LAB 8: Write a program to Create CORBA based server-client application in JAVA.

Theory:

This experiment involves a client-server interaction using CORBA (Common Object Request Broker Architecture). The server, implemented in the `EchoServer` class, extends `EchoPOA` and initializes the Object Request Broker (ORB), Portable Object Adapter (POA), and a servant. The server registers the servant in the naming service, allowing it to be located by a specified name. The client, in the `Client` class, initializes the ORB, obtains the naming service reference, and looks up the server object using the registered name. The client then invokes the `echoString` method on the server, printing the result. This experiment highlights key CORBA concepts, such as ORB initialization, servant creation, object binding in the naming service, and remote method invocation, showcasing the fundamentals of distributed object communication.

Code:

Client.java

```
import EchoApp.Echo;
import EchoApp.EchoHelper;
import org.omg.CORBA.ORB;
import org.omg.CORBA.ORBPackage.InvalidName;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.CosNaming.NamingContextPackage.CannotProceed;
import org.omg.CosNaming.NamingContextPackage.NotFound;

public class Client {

   public static void main(String args[]) {
      try {
         // create and initialize the ORB
         ORB orb = ORB.init(args, null);
         org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
         NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
         Echo href = EchoHelper.narrow(ncRef.resolve_str("ECHO-SERVER"));

         String hello = href.echoString();
         System.out.println(hello);
      } catch (InvalidName invalidName) {
         invalidName.printStackTrace();
```

```
        } catch (CannotProceed cannotProceed) {
            cannotProceed.printStackTrace();
        } catch (org.omg.CosNaming.NamingContextPackage.InvalidName invalidName) {
            invalidName.printStackTrace();
        } catch (NotFound notFound) {
            notFound.printStackTrace();
        }

    }

}
```

Output:

Server.java

```
import EchoApp.Echo;
import EchoApp.EchoHelper;
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 Server {

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

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

            // create servant
            EchoServer server = new EchoServer();
```

```
        // get object reference from the servant
        org.omg.CORBA.Object ref = rootpoa.servant_to_reference(server);
        Echo href = EchoHelper.narrow(ref);

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

        NameComponent path[] = ncRef.to_name( "ECHO-SERVER" );
        ncRef.rebind(path, href);

        System.out.println("Server ready and waiting ...");

        // wait for invocations from clients
        orb.run();
      }

      catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }

      System.out.println("Exiting ...");

    }
}
```

Output:

```
Server ready and waiting ...
```

LAB 6: Write a program to Simulate the Distributed Mutual Exclusion in JAVA

Theory:
Lamport's logical clocks are a mechanism to order events in a distributed system. Each process maintains a logical clock that is incremented with each local event and updated when a message is sent or received. The logical timestamps assigned to events provide a consistent ordering across all processes, facilitating coordination. In this simulation, each process simulates a request for the critical section by broadcasting a message with its logical timestamp to other processes. Upon receiving messages from all other processes, a process enters the critical section, executes the critical section code, and exits. The use of logical clocks ensures that processes can make informed decisions about granting access based on the ordering of events in the system. This experiment serves as a hands-on exploration of Lamport's logical clocks and distributed mutual exclusion concepts, providing insights into the challenges and solutions for synchronization in distributed systems.

Code:

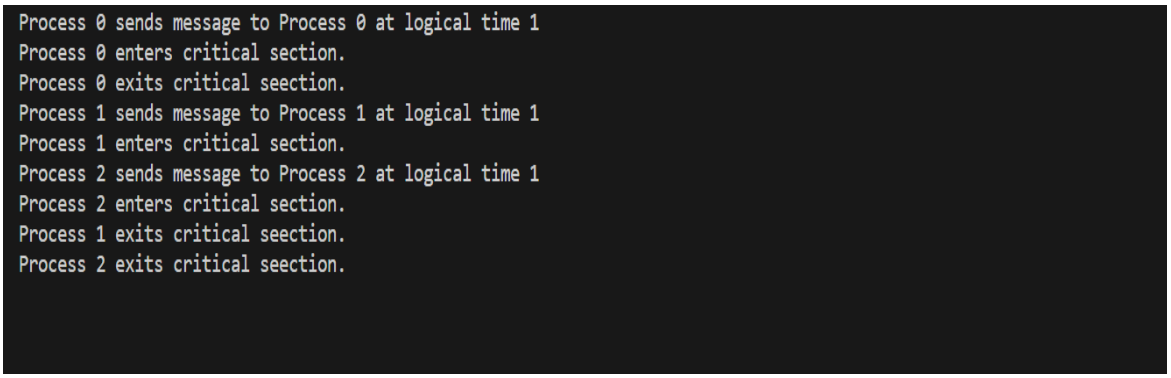
```
import java.util.PriorityQueue;
import java.util.Queue;

public class LamportMutexSimulation1 {

    public static void main(String[] args) {
        int numProcesses = 3;
        Process[] processes = new Process[numProcesses];

        for (int i = 0; i < numProcesses; i++) {
            processes[i] = new Process(i);
        }

        for (int i = 0; i < numProcesses; i++) {
            for (int j = 0; j < numProcesses; j++) {
                if (i != j) {
                    processes[i].addMessageToQueue(new Message(j, 0));
                }
            }
        }

        Thread[] threads = new Thread[numProcesses];
        for (int i = 0; i < numProcesses; i++) {
```

```
            threads[i] = new Thread(processes[i]);
            threads[i].start();
        }
    }
}

class Process implements Runnable {
    private int processId;
    private int logicalClock;
    private Queue<Message> messageQueue;

    public Process(int processId) {
        this.processId = processId;
        this.logicalClock = 0;
        this.messageQueue = new PriorityQueue<>();
    }

    private void sendMessage(int toProcessId) {
        logicalClock++;
        Message message = new Message(processId, logicalClock);
        System.out.println("Process " + processId + " sends message to Process " + toProcessId
+ " at logical time " + logicalClock);
        // In a real distributed system, you would send this message to the destination process.
    }

    private void receiveMessage(Message message) {
        logicalClock = Math.max(logicalClock, message.getTimestamp()) + 1;
        System.out.println("Process " + processId + " receives message from Process " +
message.getProcessId() + " at logical time " + logicalClock);
        // In a real distributed system, you would process the received message.
    }

    private void requestCriticalSection() {
        sendMessage(processId); // Broadcast request to all processes
        // Wait for replies from all other processes
        while (messageQueue.size() < 2) {
            Thread.yield(); // Simulate waiting
        }
        enterCriticalSection();
    }

    private void enterCriticalSection() {
        System.out.println("Process " + processId + " enters critical section.");
        // Simulate critical section work
```

```
        try {
            Thread.sleep((int) (Math.random() * 1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Process " + processId + " exits critical seection.");
    }

    public void run() {
        // Simulate some non-critical section work
        try {
            Thread.sleep((int) (Math.random() * 1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        requestCriticalSection();
    }

    public void addMessageToQueue(Message message) {
        messageQueue.add(message);
    }
}

class Message implements Comparable<Message> {
    private int processId;
    private int timestamp;

    public Message(int processId, int timestamp) {
        this.processId = processId;
        this.timestamp = timestamp;
    }

    public int getProcessId() {
        return processId;
    }

    public int getTimestamp() {
        return timestamp;
    }

    @Override
    public int compareTo(Message other) {
        return this.timestamp - other.timestamp;
```

```
    }
}
```

Output:


Process 0 sends message to Process 0 at logical time 1
Process 0 enters critical section.
Process 0 exits critical seection.
Process 1 sends message to Process 1 at logical time 1
Process 1 enters critical section.
Process 2 sends message to Process 2 at logical time 1
Process 2 enters critical section.
Process 1 exits critical seection.
Process 2 exits critical seection.