1
        process1.setToken(true);


        // Start the processes
        process1.start();
        process2.start();
        process3.start();
    }
}
```

Step 3: Run the Program

1. Compile the Code:

   - Right-click the project → `Build Project`.

2. Run the Program:

   - Right-click `Main.java` → `Run As → Java Application`.

Expected Output

```
Process 1 is in the critical section.

Process 1 exited the critical section.

Process 1 passed the token to Process 2

Process 2 is in the critical section.

Process 2 exited the critical section.

Process 2 passed the token to Process 3

Process 3 is in the critical section.

Process 3 exited the critical section.

Process 3 passed the token to Process 1

...
```

How It Works

1. Ring Setup:

- Processes are arranged in a ring (e.g., Process 1 → Process 2 → Process 3 → Process 1).

2. Token Passing:

- The token is passed around the ring.

- Only the process holding the token can enter the critical section.

3. Critical Section:

- The process holding the token accesses the shared resource (simulated by a sleep).

4. Token Release:

- After exiting the critical section, the process passes the token to the next process.

Key Points

- Token: Acts as a permission to access the shared resource.

- Ring Topology: Ensures fairness and prevents starvation.

- Mutual Exclusion: Only one process can be in the critical section at a time.

7. Election Algorithms (Bully and Ring):

Let's proceed to Practical 7: Election Algorithms (Bully and Ring). This practical involves implementing Bully and Ring Election Algorithms to elect a coordinator in a distributed system.

Objective

To implement Bully and Ring Election Algorithms for leader election in a distributed system.

Key Concepts

1. Leader Election:

   - A process is elected as the coordinator to manage the system.

   - Ensures fault tolerance and coordination.

2. Bully Algorithm:

   - The process with the highest ID wins the election.

   - Processes send election messages to higher-ID processes.

3. Ring Algorithm:

   - Processes are arranged in a logical ring.

   - Election messages are passed around the ring until the highest-ID process is elected.

Step-by-Step Implementation

Step 1: Set Up the Project

1. Create a New Java Project:

   - Open Eclipse → `File → New → Java Project`.

   - Name: `ElectionAlgorithms`.

2. Create a Package:

   - Right-click `src` → `New → Package`.

   - Name: `com.election`.

Step 2: Implement the Bully Algorithm

1. Create the `Process` Class:

   - This class represents a process in the system.

   - Create a file named `Process.java` in the `com.election` package:

```
package com.election;

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

public class Process {
    private int id;
    private boolean isCoordinator;
    private List<Process> processes;

    public Process(int id) {
        this.id = id;
        this.isCoordinator = false;
        this.processes = new ArrayList<>();
    }

    public void addProcess(Process process) {
        processes.add(process);
    }
```

```
    public void startElection() {
        System.out.println("Process " + id + " started an election.");
        for (Process process : processes) {
            if (process.id > this.id) {
                System.out.println("Process " + id + " sent election message to Process " +
process.id);
                if (process.receiveElection(this.id)) {
                    return; // Higher process responded, stop election
                }
            }
        }
        declareVictory();
    }

    public boolean receiveElection(int senderId) {
        if (this.id > senderId) {
            System.out.println("Process " + id + " responded to Process " + senderId);
            startElection();
            return true;
        }
        return false;
    }

    public void declareVictory() {
        this.isCoordinator = true;
```

```java
            System.out.println("Process " + id + " is the new coordinator.");
            for (Process process : processes) {
                process.receiveCoordinator(this.id);
            }
        }

        public void receiveCoordinator(int coordinatorId) {
            this.isCoordinator = false;
            System.out.println("Process " + id + " acknowledged Process " + coordinatorId + " as coordinator.");
        }
    }
```

2. Create the `Main` Class:

   - This class sets up the processes and starts the election.

   - Create a file named `Main.java` in the `com.election` package:

```java
    package com.election;

    public class Main {
        public static void main(String[] args) {
            // Create processes
            Process process1 = new Process(1);
            Process process2 = new Process(2);
            Process process3 = new Process(3);
```

```java
        // Add processes to each other
        process1.addProcess(process2);
        process1.addProcess(process3);
        process2.addProcess(process1);
        process2.addProcess(process3);
        process3.addProcess(process1);
        process3.addProcess(process2);

        // Start the election from Process 1
        process1.startElection();
    }
}
```

Step 3: Run the Bully Algorithm

1. Compile the Code:

   - Right-click the project → `Build Project`.

2. Run the Program:

   - Right-click `Main.java` → `Run As → Java Application`.

Expected Output (Bully Algorithm)

```
Process 1 started an election.
Process 1 sent election message to Process 2
Process 2 responded to Process 1
Process 2 started an election.
Process 2 sent election message to Process 3
Process 3 responded to Process 2
Process 3 started an election.
Process 3 is the new coordinator.
Process 1 acknowledged Process 3 as coordinator.
Process 2 acknowledged Process 3 as coordinator.
```

Step 4: Implement the Ring Algorithm

1. Create the `RingProcess` Class:

   - This class represents a process in the ring.

   - Create a file named `RingProcess.java` in the `com.election` package:

```
package com.election;

public class RingProcess extends Thread {
    private int id;
    private RingProcess nextProcess;
```

```
private boolean isCoordinator;
private int[] electionMessage;

public RingProcess(int id) {
    this.id = id;
    this.isCoordinator = false;
    this.electionMessage = new int[0];
}

public void setNextProcess(RingProcess nextProcess) {
    this.nextProcess = nextProcess;
}

public void startElection() {
    System.out.println("Process " + id + " started an election.");
    electionMessage = new int[] { id };
    nextProcess.receiveElection(electionMessage);
}

public void receiveElection(int[] message) {
    if (message.length == 0) {
        declareVictory();
        return;
    }

    int maxId = message[0];
```

```
        for (int id : message) {
            if (id > maxId) maxId = id;
        }

        if (maxId == this.id) {
            declareVictory();
        } else {
            int[] newMessage = new int[message.length + 1];
            System.arraycopy(message, 0, newMessage, 0, message.length);
            newMessage[message.length] = this.id;
            nextProcess.receiveElection(newMessage);
        }
    }

    public void declareVictory() {
        this.isCoordinator = true;
        System.out.println("Process " + id + " is the new coordinator.");
        nextProcess.receiveCoordinator(this.id);
    }

    public void receiveCoordinator(int coordinatorId) {
        this.isCoordinator = false;
        System.out.println("Process " + id + " acknowledged Process " + coordinatorId + " as
coordinator.");
        if (coordinatorId != this.id) {
            nextProcess.receiveCoordinator(coordinatorId);
```

```
        }
    }
}
```

2. Update the `Main` Class:

- Modify `Main.java` to use the `RingProcess` class:

```
package com.election;

public class Main {
    public static void main(String[] args) {
        // Create processes
        RingProcess process1 = new RingProcess(1);
        RingProcess process2 = new RingProcess(2);
        RingProcess process3 = new RingProcess(3);

        // Set up the ring
        process1.setNextProcess(process2);
        process2.setNextProcess(process3);
        process3.setNextProcess(process1);

        // Start the election from Process 1
        process1.startElection();
    }
}
```

Step 5: Run the Ring Algorithm

1. Compile the Code:

   - Right-click the project → `Build Project`.

2. Run the Program:

   - Right-click `Main.java` → `Run As → Java Application`.

Expected Output (Ring Algorithm)

```
Process 1 started an election.
Process 2 received election message: [1]
Process 3 received election message: [1, 2]
Process 1 received election message: [1, 2, 3]
Process 1 is the new coordinator.
Process 2 acknowledged Process 1 as coordinator.
Process 3 acknowledged Process 1 as coordinator.
```

How It Works

1. Bully Algorithm:

   - The process with the highest ID wins the election.

   - Processes send election messages to higher-ID processes.

2. Ring Algorithm:

   - Processes are arranged in a ring.

   - Election messages are passed around the ring until the highest-ID process is elected.

---

8. Web Services:

Let’s proceed to Practical 8: Web Services. This practical involves creating a simple web service and writing a distributed application to consume it.

Objective

To create a web service using Java and write a client application to consume the service.

Key Concepts

1. Web Service:

   - A service exposed over the web using standard protocols (e.g., HTTP, SOAP, REST).

   - Allows applications to communicate over a network.

2. SOAP vs REST:

   - SOAP: Uses XML for messaging and follows a strict protocol.

   - REST: Uses HTTP methods (GET, POST, etc.) and is lightweight.

3. JAX-WS:

- Java API for XML Web Services (used for SOAP-based services).

Step-by-Step Implementation

Step 1: Set Up the Project

1. Create a New Java Project:

- Open Eclipse → `File → New → Java Project`.

- Name: `WebServicePractical`.

2. Create a Package:

- Right-click `src` → `New → Package`.

- Name: `com.webservice`.

Step 2: Create the Web Service

1. Create the Service Interface:

- Create a file named `CalculatorService.java` in the `com.webservice` package:

```
package com.webservice;

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

```java
@WebService
public interface CalculatorService {
    @WebMethod
    double add(double x, double y);

    @WebMethod
    double subtract(double x, double y);

    @WebMethod
    double multiply(double x, double y);

    @WebMethod
    double divide(double x, double y);
}
```

2. Create the Service Implementation:

  - Create a file named `CalculatorServiceImpl.java` in the `com.webservice` package:

```java
package com.webservice;

import javax.jws.WebService;

@WebService(endpointInterface = "com.webservice.CalculatorService")
public class CalculatorServiceImpl implements CalculatorService {
    public double add(double x, double y) {
```

```
        return x + y;
    }

    public double subtract(double x, double y) {
        return x - y;
    }

    public double multiply(double x, double y) {
        return x * y;
    }

    public double divide(double x, double y) {
        if (y == 0) throw new ArithmeticException("Division by zero");
        return x / y;
    }
}
```

3. Publish the Web Service:

   - Create a file named `CalculatorServicePublisher.java` in the `com.webservice` package:

```
package com.webservice;

import javax.xml.ws.Endpoint;

public class CalculatorServicePublisher {
```

```java
    public static void main(String[] args) {
        Endpoint.publish("http://localhost:8080/ws/calculator", new CalculatorServiceImpl());
        System.out.println("Service is published at http://localhost:8080/ws/calculator");
    }
}
```

Step 3: Run the Web Service

1. Compile the Code:

   - Right-click the project → `Build Project`.

2. Run the Service:

   - Right-click `CalculatorServicePublisher.java` → `Run As → Java Application`.

   - Output: `Service is published at http://localhost:8080/ws/calculator`.

Step 4: Create the Client Application

1. Generate Client Stubs:

   - Open a command prompt and navigate to the `src` folder:

```cmd
cd C:\Users\<YourName>\eclipse-workspace\WebServicePractical\src
```

   - Run the `wsimport` tool to generate client stubs:

cmd

```
wsimport -keep http://localhost:8080/ws/calculator?wsdl
```

- This generates client-side classes in the `com.webservice` package.

2. Create the Client:

- Create a file named `CalculatorClient.java` in the `com.webservice` package:

```java
package com.webservice;

public class CalculatorClient {
    public static void main(String[] args) {
        CalculatorServiceImplService service = new CalculatorServiceImplService();
        CalculatorService calculator = service.getCalculatorServiceImplPort();

        System.out.println("5 + 3 = " + calculator.add(5, 3));
        System.out.println("5 - 3 = " + calculator.subtract(5, 3));
        System.out.println("5 * 3 = " + calculator.multiply(5, 3));
        System.out.println("5 / 3 = " + calculator.divide(5, 3));
    }
}
```

Step 5: Run the Client

1. Compile the Code:

   - Right-click the project → `Build Project`.

2. Run the Client:

   - Right-click `CalculatorClient.java` → `Run As → Java Application`.

Expected Output

```
5 + 3 = 8.0

5 - 3 = 2.0

5 * 3 = 15.0

5 / 3 = 1.6666666666666667
```

How It Works

1. Web Service:

   - The service is published at `http://localhost:8080/ws/calculator`.

   - It exposes methods for addition, subtraction, multiplication, and division.

2. Client:

   - Uses the `wsimport` tool to generate client stubs.

   - Invokes the web service methods using the stubs.
