LAB1: Implementation of socket programming using java programming language

A basic implementation of a client-server model using Java's '`Socket`' and '`ServerSocket`' classes

INTRODUCTION TO DISTRIBUTED SYSTEMS
LAB WORKOUTS
YEAR 4 SECTION 1 & 2

Server Code:

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

public class Server {
    public static void main(String[] args) {
        try {
            // Create a ServerSocket to listen for connections
            ServerSocket serverSocket = new ServerSocket(12345);
            System.out.println("Server is listening for connections...");

            // Accept incoming connections
            Socket clientSocket = serverSocket.accept();
            System.out.println("Connection established with client: " + clientSocket);

            // Create input and output streams
            BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

            // Read messages from the client and send responses
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println("Received: " + inputLine);
                out.println("Server received: " + inputLine);
            }

            // Clean up
            in.close();
            out.close();
            clientSocket.close();
            serverSocket.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Client Code:

```
import java.io.*;
import java.net.*;
public class Client {
    public static void main(String[] args) {
        try {
            // Connect to the server
            Socket socket = new Socket("localhost", 12345);

            // Create input and output streams
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

            // Send messages to the server
            out.println("Hello, server!");

            // Receive responses from the server
            String response;
            while ((response = in.readLine()) != null) {
                System.out.println("Server response: " + response);
            }
            // Clean up
            out.close();
            in.close();
            socket.close();

        } catch (IOException e) {
            e.printStackTrace();
        } } }
```

How to Run:

1. Compile both **Server.java** and **Client.java** files using **javac Server.java** and **javac Client.java**.
2. Start the server by running **java Server** in one terminal window.
3. Start the client by running **java Client** in another terminal window.
4. You should see the client message being received by the server, and the server responding back to the client.

Lab 2: Implementation of Client Server based program using RMI using java programming language

RMI (Remote Method Invocation) is a Java API that enables an object running in one Java virtual machine (JVM) to invoke methods on an object running in another JVM. Here's an example of a simple client-server program using RMI:

Server Code:

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

// Define the remote interface
interface HelloInterface extends Remote {
  String sayHello() throws RemoteException;
}

// Implement the remote interface
public class HelloImpl extends UnicastRemoteObject implements HelloInterface {
  public HelloImpl() throws RemoteException {}

  // Implement the method defined in the remote interface
  public String sayHello() throws RemoteException {
    return "Hello, client!";
  }

  // Main method to create the RMI registry and bind the remote object
  public static void main(String args[]) {
    try {
      HelloImpl obj = new HelloImpl();
      Naming.rebind("HelloServer", obj);
      System.out.println("HelloServer bound in registry");
    } catch (Exception e) {
      System.out.println("HelloImpl err: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Client Code:

```
import java.rmi.*;

public class HelloClient {
  public static void main(String args[]) {
    try {
      // Lookup the remote object
      HelloInterface obj = (HelloInterface) Naming.lookup("//localhost/HelloServer");

      // Call the remote method
      System.out.println(obj.sayHello());
    } catch (Exception e) {
      System.out.println("HelloClient exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Steps to Run:

1. Compile both **HelloImpl.java** and **HelloClient.java** files using **javac**.
2. Start the RMI registry by running **rmiregistry** in a terminal window.
3. In another terminal window, start the server by running **java HelloImpl**.
4. Finally, in another terminal window, start the client by running **java HelloClient**.
5. You should see the client output: "Hello, client!".

This is a simple example of RMI in Java. In a real-world scenario, you would typically have more complex interfaces and implementations, error handling, security considerations, etc.

Lab 3: Implementation of Client Server based program using RPC

RPC (Remote Procedure Call) is a protocol that one program can use to request a service from a program located on another computer on a network without having to understand the network's details. Here's an example of a simple client-server program using RPC in Java, utilizing the Java RMI (Remote Method Invocation) framework:

Server Code:

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

// Define the remote interface
interface HelloInterface extends Remote {
  String sayHello() throws RemoteException;
}

// Implement the remote interface
public class HelloImpl extends UnicastRemoteObject implements HelloInterface {
  public HelloImpl() throws RemoteException {}

  // Implement the method defined in the remote interface
  public String sayHello() throws RemoteException {
    return "Hello, client!";
  }

  // Main method to create the RMI registry and bind the remote object
  public static void main(String args[]) {
    try {
      HelloImpl obj = new HelloImpl();
      Naming.rebind("rmi://localhost/HelloServer", obj);
      System.out.println("HelloServer bound in registry");
    } catch (Exception e) {
      System.out.println("HelloImpl err: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Client Code:

```
import java.rmi.*;

public class HelloClient {
  public static void main(String args[]) {
    try {
      // Lookup the remote object
      HelloInterface obj = (HelloInterface) Naming.lookup("rmi://localhost/HelloServer");

      // Call the remote method
      System.out.println(obj.sayHello());
    } catch (Exception e) {
      System.out.println("HelloClient exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Steps to Run:

1. Compile both HelloImpl.java and HelloClient.java files using javac.
2. Start the RMI registry by running rmiregistry in a terminal window.
3. In another terminal window, start the server by running java HelloImpl.
4. Finally, in another terminal window, start the client by running java HelloClient.
5. You should see the client output: "Hello, client!".

This example demonstrates RPC using Java RMI, which is a higher-level abstraction of RPC. It hides many of the complexities involved in implementing remote procedure calls over a network.

Lab 4: Implementation of Clock Synchronization (logical/physical)

Clock synchronization can be implemented using either logical clocks or physical clocks. Logical clocks, such as Lamport clocks or vector clocks, order events based on their logical relationship. Physical clocks, on the other hand, aim to synchronize the time across different machines using the time reported by hardware clocks or NTP (Network Time Protocol).

A basic example of clock synchronization using Lamport logical clocks in Java:

Logical Clock Synchronization using Lamport Clocks:

```
import java.util.concurrent.atomic.AtomicInteger;

class LamportClock {
    private AtomicInteger time = new AtomicInteger(0);

    public int getTime() {
        return time.get();
    }

    public void tick() {
        time.incrementAndGet();
    }

    public void update(int receivedTime) {
        int current = time.get();
        int newTime = Math.max(current, receivedTime) + 1;
        time.set(newTime);
    }
}

class Process {
    private int processId;
    private LamportClock clock;

    public Process(int processId, LamportClock clock) {
        this.processId = processId;
        this.clock = clock;
    }

    public void sendEvent() {
```

```
        clock.tick();
        System.out.println("Process " + processId + " sent an event at time " + clock.getTime());
    }

    public void receiveEvent(int senderTime) {
        clock.update(senderTime);
        clock.tick();
        System.out.println("Process " + processId + " received an event at time " + clock.getTime());
    }
}

public class ClockSynchronization {
    public static void main(String[] args) {
        LamportClock clock1 = new LamportClock();
        LamportClock clock2 = new LamportClock();
        Process process1 = new Process(1, clock1);
        Process process2 = new Process(2, clock2);

        // Simulate events
        process1.sendEvent();
        process2.receiveEvent(clock1.getTime()); // Synchronize clocks
        process2.sendEvent();
        process1.receiveEvent(clock2.getTime()); // Synchronize clocks
    }
}
```

This code simulates two processes communicating events and synchronizing their logical clocks using Lamport clocks. The **LamportClock** class provides methods for tick and update operations, while the **Process** class represents a process with its associated Lamport clock.

In a real-world scenario, you would need to handle network communication for clock synchronization across multiple processes or machines. Additionally, for physical clock synchronization, you might use NTP or other time synchronization protocols.

Lab 5: Implementation of Election algorithm

A simple implementation of the Bully Election Algorithm in Java:

```
import java.util.*;
class Process {
    private int id;
    private boolean active;

    public Process(int id) {
        this.id = id;
        this.active = true;
    }

    public int getId() {
        return id;
    }

    public boolean isActive() {
        return active;
    }

    public void deactivate() {
        active = false;
    }

    public void sendMessage(Process receiver) {
        System.out.println("Process " + id + " sends election message to Process " +
receiver.getId());
        receiver.receiveMessage(this);
    }

    public void receiveMessage(Process sender) {
        if (sender.getId() > id) {
            System.out.println("Process " + id + " receives election message from Process " +
sender.getId());
            sendMessage(sender);
        } else {
```

```
            System.out.println("Process " + id + " acknowledges election message from
Process " + sender.getId());
            sender.deactivate();
        }
    }
}

public class ElectionAlgorithm {
    public static void main(String[] args) {
        // Create processes
        Process[] processes = new Process[5];
        for (int i = 0; i < processes.length; i++) {
            processes[i] = new Process(i + 1);
        }
        // Simulate an election
        for (Process process : processes) {
            if (process.isActive()) {
                System.out.println("Process " + process.getId() + " starts election");
                for (Process otherProcess : processes) {
                    if (otherProcess.isActive() && otherProcess.getId() > process.getId()) {
                        process.sendMessage(otherProcess);
                    }
                }
            }
        }
        // Find the coordinator
        Process coordinator = null;
        for (Process process : processes) {
            if (process.isActive()) {
                coordinator = process;
                break;
            }
        }

        // Print the coordinator
        System.out.println("Coordinator: Process " + coordinator.getId());
    }
}
```

This code simulates the Bully Election Algorithm among a set of processes. Each process has an ID and a status (active or inactive). Processes send election messages to higher-ranked processes. If a lower-ranked process receives a message from a higher-ranked process, it forwards the message to an even higher-ranked process. If a process does not receive any message, it declares itself as the coordinator.

In a real-world scenario, you may need to extend this implementation to handle network communication, timeouts, failures, and recovery scenarios. Additionally, you may consider implementing other election algorithms like the Ring Election Algorithm or the Chang and Roberts Algorithm.

Lab 6: Implementation of Mutual Exclusion algorithms

An implementation of the Peterson's Algorithm for mutual exclusion in Java:

```
import java.util.concurrent.atomic.AtomicInteger;

class PetersonMutex {
   private volatile boolean[] flag = new boolean[2];
   private volatile int turn;

   public void lock(int id) {
      int other = 1 - id;
      flag[id] = true;
      turn = other;
      while (flag[other] && turn == other) {
         // Wait until it's our turn and the other process is not in the critical section
      }
   }

   public void unlock(int id) {
      flag[id] = false;
   }
}

class Process extends Thread {
   private static AtomicInteger counter = new AtomicInteger(0);
   private int id;
   private PetersonMutex mutex;

   public Process(PetersonMutex mutex) {
      this.id = counter.getAndIncrement();
      this.mutex = mutex;
   }

   public void run() {
      for (int i = 0; i < 5; i++) {
         mutex.lock(id);
         System.out.println("Process " + id + " is in the critical section");
         try {
            Thread.sleep((int) (Math.random() * 1000)); // Simulate some work
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            mutex.unlock(id);
            System.out.println("Process " + id + " is out of the critical section");
            try {
                Thread.sleep((int) (Math.random() * 1000)); // Simulate some work
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class MutualExclusion {
    public static void main(String[] args) {
        PetersonMutex mutex = new PetersonMutex();
        Process p1 = new Process(mutex);
        Process p2 = new Process(mutex);

        p1.start();
        p2.start();
    }
}
```

In this implementation, we have a class **PetersonMutex** which implements Peterson's algorithm for mutual exclusion. Each process calls **lock(id)** to enter the critical section and **unlock(id)** to exit the critical section.

The **Process** class represents a thread that runs and tries to access the critical section. We create two processes **p1** and **p2** that share the same mutex. When run, these processes try to access the critical section while ensuring mutual exclusion using Peterson's algorithm.

In a real-world scenario, you would typically use more advanced synchronization mechanisms provided by Java, such as **java.util.concurrent.locks.ReentrantLock** or **java.util.concurrent.Semaphore**. However, Peterson's algorithm serves as a simple example to understand the concept of mutual exclusion.

Lab 7: Implementation of multi-threaded client/server processes

a simple implementation of a multi-threaded client-server application in Java using sockets:

Server Code:

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

public class Server {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(12345);
            System.out.println("Server started. Waiting for clients...");

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Client connected: " + clientSocket.getInetAddress());

                // Create a new thread for each client
                ClientHandler clientHandler = new ClientHandler(clientSocket);
                clientHandler.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ClientHandler extends Thread {
    private Socket clientSocket;

    public ClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    public void run() {
        try {
```

```
        BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

        String message;
        while ((message = in.readLine()) != null) {
          System.out.println("Message from client: " + message);
          out.println("Echo: " + message);
        }

        clientSocket.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
}
```

Client Code:

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

public class Client {
   public static void main(String[] args) {
      try {
         Socket socket = new Socket("localhost", 12345);
         BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
         PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

         // Send messages to the server
         out.println("Hello, server!");
         out.println("How are you?");
         out.println("Bye!");

         // Receive responses from the server
         String response;
         while ((response = in.readLine()) != null) {
            System.out.println("Server response: " + response);
         }
```

```
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

This implementation creates a server that listens for incoming connections from clients. When a client connects, the server creates a new thread (**ClientHandler**) to handle communication with that client. The **ClientHandler** thread reads messages from the client, echoes them back, and continues to listen for new messages until the client disconnects.

The client connects to the server, sends multiple messages, and prints the server's responses.

This approach allows the server to handle multiple clients concurrently by creating a separate thread for each client.