NAME: Gaven Dcosta	ROLL NO: 09
BATCH: B	DATE: 8/1/2024

Distributed Computing

Experiment – 1

Program to demonstrate Inter- Process communication using Java	
Learning Objective:	To understand basic underlying concepts of forming distributed systems.
Learning Outcome:	students will be able to demonstrate Inter-Process communication.
Course Outcome:	**CSL801.1**
Program Outcome:	(PO1) Engineering knowledge : Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems. (PO2) Problem Analysis : Identify, Formulate, review research literature and analyze complex engineering problems reaching substantiated conclusion using first principles of mathematics, natural science and engineering sciences.
Bloom's Taxonomy Level:	Analysis, Create
Theory:	**<u>Inter-Process Communication</u>** Inter-Process Communication (IPC) is a crucial aspect of distributed computing, enabling communication and coordination between processes running on different nodes within a distributed system. 1. Definition of IPC: Inter-Process Communication (IPC) refers to the mechanisms and techniques that allow processes to exchange information, coordinate their activities, and synchronize their execution. 2. Purpose of IPC in Distributed Computing: Coordination: Processes in a distributed system need to coordinate their actions to

achieve common goals. IPC facilitates the sharing of information and synchronization between distributed processes.

Communication: Distributed applications often consist of multiple processes that need to communicate to achieve a specific task or provide a service.

3. Examples of IPC in Distributed Systems:

Java RMI (Remote Method Invocation): A Java-based RPC mechanism.

Message-oriented Middleware (MOM): Systems like Apache Kafka and RabbitMQ provide message queues for communication.

Advantages of IPC -

1. Parallelism and Concurrency: IPC enables parallelism by allowing multiple processes to execute simultaneously. It facilitates concurrent execution and improves overall system performance.
2. Modularity and Decoupling: Processes can be designed as modular components, each performing a specific task. IPC allows for the decoupling of these components, making the system more maintainable and scalable.
3. Communication and Collaboration: IPC facilitates communication and collaboration between different processes, which is crucial for distributed systems where components often need to work together to achieve a common goal.
4. Resource Sharing: Processes can share resources, such as memory or data, through IPC mechanisms like shared memory. This can lead to more efficient resource utilization.
5. Fault Isolation: In the event of a failure in one process, IPC mechanisms can help isolate the impact, preventing the failure from affecting other parts of the system.

Disadvantages of IPC -

1. Complexity: Implementing IPC mechanisms can introduce complexity to the system. Developers need to carefully manage synchronization, message passing, and error handling.
2. Synchronization Issues: Coordinating access to shared resources through IPC requires careful synchronization to avoid race conditions, deadlocks,

and other concurrency issues.

3. Security Concerns: IPC introduces potential security risks, especially when processes communicate over networks. Unauthorized access or interception of messages can lead to data breaches.
4. Dependency on Communication: Processes relying heavily on IPC may become highly dependent on the performance and reliability of the communication channels. Failures in communication can impact the overall system functionality.
5. Compatibility Challenges: Ensuring compatibility between different versions of software or between processes developed by different teams can be challenging when using IPC.

Types of IPC –

1. Message Passing:

Synchronous Message Passing: Processes communicate in a blocking manner. The sender waits until the receiver acknowledges the message.

Examples include RPC (Remote Procedure Call) and synchronous message queues.

Asynchronous Message Passing: Processes communicate without waiting for each other.

Examples include message queues (asynchronous), publish-subscribe systems, and event-driven architectures.

2. Shared Memory:

Mapped Memory: Processes share a common memory region. Changes made by one process are immediately visible to others.

Requires synchronization mechanisms (e.g., locks) to avoid conflicts.

Examples include memory-mapped files and shared data structures.

Distributed Shared Memory (DSM): Extends the concept of shared memory to a distributed environment. Allows processes to access a shared address space, even if physically located on different machines.

3. Sockets:

Network Sockets: Processes communicate over a network using sockets.

Commonly used in client-server architectures and distributed systems.

Examples include TCP/IP sockets, UDP sockets.

Unix Domain Sockets: Processes communicate within the same machine using a special file. Provides a faster communication channel compared to network sockets for local communication.

4. Remote Procedure Call (RPC):

Synchronous RPC: Similar to a local procedure call but involves executing code on a remote machine. Uses client and server stubs to hide the complexities of network communication.

Examples include Java RMI, CORBA.

Asynchronous RPC: Allows non-blocking communication between processes. Useful for scenarios where waiting for a response might be inefficient.

5. Message Queues:

Point-to-Point Queues: One-to-one communication using message queues. Processes send and receive messages through a queue.

Examples include IBM MQ, RabbitMQ.

Publish-Subscribe Queues: One-to-many communication using message queues.

Publishers send messages to a topic, and multiple subscribers can receive messages from that topic.

Examples include Apache Kafka, MQTT.

6. Signals:

Unix Signals: Lightweight notifications sent by the operating system to a process.

Used for simple communication and process control.

	Examples include SIGINT (interrupt signal) and SIGTERM (termination signal). 7. Pipes and FIFOs: Pipes: Unidirectional communication channel between two processes. Typically used for sequential data flow. Examples include Unix pipes. FIFOs (Named Pipes): Similar to pipes but can be used for communication between unrelated processes. Allows processes to communicate by writing and reading from a named pipe. 8. Memory-mapped Files: File-backed Shared Memory: Processes map a file into their address space, allowing them to read and write to the file as if it were memory. Changes made by one process are reflected in the file and visible to others.
Algorithm:	**Sender Process:** 1. Initialize IPC Mechanism: Create or connect to the IPC mechanism (e.g., socket, message queue, shared memory). 2. Prepare Data: Prepare the data or message to be sent to the receiver process. 3. Send Data: Use the IPC mechanism to send the data to the receiver process. 4. Wait for Acknowledgment (Optional): If acknowledgment is required, wait for a response from the receiver indicating successful reception or processing. 5. Cleanup: Close or release resources associated with the IPC mechanism. 6. Receiver Process: Initialize IPC Mechanism: Create or connect to the same IPC mechanism used by the sender process.

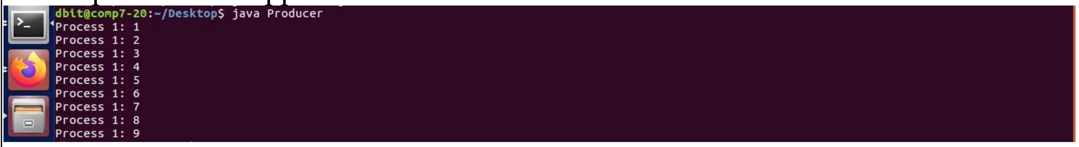
Receive Data:

Use the IPC mechanism to receive the data sent by the sender.

1. Process Data: Process the received data as needed based on the application's requirements.
2. Send Acknowledgment (Optional): If acknowledgment is required, send a response back to the sender to indicate successful reception or processing.
3. Cleanup: Close or release resources associated with the IPC mechanism.

Data Set:

The input file : - mapped.text



```
dbit@comp7-20:~/Desktop$ java Producer
Process 1: 1
Process 1: 2
Process 1: 3
Process 1: 4
Process 1: 5
Process 1: 6
Process 1: 7
Process 1: 8
Process 1: 9
```

Outcome:

Producer Code:

```
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;

public class Producer {
   public static void main(String args[]) throws IOException, InterruptedException
{
       RandomAccessFile rd = new
RandomAccessFile("C:\\Users\\nashr\\OneDrive\\Desktop\\Sem 8\\DC-
LAB\\Experiments\\mapped.txt", "rw");
       FileChannel fc = rd.getChannel();
       MappedByteBuffer mem = fc.map(FileChannel.MapMode.READ_WRITE, 0,
1000);

       try {
          Thread.sleep(10000);
       } catch (InterruptedException e)
          { e.printStackTrace();
       }

       // Write numbers as strings to the memory-mapped file
```

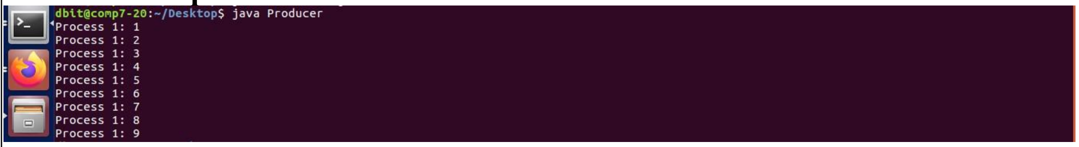
```
        for (int i = 1; i <= 9; i++)
            { String str =
            String.valueOf(i);
            byte[] strBytes = str.getBytes();
            mem.put(strBytes);
            mem.put((byte) '\n'); // Add a newline between strings
            System.out.println("Process 1: " + i);
            Thread.sleep(1000); // Simulate time for CPU cache to refresh
        }

        // Close resources
        fc.close();
        rd.close();
    }
}
```

Consumer Code

```
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;

public class Consumer {
    public static void main(String args[]) throws IOException, InterruptedException
{
        RandomAccessFile rd = new
RandomAccessFile("C:\\Users\\nashr\\OneDrive\\Desktop\\Sem 8\\DC-
LAB\\Experiments\\mapped.txt", "r");
        FileChannel fc = rd.getChannel();
        MappedByteBuffer mem = fc.map(FileChannel.MapMode.READ_ONLY, 0,
1000);

        // Assuming that the producer has already written the data
        StringBuilder currentString = new StringBuilder();

        while (mem.hasRemaining())
            { byte currentByte =
            mem.get(); if (currentByte ==
            (byte) '\n') {
                System.out.println("Process 2: " + currentString.toString());
                currentString.setLength(0); // Clear the StringBuilder for the next string
            } else {
                currentString.append((char) currentByte);
            }
        }

        // Close resources
        fc.close();
        rd.close();
    }
}
```

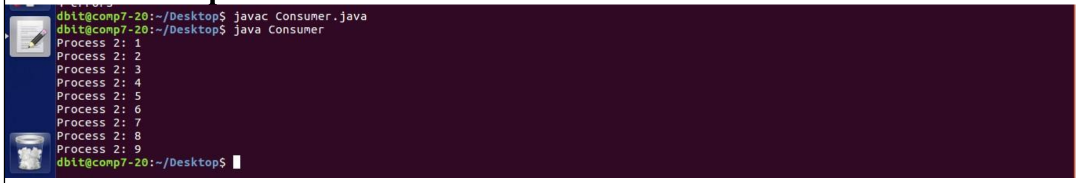
Output

Producer Output



```
dbit@comp7-20:~/Desktop$ java Producer
Process 1: 1
Process 1: 2
Process 1: 3
Process 1: 4
Process 1: 5
Process 1: 6
Process 1: 7
Process 1: 8
Process 1: 9
```

Consumer Output



```
dbit@comp7-20:~/Desktop$ javac Consumer.java
dbit@comp7-20:~/Desktop$ java Consumer
Process 2: 1
Process 2: 2
Process 2: 3
Process 2: 4
Process 2: 5
Process 2: 6
Process 2: 7
Process 2: 8
Process 2: 9
dbit@comp7-20:~/Desktop$ █
```

Conclusion	• Memory-mapped files provide a mechanism for processes to share data by mapping a file directly into memory. • The FileChannel and MappedByteBuffer classes in Java facilitate memory-mapped file operations. • The Producer writes data to the memory-mapped file, simulating the production of information over time. • The Consumer reads and processes the data from the same memory-mapped file. Memory-mapped files provide an efficient and flexible wayfor processes to share data in a concurrent environment.
References	https://www.geeksforgeeks.org/inter-process-communication-ipc/

Rubrics for Assessment

Timely Submission	Submitted after 2 weeks 0	Submitted after deadline 1	On time Submission 2
Understanding	Student is confused about the concept 0	Students has justifiably understood the concept 2	Students is very clear about the concepts 3
Performance	Students has not performed the Experiment 0	Student has performed with help 2	Student has independently performed the experiment 3
Development	Student struggles to write code 0	Student can write code the requirement stated 1	Student can write exceptional code with his own ideas 2

Name : Gaven Dcosta Roll No: 09   Date: 15-01-2025

Batch B

Distributed Computing

Experiment – 2

Program to demonstrate Client/Server application Using RMI	
Learning Objective:	To understand basic underlying concepts of forming distributed systems.
Learning Outcome:	Students will gain ability to demonstrate Client/Server application.
Course Outcome:	**CSL801.1**
Program Outcome:	**(PO**1) **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. **(PO**2) **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences. **(PO**3) **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations
Theory:	*RMI, Advantages and Disadvantages of RMI, Types of RMI* **Remote Method Invocation (RMI)** The RMI is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects stub and skeleton.

Stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

Skeleton

The skeleton is an object, acts as a gateway for the server-side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

Advantages of RMI:

Object-Oriented Nature: RMI is designed to work seamlessly with Java's object-oriented features, allowing developers to invoke methods on remote objects as if they were local.

Simplicity and Productivity: RMI abstracts much of the complexity associated with network communication, making it relatively easy to use compared to lower-level network programming.

Integration with Java: RMI is tightly integrated with the Java programming language, allowing developers to use familiar constructs and tools.

Dynamic Class Loading: RMI supports dynamic class loading, allowing objects to be downloaded and executed on the client side, which can be useful for updating applications without restarting them.

Security: RMI provides security features such as authentication and encryption, ensuring that communication between objects is secure.

Disadvantages of RMI:

Java-Centric: RMI is specific to Java, which limits its interoperability with systems implemented in other languages.

Complex Setup: Setting up RMI can be complex, especially for beginners. Configuring the RMI registry, dealing with class loading issues, and understanding the distributed object model can be challenging.

Performance Overhead: RMI may introduce performance overhead due to network communication, serialization/deserialization of objects, and other related processes.

Firewall Issues: RMI often requires special firewall configurations to allow communication between distributed components, which can be a security concern.

Limited Platform Support: Although RMI is designed for Java, it may not be supported on all platforms, limiting its usability in heterogeneous environments.

Types of RMI:

Simple RMI: In this basic form of RMI, communication between client and server involves invoking methods on remote objects.

	Parameter Passing in RMI: Call by Value: Parameters are passed by value, meaning a copy of the parameter is sent to the remote object. Call by Reference: RMI also supports passing parameters by reference, allowing the remote object to directly access and modify the parameter. **RMI with Callbacks (Two-Way RMI)**: This involves the client invoking methods on the server and vice versa. Callbacks enable bidirectional communication between client and server. **Activatable RMI**: Activatable objects are server-side objects that are activated (instantiated) on-demand. They are only loaded into memory when needed, which can be useful for managing resources efficiently.
Algorithm :	**Creating and Running an RMI Application** The following steps are required to create and run an RMI application: 1. Create the remote interface 2. Provide the implementation class for the remote interface 3. Compile the implementation class and create the stub and skeleton objects using the rmic tool 4. Start the registry service by rmiregistry tool 5. Start the server 6. Create and start the client application

Program

1. RMI_Chat_Interface.java -

```
import java.rmi.Remote;

import java.rmi.RemoteException;

public interface RMI_Chat_Interface extends Remote {

    public void sendToServer(String message) throws RemoteException;

}
```

2. RMI_Server.java -

```
import java.rmi.RemoteException;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

import java.rmi.server.UnicastRemoteObject;


public class RMI_Server extends UnicastRemoteObject implements
RMI_Chat_Interface {
```

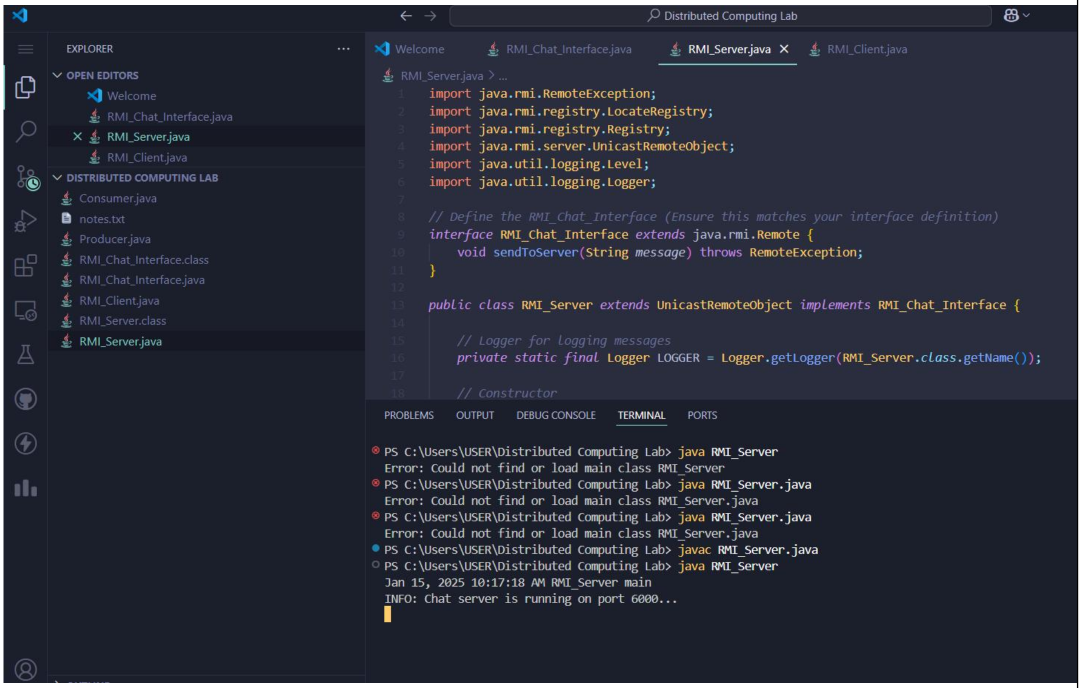
	```
	    public RMI_Server() throws RemoteException
	        { super();
	    }
	    @Override
	    public void sendToServer(String message) throws RemoteException
	        { System.out.println("Client says: " + message);
	    }
	    public static void main(String[] args) throws Exception
	        { Registry rmiregistry = LocateRegistry.createRegistry(6000);
	        rmiregistry.bind("chat", new RMI_Server());
	        System.out.println("Chat server is running...");
	    }
	}
	```

	3. RMI_Client.java -
	```
	import java.rmi.Naming;
	import java.util.Scanner;
	```

```
public class RMI_Client {


    static Scanner input = null;


    public static void main(String[] args) throws Exception
        { RMI_Chat_Interface chatapi = (RMI_Chat_Interface)


        Naming.lookup("rmi://localhost:6000/chat");
        input = new Scanner(System.in);
        System.out.println("Connected to server...");


        System.out.println("Type a message for sending to server...");
        String message = input.nextLine();
        while (!message.equals("Bye"))
            { chatapi.sendToServer(message
            ); message = input.nextLine();
        }
    }
}
```

Output	1 RMI_Server.java

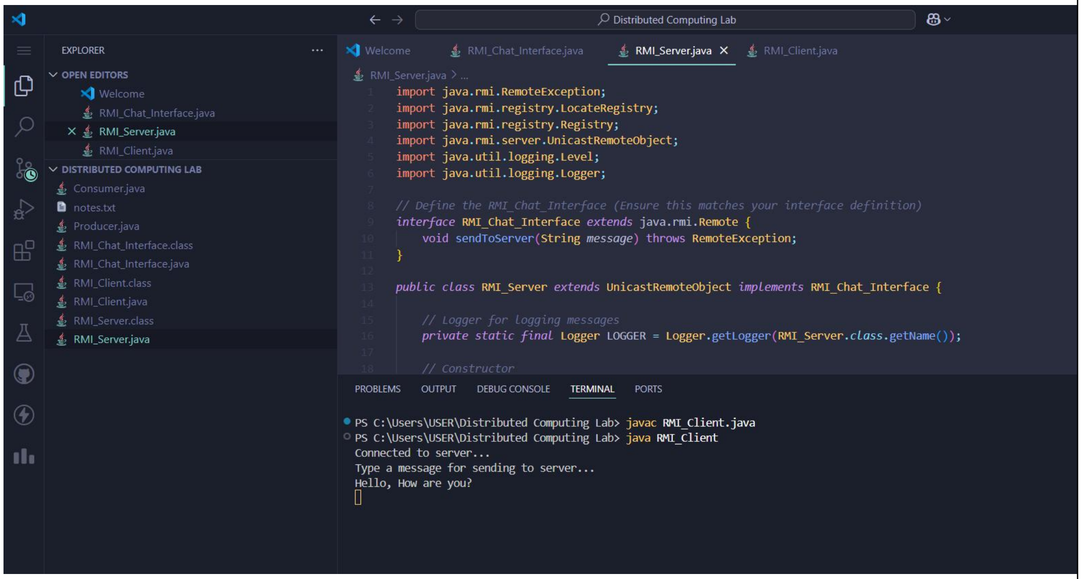
Server Start -



```java
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Level;
import java.util.logging.Logger;

// Define the RMI_Chat_Interface (Ensure this matches your interface definition)
interface RMI_Chat_Interface extends java.rmi.Remote {
    void sendToServer(String message) throws RemoteException;
}

public class RMI_Server extends UnicastRemoteObject implements RMI_Chat_Interface {

    // Logger for logging messages
    private static final Logger LOGGER = Logger.getLogger(RMI_Server.class.getName());

    // Constructor
```

```
PS C:\Users\USER\Distributed Computing Lab> java RMI_Server
Error: Could not find or load main class RMI_Server
PS C:\Users\USER\Distributed Computing Lab> java RMI_Server.java
Error: Could not find or load main class RMI_Server.java
PS C:\Users\USER\Distributed Computing Lab> java RMI_Server.java
Error: Could not find or load main class RMI_Server.java
PS C:\Users\USER\Distributed Computing Lab> javac RMI_Server.java
PS C:\Users\USER\Distributed Computing Lab> java RMI_Server
Jan 15, 2025 10:17:18 AM RMI_Server main
INFO: Chat server is running on port 6000...
```

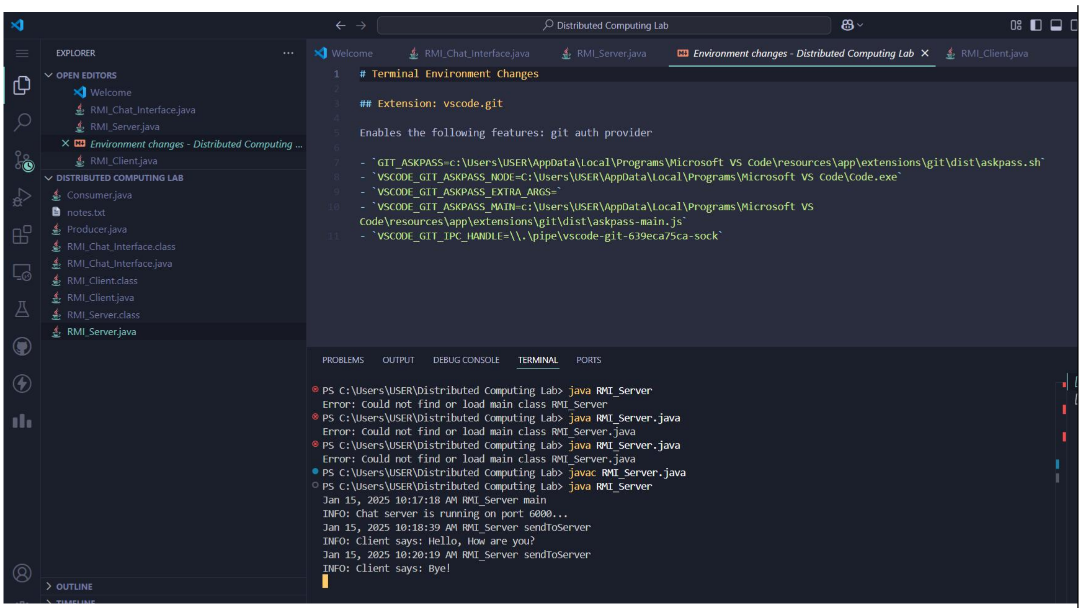
Client Message Request -



```java
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Level;
import java.util.logging.Logger;

// Define the RMI_Chat_Interface (Ensure this matches your interface definition)
interface RMI_Chat_Interface extends java.rmi.Remote {
    void sendToServer(String message) throws RemoteException;
}

public class RMI_Server extends UnicastRemoteObject implements RMI_Chat_Interface {

    // Logger for logging messages
    private static final Logger LOGGER = Logger.getLogger(RMI_Server.class.getName());

    // Constructor
```

```
PS C:\Users\USER\Distributed Computing Lab> javac RMI_Client.java
PS C:\Users\USER\Distributed Computing Lab> java RMI_Client
Connected to server...
Type a message for sending to server...
Hello, How are you?
```

Server Stop Request – Message (Bye)

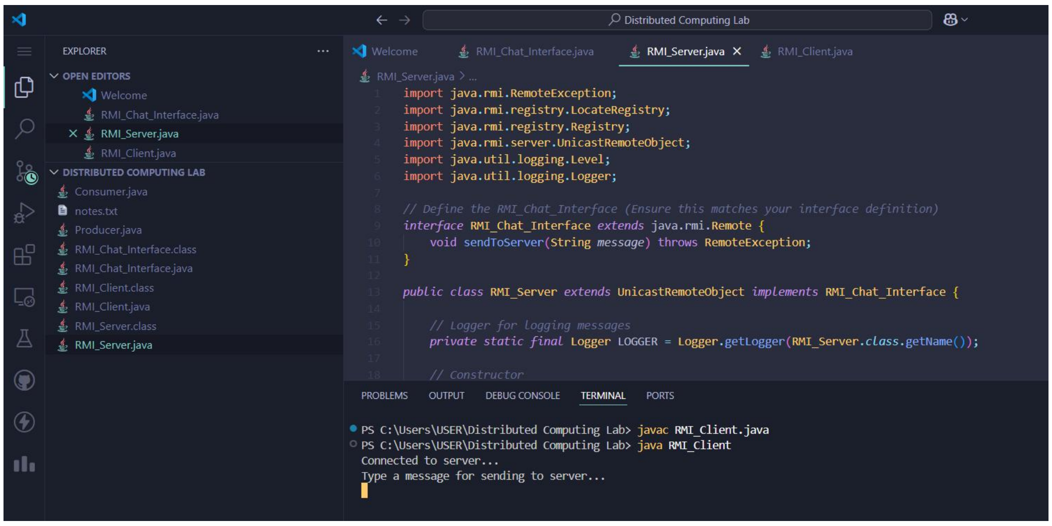


```
# Terminal Environment Changes

## Extension: vscode.git

Enables the following features: git auth provider

- `GIT_ASKPASS=c:\Users\USER\AppData\Local\Programs\Microsoft VS Code\resources\app\extensions\git\dist\askpass.sh`
- `VSCODE_GIT_ASKPASS_NODE=C:\Users\USER\AppData\Local\Programs\Microsoft VS Code\Code.exe`
- `VSCODE_GIT_ASKPASS_EXTRA_ARGS=`
- `VSCODE_GIT_ASKPASS_MAIN=c:\Users\USER\AppData\Local\Programs\Microsoft VS
Code\resources\app\extensions\git\dist\askpass-main.js`
- `VSCODE_GIT_IPC_HANDLE=\\.\pipe\vscode-git-639eca75ca-sock`
```

```
PS C:\Users\USER\Distributed Computing Lab> java RMI_Server
Error: Could not find or load main class RMI_Server
PS C:\Users\USER\Distributed Computing Lab> java RMI_Server.java
Error: Could not find or load main class RMI_Server.java
PS C:\Users\USER\Distributed Computing Lab> java RMI_Server.java
Error: Could not find or load main class RMI_Server.java
PS C:\Users\USER\Distributed Computing Lab> javac RMI_Server.java
PS C:\Users\USER\Distributed Computing Lab> java RMI_Server
Jan 15, 2025 10:17:18 AM RMI_Server main
INFO: Chat server is running on port 6000...
Jan 15, 2025 10:18:39 AM RMI_Server sendToServer
INFO: Client says: Hello, How are you?
Jan 15, 2025 10:20:19 AM RMI_Server sendToServer
INFO: Client says: Bye!
```

2. RMI_Client.java -

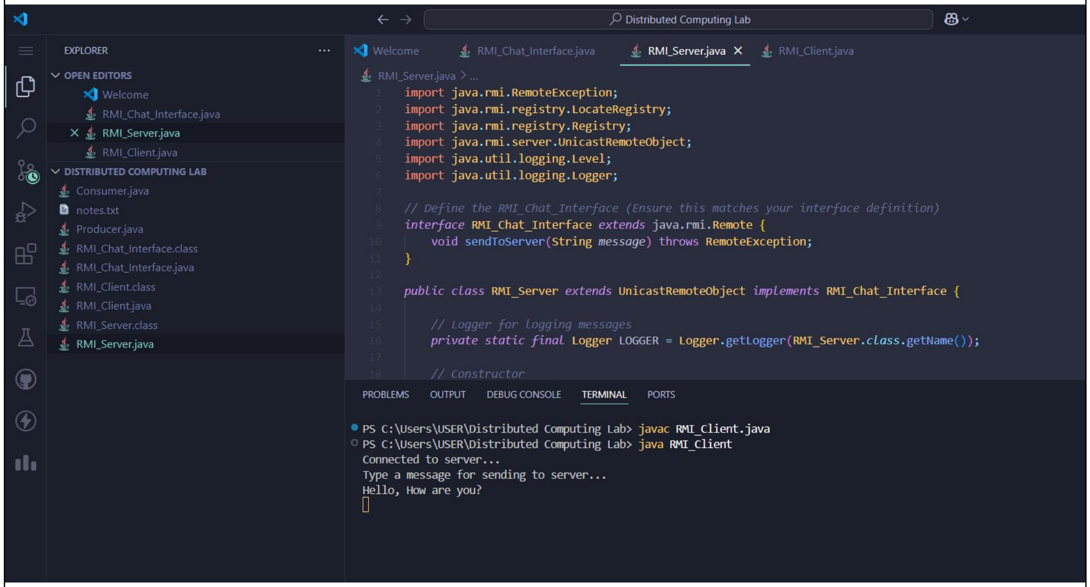
Client Start -



```java
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Level;
import java.util.logging.Logger;

// Define the RMI_Chat_Interface (Ensure this matches your interface definition)
interface RMI_Chat_Interface extends java.rmi.Remote {
    void sendToServer(String message) throws RemoteException;
}

public class RMI_Server extends UnicastRemoteObject implements RMI_Chat_Interface {

    // Logger for logging messages
    private static final Logger LOGGER = Logger.getLogger(RMI_Server.class.getName());

    // Constructor
```

```
PS C:\Users\USER\Distributed Computing Lab> javac RMI_Client.java
PS C:\Users\USER\Distributed Computing Lab> java RMI_Client
Connected to server...
Type a message for sending to server...
```

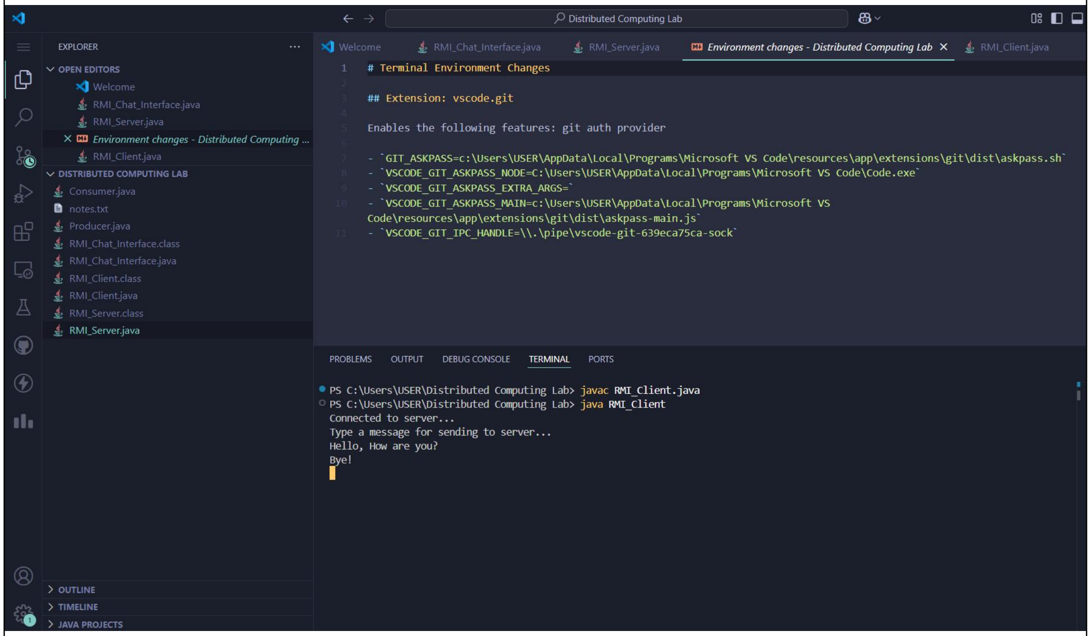
Client Message Request -



```java
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Level;
import java.util.logging.Logger;

// Define the RMI_Chat_Interface (Ensure this matches your interface definition)
interface RMI_Chat_Interface extends java.rmi.Remote {
    void sendToServer(String message) throws RemoteException;
}

public class RMI_Server extends UnicastRemoteObject implements RMI_Chat_Interface {

    // Logger for logging messages
    private static final Logger LOGGER = Logger.getLogger(RMI_Server.class.getName());

    // Constructor
```

```
PS C:\Users\USER\Distributed Computing Lab> javac RMI_Client.java
PS C:\Users\USER\Distributed Computing Lab> java RMI_Client
Connected to server...
Type a message for sending to server...
Hello, How are you?
```

Client Bye Message:



```
# Terminal Environment Changes

## Extension: vscode.git

Enables the following features: git auth provider

- `GIT_ASKPASS=c:\Users\USER\AppData\Local\Programs\Microsoft VS Code\resources\app\extensions\git\dist\askpass.sh`
- `VSCODE_GIT_ASKPASS_NODE=C:\Users\USER\AppData\Local\Programs\Microsoft VS Code\Code.exe`
- `VSCODE_GIT_ASKPASS_EXTRA_ARGS=`
- `VSCODE_GIT_ASKPASS_MAIN=c:\Users\USER\AppData\Local\Programs\Microsoft VS
Code\resources\app\extensions\git\dist\askpass-main.js`
- `VSCODE_GIT_IPC_HANDLE=\\.\pipe\vscode-git-639eca75ca-sock`
```

```
PS C:\Users\USER\Distributed Computing Lab> javac RMI_Client.java
PS C:\Users\USER\Distributed Computing Lab> java RMI_Client
Connected to server...
Type a message for sending to server...
Hello, How are you?
Bye!
```

Conclusion : Thus, we have studied basic underlying concepts of forming distributed systems and performed Implementation on Client/ Server Application with the help of Remote Method Invocation (RMI)

References:	https://en.wikipedia.org/wiki/Java_remote_method_invocation https://www.oracle.com/java/technologies/jpl1-remote-method-invocation.html http://infolab.stanford.edu/CHAIMS/Doc/Details/Protocols/rmi/rmi_description.html

Rubrics for Assessment

Timely Submission	Submitted after 2 weeks 0	Submitted after deadline 1	On time Submission 2
Understanding	Student is confused about the concept 0	Students has justifiably understood the concept 2	Students is very clear about the concepts 3
Performance	Students has not performed the Experiment 0	Student has performed with help 2	Student has independently performed the experiment 3
Development	Student struggles to write code 0	Student can write code the requirement stated 1	Student can write exceptional code with his own ideas 2

Distributed Computing Experiment – 3

Name: Gaven Dcosta

Roll No: 09

Batch: A

Program to demonstrate Group Communication	
Learning Objective:	To understand basic underlying concepts of forming distributed systems.
Learning Outcome :	Students will gain ability to understand basic underlying concepts of forming distributed systems.
Course Outcome :	**CSL801.1**
Program Outcome :	**(PO1) Engineering Knowledge:** The program demonstrates the application of distributed system concepts, leveraging knowledge of synchronization algorithms to address time consistency challenges across multiple nodes. **(PO2) Problem Analysis:** The algorithm effectively identifies the challenge of time discrepancies in distributed systems and analyzes the problem by formulating a solution based on the master-slave synchronization model. **(PO3) Design/Development of Solutions:** The system is designed to synchronize the clocks of multiple slave nodes with the master node. It handles concurrent communication and time adjustment, ensuring accurate synchronization across the system.

Theory:	Group communication is a key concept in computer networks where messages are sent to a group of recipients. It typically involves three types of communication methods: unicast, broadcast, and multicast. Each of these communication models serves different purposes and is used based on the specific requirements of the communication. **1. Unicast** • **Definition**: Unicast refers to one-to-one communication between a sender and a single recipient. In unicast communication, a message is sent from one sender to one receiver over a network. • **Example**: When you send an email to one specific person, it’s a unicast communication. • **Key Characteristics**: o One-to-one communication. o Every message sent requires a separate communication link. o Can be inefficient when the sender needs to send the same message to many recipients, as it requires multiple copies of the same message. **2. Broadcast** • **Definition**: Broadcast is one-to-all communication. It involves sending a message to all devices within a network segment or a broader network. • **Example**: A television or radio signal broadcast to everyone within range. • **Key Characteristics**: o One-to-all communication, where every device within a particular broadcast domain (or network) receives the message. o Typically used in technologies like Ethernet (in local area networks). o Inefficient in larger networks as all devices must process the broadcasted message, even if the message is not relevant to them. o **Scope**: Limited to a network or subnet; routers typically don't forward broadcast messages. **3. Multicast** • **Definition**: Multicast is a one-to-many or many-to-many communication

method, where a sender sends a message to a group of selected recipients.

- **Example**: Streaming video or audio to a group of viewers.
- **Key Characteristics**:
    - One-to-many communication, but unlike broadcast, only specific devices that have expressed interest in the message (members of a multicast group) will receive it.
    - Efficient because it avoids sending the message to every device (as with broadcast).
    - Multicast uses specific IP address ranges (e.g., IPv4 multicast addresses 224.0.0.0 to 239.255.255.255).
    - Requires specific network protocols and infrastructure to manage group memberships, like Internet Group Management Protocol (IGMP) for IPv4.
    - **Scope**: Can be constrained to a particular group of devices or users, but it can be routed across networks, unlike broadcast.

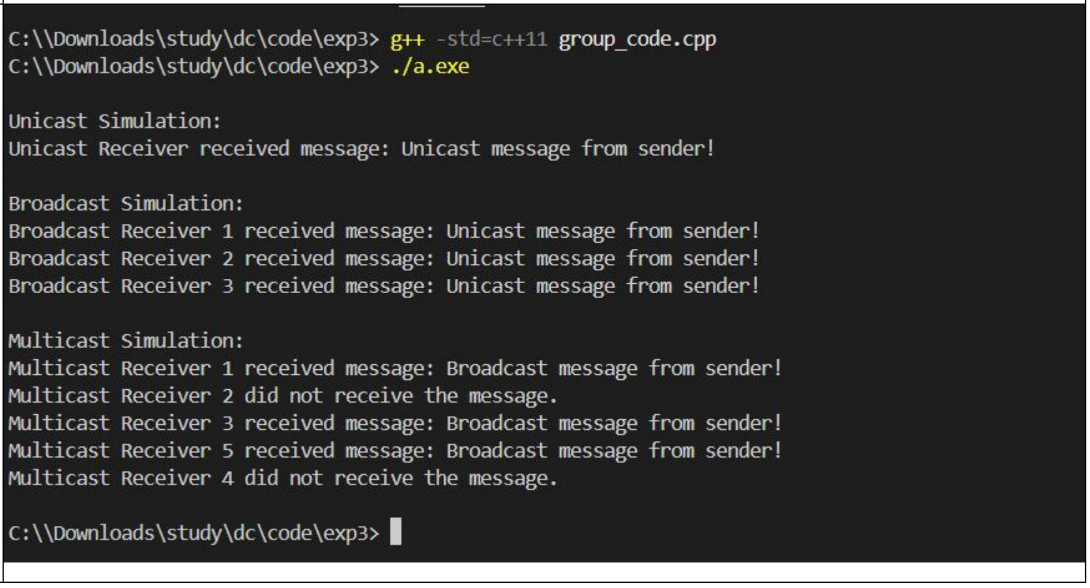
Program :

```
#include <iostream>
#include <thread>
#include <vector>
#include <condition_variable>
#include <mutex>

std::mutex mtx;
std::condition_variable cv;
bool data_ready = false;
std::string received_message;

void unicast_receiver_thread() {
    std::unique_lock<std::mutex> lock(mtx);
    while (!data_ready) {
        cv.wait(lock); // Wait for the signal from the sender thread
    }
     std::cout << "Unicast Receiver received message: " << received_message <<
```

```
std::endl;
}

void unicast_sender_thread() {
    std::this_thread::sleep_for(std::chrono::seconds(1)); // Simulate delay
    std::lock_guard<std::mutex> lock(mtx);
    received_message = "Unicast message from sender!";
    data_ready = true;
    cv.notify_all(); // Notify the receiver thread that data is ready
}

void broadcast_receiver_thread(int id) {
    std::unique_lock<std::mutex> lock(mtx);
    while (!data_ready) {
        cv.wait(lock); // Wait for the signal from the sender thread
    }
    std::cout << "Broadcast Receiver " << id << " received message: " <<
received_message << std::endl;
}

void broadcast_sender_thread() {
    std::this_thread::sleep_for(std::chrono::seconds(1)); // Simulate delay
    std::lock_guard<std::mutex> lock(mtx);
    received_message = "Broadcast message from sender!";
    data_ready = true;
    cv.notify_all(); // Notify all receiver threads
}

void multicast_receiver_thread(int id, bool is_subscriber) {
    std::unique_lock<std::mutex> lock(mtx);
    while (!data_ready) {
        cv.wait(lock); // Wait for the signal from the sender thread
    }
    if (is_subscriber) {
            std::cout << "Multicast Receiver " << id << " received message: " <<
received_message << std::endl;
    } else {
        std::cout << "Multicast Receiver " << id << " did not receive the message."
<< std::endl;
    }
}

void multicast_sender_thread() {
    std::this_thread::sleep_for(std::chrono::seconds(1)); // Simulate delay
    std::lock_guard<std::mutex> lock(mtx);
    received_message = "Multicast message from sender!";
    data_ready = true;
    cv.notify_all(); // Notify all receiver threads
}

int main() {
    // --- Unicast ---
    std::cout << std::endl << "Unicast Simulation:\n";
    std::thread unicast_receiver(unicast_receiver_thread);
    std::thread unicast_sender(unicast_sender_thread);
    unicast_sender.join();
    unicast_receiver.join();

    // --- Broadcast ---
    std::cout << "\nBroadcast Simulation:\n";
    const int num_broadcast_receivers = 3;
    std::vector<std::thread> broadcast_receivers;
    for (int i = 0; i < num_broadcast_receivers; ++i) {
            broadcast_receivers.push_back(std::thread(broadcast_receiver_thread,
i+1));
    }
    std::thread broadcast_sender(broadcast_sender_thread);
    broadcast_sender.join();
```

```
  for (auto& receiver : broadcast_receivers) {
    receiver.join();
  }

  // --- Multicast ---
  std::cout << "\nMulticast Simulation:\n";
  const int num_multicast_receivers = 5;
  std::vector<std::thread> multicast_receivers;
   std::vector<bool> is_subscriber = {true, false, true, false, true}; // Simulating
who will receive
  for (int i = 0; i < num_multicast_receivers; ++i) {
      multicast_receivers.push_back(std::thread(multicast_receiver_thread, i+1,
is_subscriber[i]));
  }
  std::thread multicast_sender(multicast_sender_thread);
  multicast_sender.join();
  for (auto& receiver : multicast_receivers) {
    receiver.join();
  }
std::cout << std::endl;
  return 0;
}
```

Output



```
C:\\Downloads\study\dc\code\exp3> g++ -std=c++11 group_code.cpp
C:\\Downloads\study\dc\code\exp3> ./a.exe

Unicast Simulation:
Unicast Receiver received message: Unicast message from sender!

Broadcast Simulation:
Broadcast Receiver 1 received message: Unicast message from sender!
Broadcast Receiver 2 received message: Unicast message from sender!
Broadcast Receiver 3 received message: Unicast message from sender!

Multicast Simulation:
Multicast Receiver 1 received message: Broadcast message from sender!
Multicast Receiver 2 did not receive the message.
Multicast Receiver 3 received message: Broadcast message from sender!
Multicast Receiver 5 received message: Broadcast message from sender!
Multicast Receiver 4 did not receive the message.

C:\\Downloads\study\dc\code\exp3> 
```

Conclusion :	In this experiment, we implemented and compared unicast, broadcast, and multicast communication methods. Unicast was effective for one-to-one communication but inefficient with multiple recipients. Broadcast, while useful for network-wide messages, showed inefficiency in larger networks due to unnecessary overhead. Multicast excelled in scalability and efficiency by sending data to specific groups, reducing network congestion. Overall, the choice of communication method depends on the scenario—unicast for direct communication, broadcast for subnet-wide messages, and multicast for scalable, one-to-many communication.
References :	https://www.geeksforgeeks.org/group-communication-in-distributed-systems/ https://en.wikipedia.org/wiki/Distributed_computing elearn moodle

Rubrics for Assessment

Timely Submission	Submitted after 2 weeks 0	Submitted after deadline 1	On time Submission 2
Understanding	Student is confused about the concept 0	Students has justifiably understood the concept 2	Students is very clear about the concepts 3
Performance	Students has not performed the Experiment 0	Student has performed with help 2	Student has independently performed the experiment 3
Development	Student struggles to write code 0	Student can write code the requirement stated 1	Student can write exceptional code with his own ideas 2

Distributed Computing Experiment – 4

Name: Gaven Dcosta

Roll No: 09

Batch: A

Program to demonstrate Clock Synchronization algorithms	
Learning Objective:	To understand basic underlying concepts of forming distributed systems.
Learning Outcome :	Students will gain ability to learn the concept of clock Synchronization.
Course Outcome :	**CSL801.2**
Program Outcome :	**(PO1) Engineering Knowledge:** Learning Outcome: Apply mathematical and engineering principles to understand and implement clock synchronization algorithms, addressing real-world challenges in distributed systems. **(PO2) Problem Analysis:** Learning Outcome: Analyze and formulate solutions for complex clock synchronization problems in distributed systems using engineering sciences, considering the impact of time discrepancies. **(PO3) Design/Development of Solutions:** Learning Outcome: Design and implement clock synchronization solutions that meet system requirements while considering safety, efficiency, and societal needs in real-world distributed applications.

Theory:	Clock synchronization is crucial in distributed systems to ensure that all nodes (or processes) in a system agree on the time, or at least are within an acceptable margin of difference. This is especially important for coordinating activities, logging events in a consistent order, and preventing issues caused by time inconsistencies (e.g., in distributed databases or real-time systems). In distributed systems, each node generally has its own local clock, which can drift over time due to various factors (e.g., hardware differences, environmental conditions). The goal of clock synchronization is to adjust the clocks of these nodes so that they stay in sync within a tolerable range. The two main types of clock synchronization are: 1. **Physical Clocks**: These are real-world clocks, such as those driven by hardware or real-time systems. 2. **Logical Clocks**: These don't represent real-world time but are used to order events and provide a "logical" ordering in distributed systems. The most commonly used algorithms for **clock synchronization** in distributed systems are: 1. **Berlekamp's Algorithm** (or **Berlekamp's Algorithm for Clock Synchronization**) 2. **Network Time Protocol (NTP)** 3. **Lamport’s Logical Clocks** 4. **Cristian's Algorithm** **Berlekamp's Algorithm for Clock Synchronization (Using Master-Slave Model)** **Overview:** • The **Master-Slave Model** assumes one node (the "master") that requests time from several other nodes (the "slaves"). • The master node computes an average time and sends adjustment messages to the slave nodes to bring their clocks closer to this average. • This approach uses a **feedback mechanism** where slaves adjust their clocks based on the master's time. **Key Components:** 1. **Master node**: Requests time from all slaves, computes the average time,

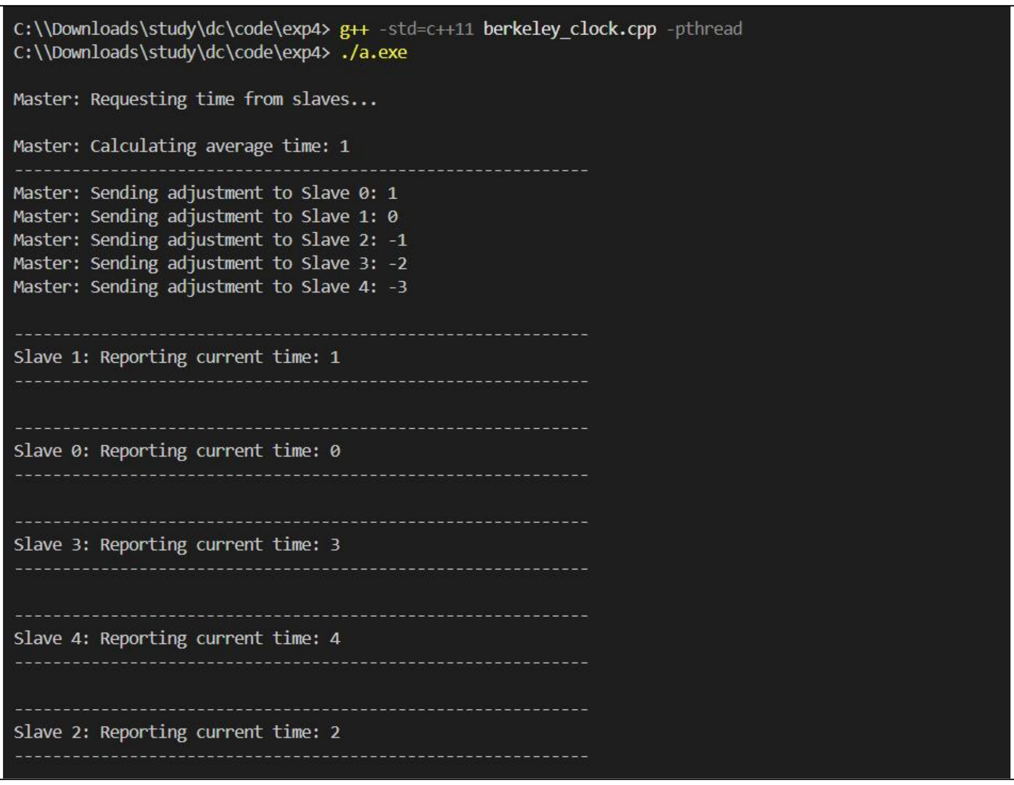
	and sends clock adjustments. 2. **Slave nodes**: Respond to the master's requests with their current time, and adjust their clocks based on the master's computed average.
Algorithm :	1. **Initialize:** o Let T_m be the current time at the master node. o Let T_s[i] be the current time at slave node i (initially set to some value). 2. **Master Requests Time:** o Master sends a time request to all slaves. 3. **Slaves Respond with Time:** o Each slave node i sends its current time T_s[i] to the master. 4. **Master Computes Average Time:** o The master computes the average time: Average Time=Tm+∑i=1nTs[i]n+1\text{Average Time} = \frac{T_m + \sum_{i=1}^{n} T_s[i]}{n + 1} 5. **Master Sends Adjustment to Slaves:** o For each slave node i, the master sends an adjustment: Adjustment[i]=Average Time−Ts[i]\text{Adjustment}[i] = \text{Average Time} - T_s[i] o The slave node adjusts its local clock by this amount. 6. **Slaves Adjust Time:** o Each slave adjusts its time: Ts[i]=Ts[i]+Adjustment[i]T_s[i] = T_s[i] + \text{Adjustment}[i] 7. **Slave Sends Final Time to Master:** o After adjustment, each slave sends the final time back to the master for verification. 8. **Repeat:** o The synchronization process repeats after a specific interval to ensure continued clock accuracy.

Program :

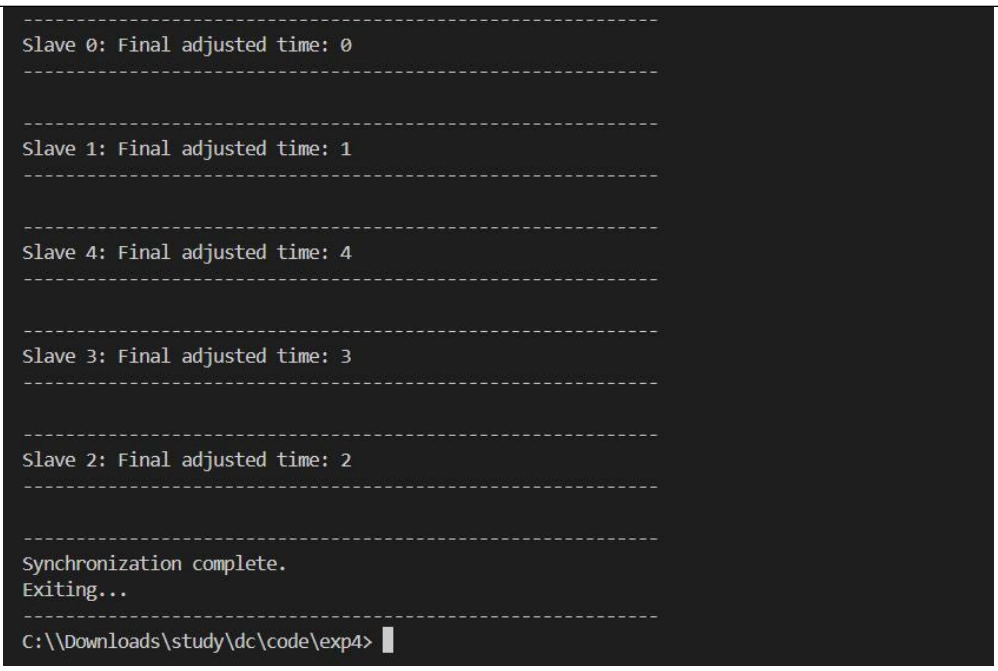
```
#include <iostream>
#include <vector>
#include <thread>
#include <chrono>
#include <mutex>
#include <random>
#include <atomic>

std::mutex mtx;  // Mutex to protect shared resources
std::atomic<int> master_time(0);  // Time of the master node, shared among threads
std::vector<int> slave_times;  // Vector to store slave times
std::atomic<bool> done(false);  // Atomic flag to signal end of synchronization

// Function for the master node to request time from slaves
void master_node(int num_slaves) {
  while (!done) {
    // Clear the slave times from previous synchronization round
    slave_times.clear();

    // Request time from all slave nodes
    std::cout << "\nMaster: Requesting time from slaves...\n";
    for (int i = 0; i < num_slaves; ++i) {
      slave_times.push_back(i);  // Store the slave node's index (using it as time here)
    }

    // Simulate receiving time from each slave
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

    // Calculate the average time from the received times
      int total_time = master_time.load();  // Include master time in the average calculation
```

```
        for (int time : slave_times) {
            total_time += time;  // Using the slave index as a stand-in for the clock time
        }

        int average_time = total_time / (num_slaves + 1);  // Include master node

        std::cout << "\nMaster: Calculating average time: " << average_time << "\n";
        std::cout << "---------------------------------------------------------\n";

        // Send clock adjustment to each slave based on the average time
        for (int i = 0; i < num_slaves; ++i) {
            int slave_time_offset = average_time - slave_times[i];
            std::cout << "Master: Sending adjustment to Slave " << i << ": " << slave_time_offset << "\n";

            // Update slave time after adjustment
            slave_times[i] += slave_time_offset;
        }

        std::this_thread::sleep_for(std::chrono::seconds(2));  // Wait before the next sync round
    }
}

// Function for each slave node to report its time to the master
void slave_node(int id) {
    int slave_time = id;  // Initial time of slave (simulated by the node ID)

    while (!done) {
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));  // Simulate doing work
        // Print current slave time (initially set to id, but will be adjusted)
        {
            std::lock_guard<std::mutex> lock(mtx);
            std::cout << "\n------------------------------------------------------------\n";
                    std::cout << "Slave " << id << ": Reporting current time: " <<
slave_time << "\n";
            std::cout << "------------------------------------------------------------\n";
        }

        // Print final adjusted time after receiving adjustment from master
         std::this_thread::sleep_for(std::chrono::milliseconds(500));  // Simulate
time for adjustment
        {
            std::lock_guard<std::mutex> lock(mtx);
            std::cout << "\n------------------------------------------------------------\n";
            std::cout << "Slave " << id << ": Final adjusted time: " << slave_time <<
"\n";
            std::cout << "------------------------------------------------------------\n";
        }
    }
}

// Main function to start the simulation
int main() {
    int num_slaves = 5;  // Number of slave nodes

    // Start slave nodes
    std::vector<std::thread> slaves;
    for (int i = 0; i < num_slaves; ++i) {
        slaves.push_back(std::thread(slave_node, i));
    }

    // Start master node
    std::thread master(master_node, num_slaves);

    // Run the simulation for a certain time
    std::this_thread::sleep_for(std::chrono::seconds(1));

    // Stop all threads
    done = true;
    master.join();
    for (auto& slave : slaves) {
        slave.join();
    }

    std::cout << "\n------------------------------------------------------------\n";
    std::cout << "Synchronization complete.\nExiting...\n";
    std::cout << "------------------------------------------------------------\n";

    return 0;
}
```

Output



```
C:\\Downloads\study\dc\code\exp4> g++ -std=c++11 berkeley_clock.cpp -pthread
C:\\Downloads\study\dc\code\exp4> ./a.exe

Master: Requesting time from slaves...

Master: Calculating average time: 1
------------------------------------------------------------
Master: Sending adjustment to Slave 0: 1
Master: Sending adjustment to Slave 1: 0
Master: Sending adjustment to Slave 2: -1
Master: Sending adjustment to Slave 3: -2
Master: Sending adjustment to Slave 4: -3

------------------------------------------------------------
Slave 1: Reporting current time: 1
------------------------------------------------------------

------------------------------------------------------------
Slave 0: Reporting current time: 0
------------------------------------------------------------

------------------------------------------------------------
Slave 3: Reporting current time: 3
------------------------------------------------------------

------------------------------------------------------------
Slave 4: Reporting current time: 4
------------------------------------------------------------

------------------------------------------------------------
Slave 2: Reporting current time: 2
------------------------------------------------------------
```



```
------------------------------------------------------------
Slave 0: Final adjusted time: 0
------------------------------------------------------------

------------------------------------------------------------
Slave 1: Final adjusted time: 1
------------------------------------------------------------

------------------------------------------------------------
Slave 4: Final adjusted time: 4
------------------------------------------------------------

------------------------------------------------------------
Slave 3: Final adjusted time: 3
------------------------------------------------------------

------------------------------------------------------------
Slave 2: Final adjusted time: 2
------------------------------------------------------------

------------------------------------------------------------
Synchronization complete.
Exiting...
------------------------------------------------------------
C:\\Downloads\study\dc\code\exp4>
```

Conclusion :	The master-slave clock synchronization model ensures that all slave nodes align their clocks with the master node. The master periodically requests time from slaves, calculates the average time, and sends adjustments to each slave. After adjusting their clocks, slaves send the final time back for verification. This process ensures accurate and consistent time across all nodes, which is crucial for distributed systems. The synchronization can be repeated to maintain time accuracy, minimizing errors caused by clock drift or discrepancies. This approach is effective for maintaining synchronization in large, distributed systems.
References :	https://en.wikipedia.org/wiki/Clock_synchronization https://en.wikipedia.org/wiki/Berkeley_algorithm https://www.geeksforgeeks.org/clock-synchronization-in-distributed-system/

Rubrics for Assessment

Timely Submission	Submitted after 2 weeks 0	Submitted after deadline 1	On time Submission 2
Understanding	Student is confused about the concept 0	Students has justifiably understood the concept 2	Students is very clear about the concepts 3
Performance	Students has not performed the Experiment 0	Student has performed with help 2	Student has independently performed the experiment 3
Development	Student struggles to write code 0	Student can write code the requirement stated 1	Student can write exceptional code with his own ideas 2

Distributed Computing

Experiment - 5

Name: Gaven Dcosta

Roll No: 09

Batch: A

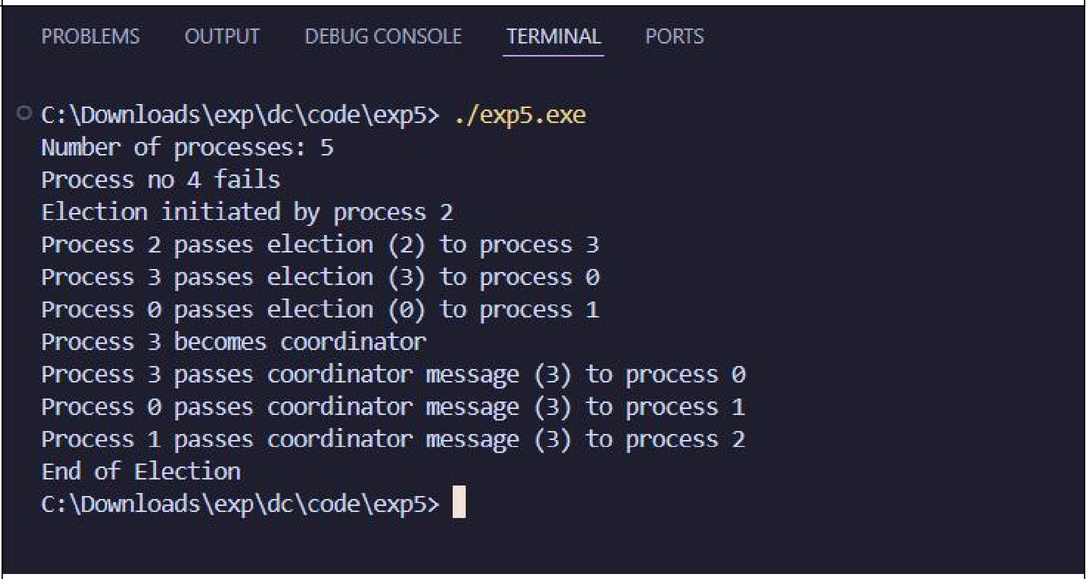
Election Algorithm	
Learning Objective:	To understand basic underlying concepts of forming distributed systems.
Learning Outcome:	Students will gain ability to Implement techniques for Election Algorithms.
Course Outcome:	**CSL801.2**
Program Outcome:	**(PO3) To learn Election Algorithm.:** Learning Outcome: Implement techniques for Election Algorithms.
Theory:	This program simulates the Bully Election Algorithm in a distributed system to select a coordinator after a failure of one of the processes. The Bully Algorithm is a well-known algorithm used in distributed systems to elect a leader or coordinator. The program aims to show how processes in a system can conduct an election when one of the processes fails. • **Processes**: In a distributed system, each process has an identifier (ID) and can either be active or inactive (failed). Active processes participate in the election, while inactive (failed) processes are excluded from the election process. • **Election**: The election starts when a process detects that the coordinator has failed or needs to be replaced. In this case, process 2 starts the election. Processes with higher IDs participate in the election by passing messages, and the process with the highest ID is elected as the new coordinator. • **Coordinator**: The coordinator is the process that takes charge of certain tasks and manages the system until a new election is triggered.

Algorithm :	• **Initialization**: Set the number of processes totalProcesses = 5. Create 5 processes (with IDs 0, 1, 2, 3, 4), all marked as active (act = true). • **Simulate Process Failure**: Randomly select a process to fail. Mark the failed process as inactive (act = false). • **Initiate Election**: Process 2 initiates the election. Process 2 sends the election message to the next process. If the next process has a higher ID and is still active, it takes over the election. • **Election Process**: The election message is passed in a cyclic manner to each process. The process with the highest ID among active processes is eventually elected as the coordinator. • **Coordinator Selection**: The process with the highest active ID becomes the coordinator. The coordinator sends a message to all processes to inform them of its new role. • **End of Election**: The election ends when all processes know the coordinator.

Program :

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Process {
    int id;
    bool active;
    Process(int id) {
        this->id = id;
        active = true;
    }
};

class ElectionSystem {
public:
    int totalProcesses;
    std::vector<Process> processes;

    ElectionSystem() {}

    void initializeSystem() {
        std::cout << "Number of processes: 5" << std::endl;
        totalProcesses = 5;
        processes.reserve(totalProcesses);
        for (int i = 0; i < processes.capacity(); i++) {
            processes.emplace_back(i);
        }
    }

    void initiateElection() {
        std::cout << "Process no " <<
processes[findMaxActiveProcess()].id << " fails" << std::endl;
        processes[findMaxActiveProcess()].active = false;
        std::cout << "Election initiated by process 2" << std::endl;

        int initiator = 2;
        int current = initiator;
        int next = current + 1;

        while (true) {
            if (processes[next].active) {
                std::cout << "Process " << processes[current].id << " passes
election (" << processes[current].id
                        << ") to process " << processes[next].id << std::endl;
                current = next;
            }

            next = (next + 1) % totalProcesses;
            if (next == initiator) {
                break;
            }
        }

        std::cout << "Process " << processes[findMaxActiveProcess()].id
<< " becomes coordinator" << std::endl;
        int coordinator = processes[findMaxActiveProcess()].id;

        current = coordinator;
        next = (current + 1) % totalProcesses;

        while (true) {
            if (processes[next].active) {
                std::cout << "Process " << processes[current].id << " passes
coordinator message (" << coordinator
                        << ") to process " << processes[next].id << std::endl;
                current = next;
            }

            next = (next + 1) % totalProcesses;
            if (next == coordinator) {
                std::cout << "End of Election" << std::endl;
                break;
            }
        }
    }

    int findMaxActiveProcess() {
        int index = 0;
        int maxId = -9999;
        for (int i = 0; i < processes.size(); i++) {
            if (processes[i].active && processes[i].id > maxId) {
                maxId = processes[i].id;
                index = i;
            }
        }
        return index;
    }
};

int main() {
    ElectionSystem system;
    system.initializeSystem();
    system.initiateElection();
    return 0;
}
```

Output



```
C:\Downloads\exp\dc\code\exp5> ./exp5.exe
Number of processes: 5
Process no 4 fails
Election initiated by process 2
Process 2 passes election (2) to process 3
Process 3 passes election (3) to process 0
Process 0 passes election (0) to process 1
Process 3 becomes coordinator
Process 3 passes coordinator message (3) to process 0
Process 0 passes coordinator message (3) to process 1
Process 1 passes coordinator message (3) to process 2
End of Election
C:\Downloads\exp\dc\code\exp5>
```

Conclusion :	This program implements a distributed **Bully Election Algorithm** to simulate the election of a new coordinator after a failure in a system with multiple processes. The election process is initiated by a specific process, and the highest ID among active processes becomes the new coordinator. The program also randomly selects a process to fail, making the simulation more dynamic and realistic for distributed systems.
References :	https://en.wikipedia.org/wiki/Bully_algorithm https://www.geeksforgeeks.org/bully-algorithm-in-distributed-system/

Rubrics for Assessment

Timely Submission	Submitted after 2 weeks 0	Submitted after deadline 1	On time Submission 2
Understanding	Student is confused about the concept 0	Students has justifiably understood the concept 2	Students is very clear about the concepts 3
Performance	Students has not performed the Experiment 0	Student has performed with help 2	Student has independently performed the experiment 3
Development	Student struggles to write code 0	Student can write code the requirement stated 1	Student can write exceptional code with his own ideas 2

Distributed Computing

Experiment - 6

Name: Gaven Dcosta

Roll No: 09

Batch: A

Mutual Exclusion Algorithm	
Learning Objective:	To explore mutual exclusion algorithms and deadlock handling in the distributed system
Learning Outcome:	Students will gain ability to demonstrate mutual exclusion algorithms and deadlock handling.
Course Outcome:	**CSL801.2**
Program Outcome:	**(PO3) To learn Election Algorithm.:** Learning Outcome: Implement techniques for Election Algorithms.
Theory:	This experiment implements **two distributed mutual exclusion algorithms**: **Maekawa's Algorithm** and **Lamport's Algorithm**. Both algorithms are designed to solve the mutual exclusion problem in a distributed system, ensuring that only one process can enter its critical section (CS) at any given time. The mutual exclusion problem arises in distributed systems where multiple processes (nodes) need to access a shared resource (critical section). Since there's no shared memory or global clock, these algorithms use message passing and logical mechanisms to ensure that the critical section is accessed by only one process at a time. **1. Maekawa's Algorithm** Maekawa's Algorithm is a **distributed mutual exclusion algorithm** where each process is assigned a *voting set*. A voting set is a subset of processes that a particular process must contact to request permission to enter the critical section. The key idea is to reduce the total number of message exchanges in comparison to algorithms that request permission from all processes in the system. • **Voting Set**: Each process has a unique voting set that determines

which other processes it needs to ask for permission. Only processes within the voting set can vote on whether the requesting process is allowed to enter the critical section.

- **Requesting the Critical Section**: A process sends a request to all processes in its voting set. The process enters the critical section only when it receives permission from every member of its voting set.
- **Releasing the Critical Section**: After exiting the critical section, the process sends a message to each member of its voting set to release the permission it held.

The algorithm ensures mutual exclusion because at least one process in each voting set will not vote for any other process until it has finished executing in the critical section.

2. Lamport's Algorithm

Lamport's Algorithm is a **logical clock-based mutual exclusion algorithm** used in distributed systems where processes use timestamps to order events. It allows processes to communicate through messages and ensures that the system behaves as though there is a global clock, even though no global clock exists.

- **Logical Clock**: Each process maintains a logical clock that is incremented each time an event occurs (e.g., sending or receiving a message). This clock is used to order requests.
- **Requesting the Critical Section**: A process sends a request message to all other processes, including the timestamp of its request (logical clock value). The process must wait for replies from all other processes to enter the critical section.
- **Releasing the Critical Section**: After exiting the critical section, the process sends a release message to all processes.
- **Logical Consistency**: The key feature of Lamport's algorithm is that it ensures the "happens-before" relationship using timestamps, meaning the order of events is correctly maintained across processes, ensuring mutual exclusion.

This algorithm guarantees mutual exclusion by using logical clocks and the "first-come, first-served" principle: a process with an earlier timestamp is prioritized in entering the critical section.

Algorithm :	Maekawa's algorithm can be described in the following steps: 1. **Initialization**: ○ Each process maintains its unique identifier, a timestamp, and a request flag. ○ Each process is assigned a voting set (a subset of other processes). 2. **Requesting the Critical Section**: ○ A process sends a request to all processes in its voting set. ○ The request includes the process's timestamp. ○ The process increments its timestamp each time it sends a request. 3. **Receiving the Requests and Granting Votes**: ○ When a process receives a request from another process, it checks if the process is eligible to enter the critical section: ▪ If the process is not requesting and is not in its critical section, it grants permission (i.e., gives a vote). ▪ If the process is already requesting, it denies the request. 4. **Entering the Critical Section**: ○ A process enters the critical section if it has received votes from all members of its voting set. ○ If not, the process waits for the votes to be received. 5. **Exiting the Critical Section**: ○ After executing its critical section, the process exits and resets its state. ○ The process sends a message to all processes in its voting set to release the votes. 6. **Termination**: ○ The process completes its execution. --- Lamport's algorithm can be described as follows: 1. **Initialization**: ○ Each process maintains a logical clock (timestamp) initialized to 0. ○ Each process is aware of all other processes in the system. 2. **Requesting the Critical Section**: ○ A process sends a request message to all other processes, including its current timestamp. ○ The request message indicates that the process wants to

enter the critical section.

3. **Receiving the Requests**:
    - When a process receives a request, it compares the timestamps:
        - If the request has a lower timestamp, the process sends a reply granting permission to enter the critical section.
        - If the timestamp is greater or equal, the process delays its reply.
4. **Entering the Critical Section**:
    - A process can enter the critical section when it has received permission from all other processes.
    - The process waits for the replies before entering the critical section.
5. **Releasing the Critical Section**:
    - After exiting the critical section, the process sends a release message to all other processes.
6. **Termination**:
    - The process completes its execution.

Program :

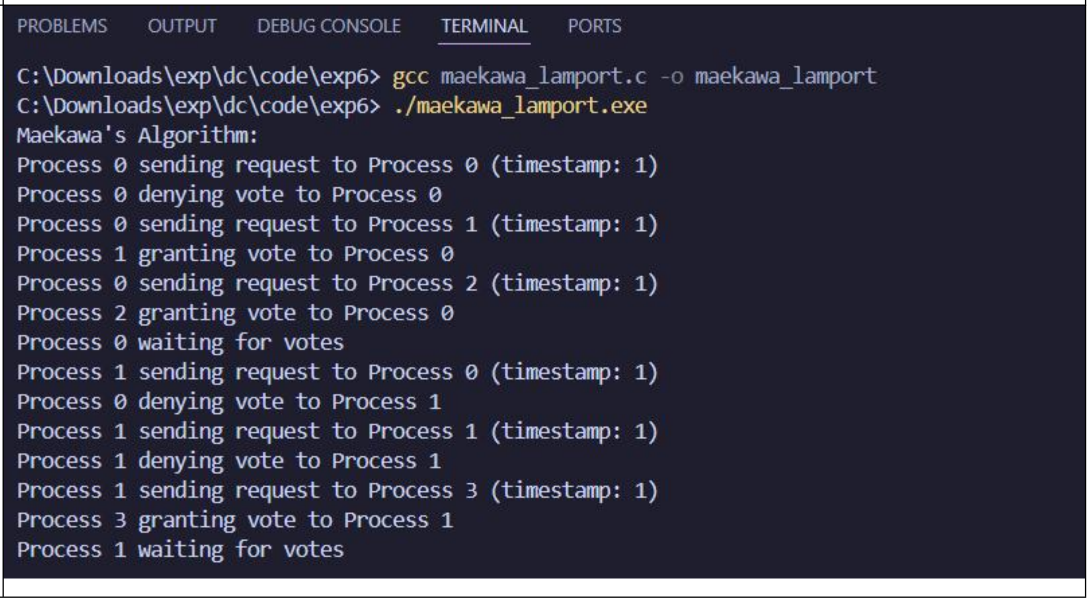
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>

// Maekawa's Algorithm

typedef struct {
    int process_id;
    bool requested;
    bool held;
    int timestamp; // Logical clock
    int vote; // 0: no vote, 1: vote given
} Process;

// Simulate a network of processes
#define NUM_PROCESSES 5
Process processes[NUM_PROCESSES];

// Intersection set for each process (simplified for demonstration)
int intersection_sets[NUM_PROCESSES][NUM_PROCESSES];
```

```
// Function to initialize processes
void initialize_processes() {
   for (int i = 0; i < NUM_PROCESSES; i++) {
      processes[i].process_id = i;
      processes[i].requested = false;
      processes[i].held = false;
      processes[i].timestamp = 0;
      processes[i].vote = 0;
   }

   // Example intersection sets (replace with your actual sets)
   int example_sets[NUM_PROCESSES][NUM_PROCESSES] = {
      {0, 1, 2, -1, -1}, // Process 0's set
      {0, 1, 3, -1, -1}, // Process 1's set
      {0, 2, 4, -1, -1}, // Process 2's set
      {1, 3, 4, -1, -1}, // Process 3's set
      {2, 3, 4, -1, -1}  // Process 4's set
   };

   for(int i = 0; i<NUM_PROCESSES; i++){
      for(int j = 0; j<NUM_PROCESSES; j++){
         intersection_sets[i][j] = example_sets[i][j];
      }
   }
}

// Function to simulate a process requesting critical section
void request_critical_section(int process_id) {
   processes[process_id].requested = true;
   processes[process_id].timestamp++; // Increment logical clock

   // Send request to all processes in the intersection set
   for (int i = 0; i < NUM_PROCESSES; i++) {
      if (intersection_sets[process_id][i] == -1) break; // End of set
      int target_process_id = intersection_sets[process_id][i];
```

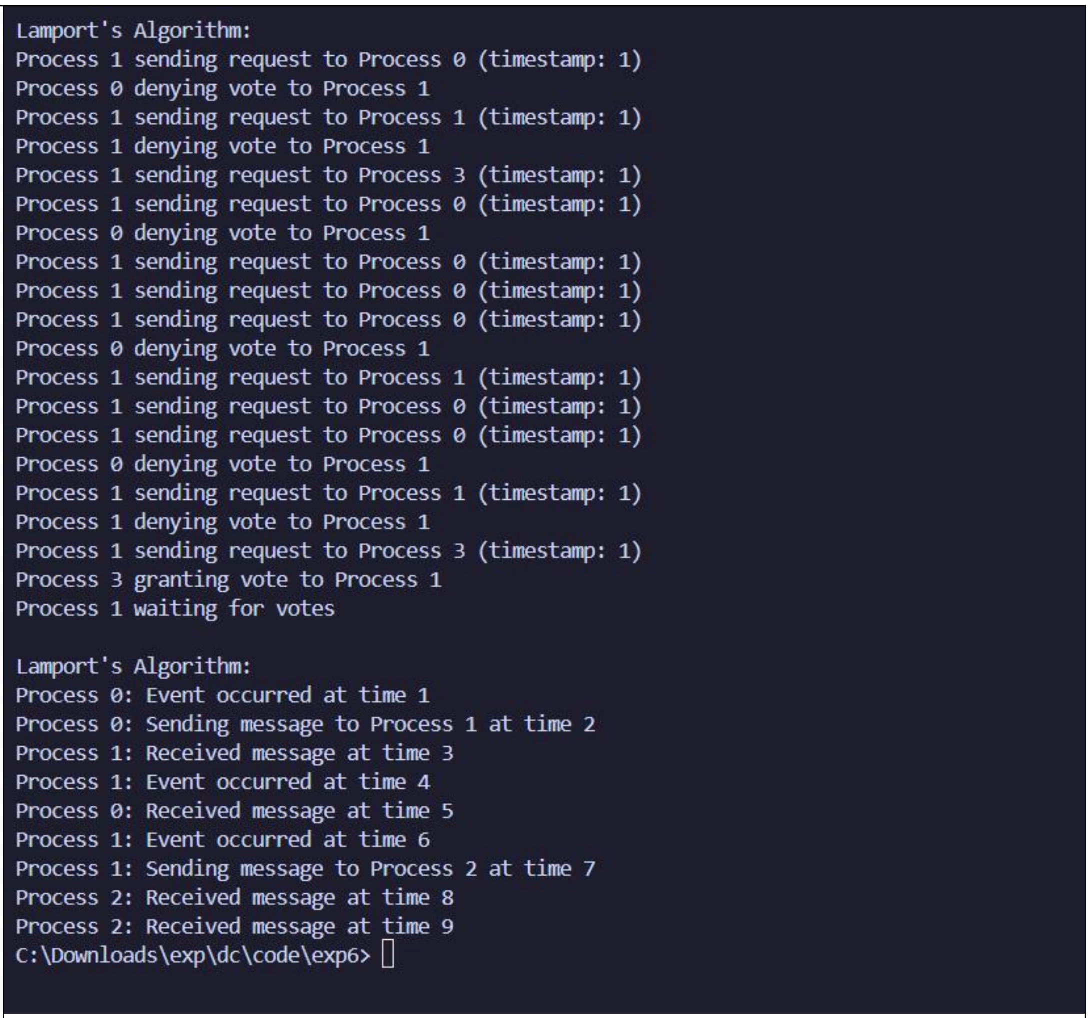
```
        // Simulate sending a request message
        printf("Process %d sending request to Process %d (timestamp: %d)\n", process_id, target_process_id, processes[process_id].timestamp);

        // Simulate receiving the request and granting vote
        if (processes[target_process_id].vote == 0 && !processes[target_process_id].requested) {
            processes[target_process_id].vote = 1;
            printf("Process %d granting vote to Process %d\n", target_process_id, process_id);
        } else {
            printf("Process %d denying vote to Process %d\n", target_process_id, process_id);
        }
    }

    // Check if the process has received votes from all members of its set
    int votes_received = 0;
    for (int i = 0; i < NUM_PROCESSES; i++) {
        if (intersection_sets[process_id][i] == -1) break;
        int target_process_id = intersection_sets[process_id][i];
        if (processes[target_process_id].vote == 1) {
            votes_received++;
        }
    }

    // If all votes received, enter critical section
    int set_size = 0;
    for (int i = 0; i < NUM_PROCESSES; i++){
        if (intersection_sets[process_id][i] == -1) break;
        set_size++;
    }

    if (votes_received == set_size) {
        processes[process_id].held = true;
        printf("Process %d entering critical section\n", process_id);

        // Simulate critical section execution (e.g., sleep)
        // ...

        // Release critical section
        processes[process_id].held = false;
        processes[process_id].requested = false;
        printf("Process %d exiting critical section\n", process_id);

        // Reset votes
        for (int i = 0; i < NUM_PROCESSES; i++) {
            if (intersection_sets[process_id][i] == -1) break;
            int target_process_id = intersection_sets[process_id][i];
            processes[target_process_id].vote = 0;
        }

    } else {
        printf("Process %d waiting for votes\n", process_id);
    }
}

// Lamport's Algorithm

int logical_clock = 0;

typedef struct {
    int process_id;
    int timestamp;
} Event;

int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```
void lamport_event(int process_id);
void lamport_send_message(int sender_id, int receiver_id);
void lamport_receive_message(int receiver_id, int message_timestamp);
//function prototype added here.

void lamport_event(int process_id) {
   logical_clock++;
       printf("Process %d: Event occurred at time %d\n", process_id,
logical_clock);
}

void lamport_send_message(int sender_id, int receiver_id) {
   logical_clock++;
    printf("Process %d: Sending message to Process %d at time %d\n",
sender_id, receiver_id, logical_clock);
   lamport_receive_message(receiver_id, logical_clock);
}

void lamport_receive_message(int receiver_id, int message_timestamp) {
   logical_clock = max(logical_clock, message_timestamp) + 1;
     printf("Process %d: Received message at time %d\n", receiver_id,
logical_clock);
}

int main() {
   // Maekawa's Algorithm Example
   printf("Maekawa's Algorithm:\n");
   initialize_processes();
   request_critical_section(0); // Process 0 requests critical section
   request_critical_section(1); // Process 1 requests critical section

   printf("\nLamport's Algorithm:\n");

   // Lamport's Algorithm Example
   lamport_event(0);
   lamport_send_message(0, 1);
```

	lamport_event(1);
	lamport_receive_message(0, logical_clock);
	lamport_event(1);
	lamport_send_message(1, 2);
	lamport_receive_message(2, logical_clock);
	
	return 0;
	}

Output	 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
C:\Downloads\exp\dc\code\exp6> gcc maekawa_lamport.c -o maekawa_lamport
C:\Downloads\exp\dc\code\exp6> ./maekawa_lamport.exe
Maekawa's Algorithm:
Process 0 sending request to Process 0 (timestamp: 1)
Process 0 denying vote to Process 0
Process 0 sending request to Process 1 (timestamp: 1)
Process 1 granting vote to Process 0
Process 0 sending request to Process 2 (timestamp: 1)
Process 2 granting vote to Process 0
Process 0 waiting for votes
Process 1 sending request to Process 0 (timestamp: 1)
Process 0 denying vote to Process 1
Process 1 sending request to Process 1 (timestamp: 1)
Process 1 denying vote to Process 1
Process 1 sending request to Process 3 (timestamp: 1)
Process 3 granting vote to Process 1
Process 1 waiting for votes
```



```
Lamport's Algorithm:
Process 1 sending request to Process 0 (timestamp: 1)
Process 0 denying vote to Process 1
Process 1 sending request to Process 1 (timestamp: 1)
Process 1 denying vote to Process 1
Process 1 sending request to Process 3 (timestamp: 1)
Process 1 sending request to Process 0 (timestamp: 1)
Process 0 denying vote to Process 1
Process 1 sending request to Process 0 (timestamp: 1)
Process 1 sending request to Process 0 (timestamp: 1)
Process 1 sending request to Process 0 (timestamp: 1)
Process 0 denying vote to Process 1
Process 1 sending request to Process 1 (timestamp: 1)
Process 1 sending request to Process 0 (timestamp: 1)
Process 1 sending request to Process 0 (timestamp: 1)
Process 0 denying vote to Process 1
Process 1 sending request to Process 1 (timestamp: 1)
Process 1 denying vote to Process 1
Process 1 sending request to Process 3 (timestamp: 1)
Process 3 granting vote to Process 1
Process 1 waiting for votes

Lamport's Algorithm:
Process 0: Event occurred at time 1
Process 0: Sending message to Process 1 at time 2
Process 1: Received message at time 3
Process 1: Event occurred at time 4
Process 0: Received message at time 5
Process 1: Event occurred at time 6
Process 1: Sending message to Process 2 at time 7
Process 2: Received message at time 8
Process 2: Received message at time 9
C:\Downloads\exp\dc\code\exp6>
```

Conclusion :

Both **Maekawa's Algorithm** and **Lamport's Algorithm** solve the mutual exclusion problem but with different strategies. Maekawa's algorithm focuses on reducing message overhead by using smaller, localized voting sets, while Lamport's algorithm relies on logical timestamps to provide a total ordering of events and ensures mutual exclusion in a distributed system.

- **Maekawa's Algorithm** minimizes message passing by ensuring that each process only needs to request permission from a subset of other processes.
- **Lamport's Algorithm** ensures mutual exclusion through the use of logical clocks and ensures a total ordering of events across processes in a distributed system.

References :

https://en.wikipedia.org/wiki/Mutual_exclusion

https://www.geeksforgeeks.org/lamports-logical-clock/

https://github.com/anoopss87/Maekawa-s-Algorithm

Rubrics for Assessment

Timely Submission	Submitted after 2 weeks 0	Submitted after deadline 1	On time Submission 2
Understanding	Student is confused about the concept 0	Students has justifiably understood the concept 2	Students is very clear about the concepts 3
Performance	Students has not performed the Experiment 0	Student has performed with help 2	Student has independently performed the experiment 3
Development	Student struggles to write code 0	Student can write code the requirement stated 1	Student can write exceptional code with his own ideas 2

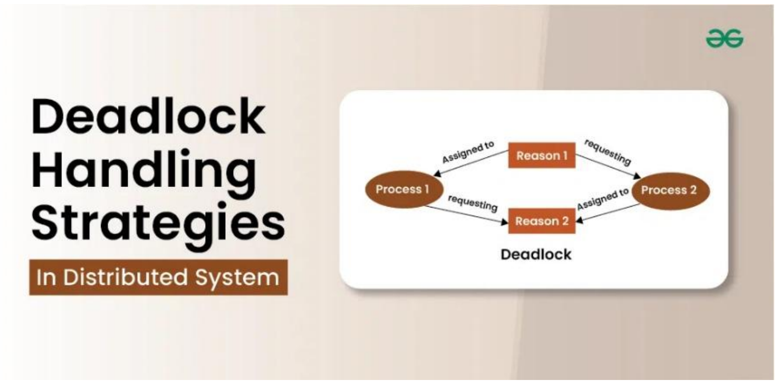
Distributed Computing

Experiment - 7

Name: Gaven Dcosta

Roll No: 09

Batch: A

Deadlock Management	
Learning Objective:	Program to demonstrate Deadlock Management in Distributed Systems
Learning Outcome:	Students will gain ability to demonstrate Deadlock Management in Distributed Systems
Course Outcome:	**CSL801.2**
Program Outcome:	**(PO3) To learn techniques of resource and process management:** CSL801.5: Implement techniques of resource and process management.
Theory:	 Deadlock Handling Strategies In Distributed System Process 1 Process 2 Reason 1 Reason 2 Assigned to requesting requesting Assigned to Deadlock In distributed systems, a deadlock is a situation where a set of distributed processes is unable to proceed because each process is waiting for resources held by others in the set, leading to a cyclic dependency among processes. Unlike centralized systems, distributed systems face additional complexities due to the lack of a global clock and the physical separation of processes across different nodes. **Chandy-Misra-Haas's distributed deadlock detection algorithm** is an edge chasing algorithm to detect deadlock in distributed systems. In edge chasing algorithm, a special message called *probe* is used in

	deadlock detection. A *probe* is a triplet *(i, j, k)* which denotes that process P_i has initiated the deadlock detection and the message is being sent by the home site of process P_j to the home site of process P_k. The probe message circulates along the edges of WFG to detect a cycle. When a blocked process receives the probe message, it forwards the probe message along its outgoing edges in WFG. A process P_i declares the deadlock if probe messages initiated by process P_i returns to itself. **Other terminologies used in the algorithm:** 1. **Dependent process:** A process P_i is said to be dependent on some other process P_j, if there exists a sequence of processes P_i, P_{i1}, P_{i2}, P_{i3}…, P_{im}, P_j such that in the sequence, each process except P_j is blocked and each process except P_i holds a resource for which previous process in the sequence is waiting. 2. **Locally dependent process:** A process P_i is said to be locally dependent on some other process P_j if the process P_i is *dependent* on process P_j and both are at same site. **Data structures:** A boolean array, **dependent$_i$**. Initially, **dependent$_i$[j]** is false for all value of i and j. **dependent$_i$[j]** is true if process P_j is dependent on process P_i. **Performance:** • Algorithm requires at most exchange of *m(n-1)/2* messages to detect deadlock. Here, m is number of processes and n is the number of sites. • The delay in detecting the deadlock is *O(n)*.
Algorithm :	*Process of sending probe:* *1. If process P_i is locally dependent on itself then declare a deadlock.* *2. Else for all P_j and P_k check following condition:* • *(a). Process P_i is locally dependent on process P_j* • *(b). Process P_j is waiting on process P_k* • *(c). Process P_j and process P_k are on different sites.* *If all of the above conditions are true, send probe (i, j, k) to the home site of*

process P_k.

On the receipt of probe (i, j, k) at home site of process P_k:

1. Process P_k checks the following conditions:

- *(a). Process P_k is blocked.*
- *(b). $dependent_k[i]$ is false.*
- *(c). Process P_k has not replied to all requests of process P_j*

If all of the above conditions are found to be true then:

1. Set $dependent_k[i]$ to true.
2. Now, If k == i then, declare the P_i is deadlocked.
3. Else for all P_m and P_n check following conditions:

- *(a). Process P_k is locally dependent on process P_m and*
- *(b). Process P_m is waiting upon process P_n and*
- *(c). Process P_m and process P_n are on different sites.*

4. Send probe (i, m, n) to the home site of process P_n if above conditions satisfy.

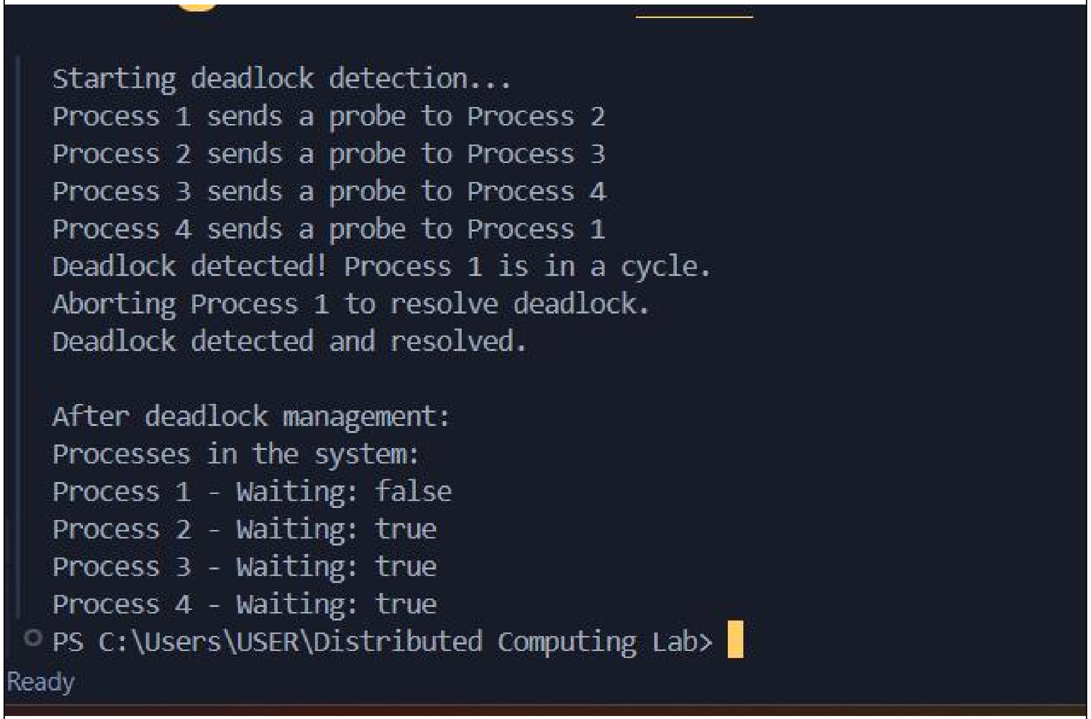
Thus, the probe message travels along the edges of transaction wait-for (TWF) graph and when the probe message returns to its initiating process then it is said that deadlock has been detected.

Program :

```
import java.util.*;


class Process {
   int id;
   boolean isWaiting;  // Track if process is waiting for a resource
    List<Process> waitingFor;  // List of processes this process is waiting on
      Set<Integer> visitedProbes; // Set to track visited probes to detect cycles


   public Process(int id) {
       this.id = id;
       this.isWaiting = false;
       this.waitingFor = new ArrayList<>();
       this.visitedProbes = new HashSet<>();
   }
```

```
    // Add dependency: process is waiting for another process
    public void addDependency(Process p) {
        this.waitingFor.add(p);
        this.isWaiting = true;
    }

    // Abort the process (deadlock management)
    public void abort() {
            System.out.println("Aborting Process " + this.id + " to resolve
deadlock.");
        this.isWaiting = false;
        this.waitingFor.clear();
    }
}

class DeadlockDetector {
    private Map<Integer, Process> processes; // Store all processes

    public DeadlockDetector() {
        this.processes = new HashMap<>();
    }

    // Add process to system
    public void addProcess(int id) {
        processes.put(id, new Process(id));
    }

    // Create a dependency: Pi waits for Pj
    public void addDependency(int from, int to) {
        if (processes.containsKey(from) && processes.containsKey(to)) {
            processes.get(from).addDependency(processes.get(to));
        }
    }

    // Start deadlock detection
    public boolean detectDeadlock() {
        System.out.println("Starting deadlock detection...");
        for (Process process : processes.values()) {
            if (process.isWaiting) {
                Set<Integer> visited = new HashSet<>();
                if (isCyclic(process, visited)) {
                    return true;
                }
            }
        }
        return false;
    }

    // Helper method to simulate probe message propagation and check
for cycles
    private boolean isCyclic(Process process, Set<Integer> visited) {
        if (visited.contains(process.id)) {
            // Cycle detected (deadlock)
             System.out.println("Deadlock detected! Process " + process.id + "
is in a cycle.");
            // Deadlock management: Abort the process involved in the cycle
            process.abort();
            return true;
        }

        // Mark the current process as visited
        visited.add(process.id);
        for (Process dependent : process.waitingFor) {
            // Simulate sending a probe to the dependent process
              System.out.println("Process " + process.id + " sends a probe to
Process " + dependent.id);
            if (isCyclic(dependent, new HashSet<>(visited))) {
                return true;
            }
        }
        return false;
    }
    // Getter method for processes
    public Map<Integer, Process> getProcesses() {
        return processes;
    }
}

public class ChandyMisraHaassWithDeadlockManagement {
    public static void main(String[] args) {
        // Step 1: Initialize the deadlock detector
        DeadlockDetector detector = new DeadlockDetector();

        // Step 2: Add processes to the system
        detector.addProcess(1);
        detector.addProcess(2);
        detector.addProcess(3);
        detector.addProcess(4);

        // Step 3: Establish dependencies (resource waits)
        detector.addDependency(1, 2); // Process 1 waits for Process 2
        detector.addDependency(2, 3); // Process 2 waits for Process 3
        detector.addDependency(3, 4); // Process 3 waits for Process 4
            detector.addDependency(4, 1); // Process 4 waits for Process 1
(creating a cycle)

        // Step 4: Detect deadlock
        if (detector.detectDeadlock()) {
            System.out.println("Deadlock detected and resolved.");
        } else {
            System.out.println("No deadlock detected.");
        }

        // Step 5: Verify if deadlock was resolved
        System.out.println("\nAfter deadlock management:");
        System.out.println("Processes in the system:");
        for (Process process : detector.getProcesses().values()) {
```

	`System.out.println("Process " + process.id + " - Waiting: " + process.isWaiting);`
	`}`
	`}`
	`}`

Output	 Starting deadlock detection... Process 1 sends a probe to Process 2 Process 2 sends a probe to Process 3 Process 3 sends a probe to Process 4 Process 4 sends a probe to Process 1 Deadlock detected! Process 1 is in a cycle. Aborting Process 1 to resolve deadlock. Deadlock detected and resolved. After deadlock management: Processes in the system: Process 1 - Waiting: false Process 2 - Waiting: true Process 3 - Waiting: true Process 4 - Waiting: true PS C:\Users\USER\Distributed Computing Lab> Ready
Conclusion :	The Chandy-Misra-Haas distributed deadlock detection algorithm effectively identifies deadlocks in a distributed system by using a wait-for graph and probe messages to detect cyclic dependencies between processes. The key feature of this simulation was the handling of such deadlocks through aborting processes involved in the cycle, which is an essential deadlock management strategy.
References :	1. https://www.geeksforgeeks.org/chandy-misra-haass-distributed-deadlock-detection-algorithm/ 2. Google 3. ChatGPT

Rubrics for Assessment

Timely Submission	Submitted after 2 weeks 0	Submitted after deadline 1	On time Submission 2
Understanding	Student is confused about the concept 0	Students has justifiably understood the concept 2	Students is very clear about the concepts 3
Performance	Students has not performed the Experiment 0	Student has performed with help 2	Student has independently performed the experiment 3
Development	Student struggles to write code 0	Student can write code the requirement stated 1	Student can write exceptional code with his own ideas 2

Distributed Computing Experiment - 8

Name: Gaven Dcosta

Roll No: 09

Batch: A

Program to demonstrate Load Balancing Algorithm in Java	
Learning Objective:	Ability to demonstrate Load Balancing Algorithm in Java
Learning Outcome:	Ability to demonstrate Load Balancing Algorithm in Java
Course Outcome:	**CSL801.5:** Implement techniques of resource and process management.
Program Outcome:	**To Learn about load balancing algorithm in distributed system**
Theory:	In distributed systems, load balancing is a crucial technique for distributing workloads across multiple servers or resources to optimize performance, improve reliability, and ensure scalability. Here's a breakdown of key concepts: **Core Concepts:** • **Distribution of Workload:** o The primary goal is to prevent any single server from becoming overloaded while others remain idle. o This involves distributing incoming network traffic, processing tasks, or data requests evenly across available resources. • **Benefits:** o **Improved Performance:** Load balancing reduces response times and latency by distributing the workload. o **Increased Availability:** If one server fails, the load balancer redirects traffic to the remaining healthy servers, ensuring continuous service. o **Enhanced Scalability:** Load balancing allows systems to handle increased traffic by adding more servers to the pool. o **Fault Tolerance:** It helps to mitigate the effects of server failures.

Key Aspects:

- **Load Balancing Algorithms:**
    - Various algorithms are used to determine how to distribute the workload. Common ones include:
        - **Round-robin:** Distributes requests sequentially to each server.
        - **Weighted round-robin:** Distributes requests based on server capacity.
        - **Least connections:** Directs requests to the server with the fewest active connections.
        - **Least response time:** Directs requests to the server with the fastest response time.
        - **Hashing:** Uses a hash function to consistently direct requests from the same client to the same server.
- **Types of Load Balancing:**
    - **Network Load Balancing (Layer 4):**
        - Operates at the network layer (TCP/IP).
        - Distributes traffic based on IP addresses and ports.
    - **Application Load Balancing (Layer 7):**
        - Operates at the application layer (HTTP/HTTPS).
        - Distributes traffic based on application-level data, such as URLs, headers, and content.
- **Static vs. Dynamic Load Balancing:**
    - **Static Load Balancing:**
        - Decisions are made before runtime.
        - Simple but less flexible.
    - **Dynamic Load Balancing:**
        - Decisions are made in real-time based on current server loads.
        - More complex but more efficient.

Round Robin Method :

The round-robin method of load balancing is one of the simplest and most widely used techniques for distributing network traffic across multiple servers. Here's a breakdown of how it works:

Core Principle:

- The round-robin algorithm distributes incoming requests sequentially to each server in a rotating order.
- Imagine a list of servers. When a new request arrives, it's sent to the first server on the list. The next request goes to the second server, and so on.
- Once the load balancer reaches the end of the list, it loops back to the beginning and starts the cycle again.

Key Characteristics:

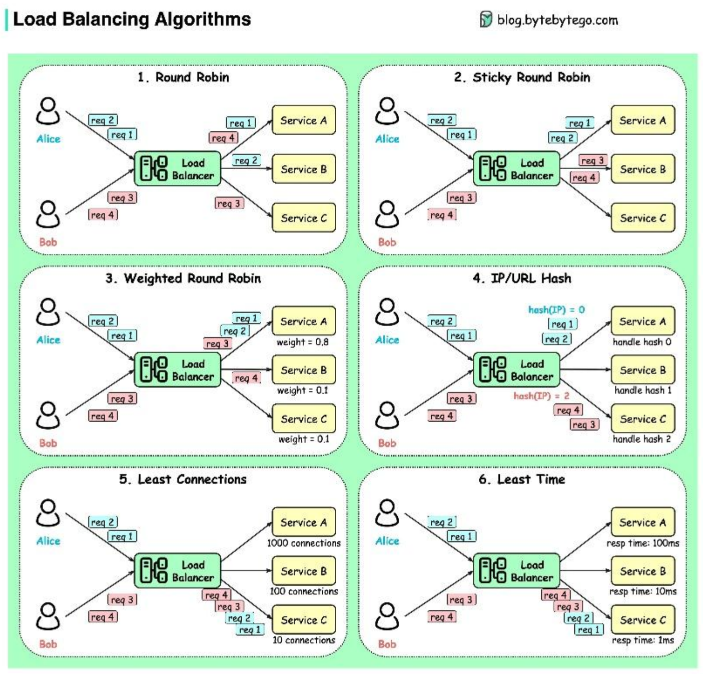
- **Simplicity:**
  - It's easy to understand and implement, making it a popular choice for basic load balancing needs.
- **Even Distribution (in ideal scenarios):**
  - In a perfect scenario where all servers have identical capabilities and handle requests at the same speed, round-robin provides an even distribution of the workload.
- **Limitations:**
  - It doesn't consider server load: A server that is already overloaded will still receive the same number of requests as an idle server.
  - It doesn't account for server performance: Servers with different processing power are treated equally.

When to Use Round Robin:

- Round robin is most effective when:
  - The servers in the pool have similar processing power and resources.
  - The workload is relatively consistent.
  - Simplicity and ease of implementation are priorities.

Variations:

- **Weighted Round Robin:**
  - This variation addresses the limitations of standard round-robin by assigning weights to each server.
  - Servers with higher weights receive a larger proportion of the workload.

	○ This allows for more efficient distribution when servers have different capacities.  Load Balancing Algorithms blog.bytebytego.com 1. Round Robin 2. Sticky Round Robin 3. Weighted Round Robin 4. IP/URL Hash 5. Least Connections 6. Least Time Alice Bob Load Balancer Service A Service B Service C req 1 req 2 req 3 req 4 weight = 0.8 weight = 0.1 weight = 0.1 hash(IP) = 0 hash(IP) = 2 handle hash 0 handle hash 1 handle hash 2 1000 connections 100 connections 10 connections resp time: 100ms resp time: 10ms resp time: 1ms
Algorithm :	**Algorithm: Round-Robin Load Balancing** **Input:** • Servers: A list of available servers (e.g., ["server1", "server2", "server3"]). • Requests: A list of incoming requests (e.g., ["requestA", "requestB", "requestC", "requestD"]). **Output:** • A mapping (e.g., a dictionary) of requests to the servers they are assigned to. **Steps:** 1. **Initialize Server Index:** ○ Set a variable serverIndex to 0. This variable will keep track of which server to use next. 2. **Initialize Assignments:** ○ Create an empty data structure (e.g., a dictionary called assignments) to store the request-to-server mappings. 3. **Iterate Through Requests:** ○ For each request in the Requests list, perform the following steps: 4. **Assign Request to Server:** ○ Get the server from the Servers list at the index serverIndex. ○ Store the mapping of the request to the server in the assignments data structure. For example, assignments[request] = Servers[serverIndex]. 5. **Increment Server Index:** ○ Increase the serverIndex by 1. 6. **Cycle Through Servers:** ○ If the serverIndex is equal to the number of servers in the Servers list, reset

the serverIndex to 0. This ensures that the servers are used in a circular manner. This is done with the modulo operator:

- serverIndex = serverIndex % (number of servers)

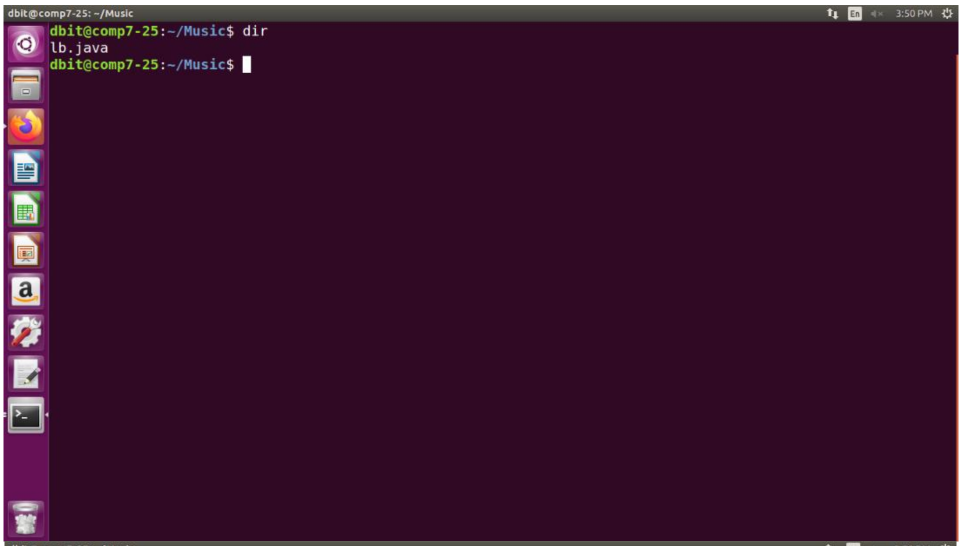
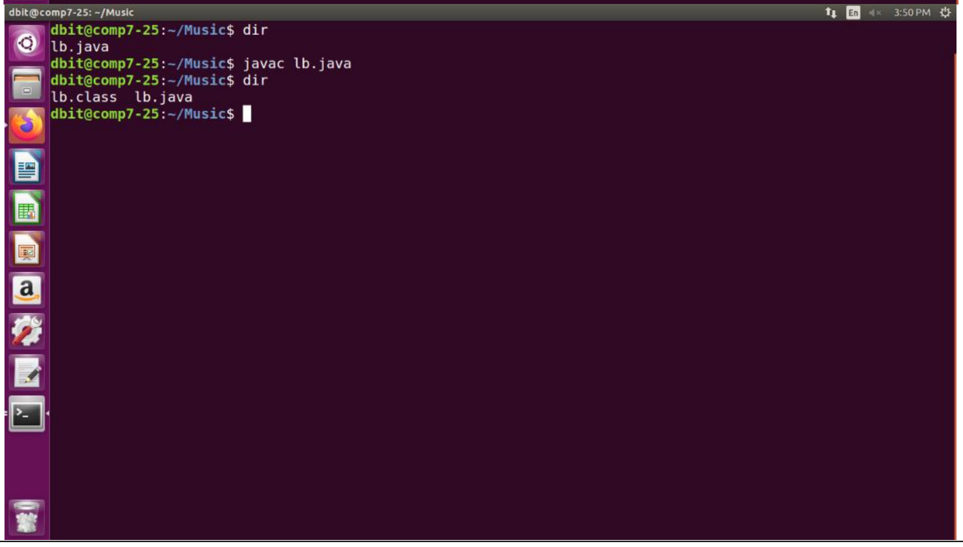
7. **Return Assignments:**
    - Once all requests have been processed, return the assignments data structure.

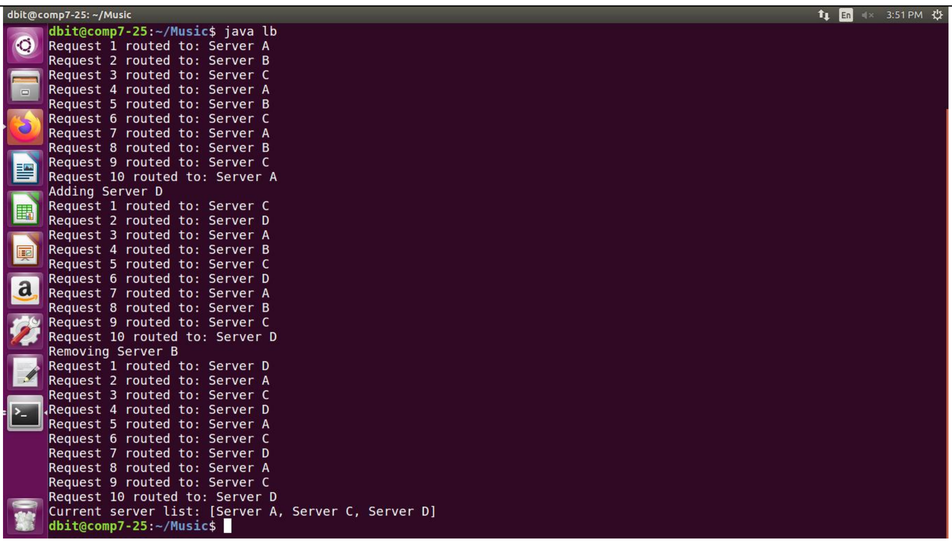
Program :

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

public class lb<T> {

  private final List<T> servers;
  private final AtomicInteger nextServer;

  public lb(List<T> servers) {
    if (servers == null || servers.isEmpty()) {
      throw new IllegalArgumentException("Servers list cannot be null or empty.");
    }
    this.servers = new ArrayList<>(servers); // Create a copy to avoid external modifications
    this.nextServer = new AtomicInteger(0);
  }

  public T getNextServer() {
    int index = nextServer.getAndIncrement();
    if (index < 0) { // Handle integer overflow if it occurs
      index = nextServer.getAndSet(0);
    }
    return servers.get(index % servers.size());
  }
```

```
    public void addServer(T server) {
        if (server != null) {
            servers.add(server);
        }
    }

    public void removeServer(T server) {
        servers.remove(server);
    }

    public List<T> getServers() {
        return new ArrayList<>(servers); // Return a copy to prevent external
modification.
    }

    public static void main(String[] args) {
        List<String> serverList = new ArrayList<>();
        serverList.add("Server A");
        serverList.add("Server B");
        serverList.add("Server C");

        lb<String> loadBalancer = new lb<>(serverList);

        for (int i = 0; i < 10; i++) {
            System.out.println("Request " + (i + 1) + " routed to: " +
loadBalancer.getNextServer());
        }

        System.out.println("Adding Server D");
        loadBalancer.addServer("Server D");

        for (int i = 0; i < 10; i++) {
            System.out.println("Request " + (i + 1) + " routed to: " +
```

	loadBalancer.getNextServer());
	        }
	
	        System.out.println("Removing Server B");
	        loadBalancer.removeServer("Server B");
	
	        for (int i = 0; i < 10; i++) {
	            System.out.println("Request " + (i + 1) + " routed to: " + loadBalancer.getNextServer());
	        }
	
	        System.out.println("Current server list: " + loadBalancer.getServers());
	    }
	}

Output	 dbit@comp7-25:~/Music$ dir lb.java dbit@comp7-25:~/Music$  dbit@comp7-25:~/Music$ dir lb.java dbit@comp7-25:~/Music$ javac lb.java dbit@comp7-25:~/Music$ dir lb.class lb.java dbit@comp7-25:~/Music$



```
dbit@comp7-25: ~/Music                                                    En  3:51 PM
dbit@comp7-25:~/Music$ java lb
Request 1 routed to: Server A
Request 2 routed to: Server B
Request 3 routed to: Server C
Request 4 routed to: Server A
Request 5 routed to: Server B
Request 6 routed to: Server C
Request 7 routed to: Server A
Request 8 routed to: Server B
Request 9 routed to: Server C
Request 10 routed to: Server A
Adding Server D
Request 1 routed to: Server C
Request 2 routed to: Server D
Request 3 routed to: Server A
Request 4 routed to: Server B
Request 5 routed to: Server C
Request 6 routed to: Server D
Request 7 routed to: Server A
Request 8 routed to: Server B
Request 9 routed to: Server C
Request 10 routed to: Server D
Removing Server B
Request 1 routed to: Server D
Request 2 routed to: Server A
Request 3 routed to: Server C
Request 4 routed to: Server D
Request 5 routed to: Server A
Request 6 routed to: Server C
Request 7 routed to: Server D
Request 8 routed to: Server A
Request 9 routed to: Server C
Request 10 routed to: Server D
Current server list: [Server A, Server C, Server D]
dbit@comp7-25:~/Music$
```

Conclusion :	This experiment validated the round-robin load balancer's basic functionality in Java. It successfully distributed requests evenly across servers, dynamically adapted to server additions and removals, and demonstrated thread-safe request routing. The results confirm a simple yet effective implementation of the round-robin algorithm.
References :	https://www.vmware.com/topics/round-robin-load-balancing https://www.cloudflare.com/learning/performance/types-of-load-balancing-algorithms/

Rubrics for Assessment

Timely Submission	Submitted after 2 weeks 0	Submitted after deadline 1	On time Submission 2
Understanding	Student is confused about the concept 0	Students has justifiably understood the concept 2	Students is very clear about the concepts 3
Performance	Students has not performed the Experiment 0	Student has performed with help 2	Student has independently performed the experiment 3
Development	Student struggles to write code 0	Student can write code the requirement stated 1	Student can write exceptional code with his own ideas 2

DC Experiment 9

Case Study on CORBA Distributed File System

Name: Gaven Dcosta Roll No.: 09 Batch: A

Introduction:

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together.

CORBA is a standard for distributing objects across networks so that operations on those objects can be called remotely. CORBA is not associated with a particular programming language, and any language with a CORBA binding can be used to call and implement CORBA objects. Objects are described in a syntax called Interface Definition Language (IDL).

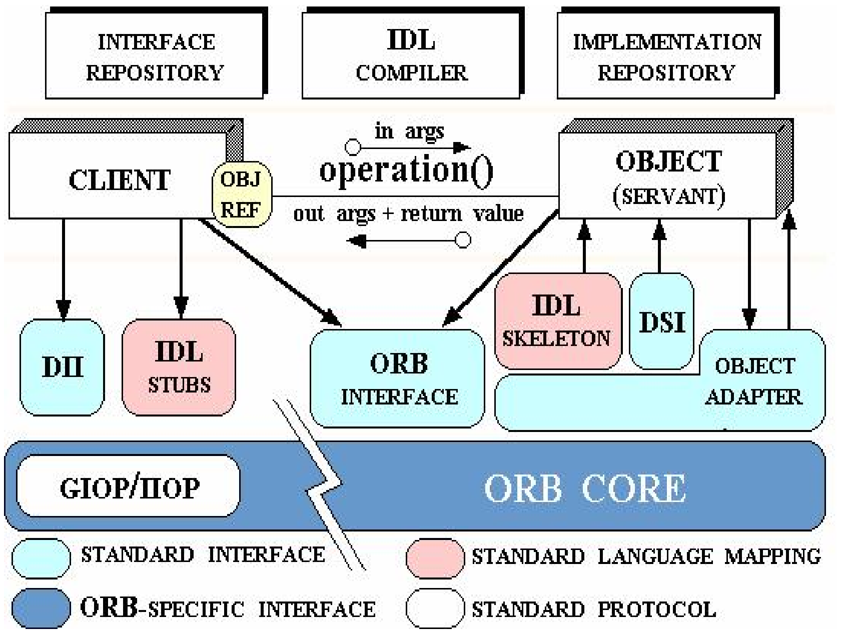
CORBA allows developers to build distributed applications in a language-independent manner, enabling interoperability between different systems and platforms. It provides features such as object persistence, object lifecycle management, and remote method invocation. CORBA was widely used in the 1990s and early 2000s for building large-scale distributed systems, but its popularity has decreased with the rise of alternative technologies such as web services and RESTful APIs

Architecture of CORBA:

The following are the components of the CORBA ORB architecture, more properly detailed in the image given below it.

- **Object** -- This is a CORBA programming entity that consists of an *identity*, an *interface*, and an *implementation*, which is known as a *Servant*.
- **Servant** -- This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.
- **Client** -- This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., obj->op(args). The remaining components in Figure 2 help to support this level of transparency.

- **Object Request Broker (ORB)** -- The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.
- **ORB Interface** -- An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.
- **CORBA IDL stubs and skeletons** -- CORBA IDL stubs and skeletons serve as the ``glue'' between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.
- **Dynamic Invocation Interface (DII)** -- This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking *deferred synchronous* (separate send and receive operations) and *oneway* (send-only) calls.
- **Dynamic Skeleton Interface (DSI)** -- This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.
- **Object Adapter** -- This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).



Features/Characteristics:

CORBA (Common Object Request Broker Architecture) provides several features that facilitate the development of distributed applications. Some of the key features include:

1. Language Independence: CORBA allows components written in different programming languages to communicate with each other seamlessly. This enables developers to choose the most appropriate language for each component without worrying about interoperability issues.

2. Location Transparency: With CORBA, distributed objects can be accessed using a standard interface regardless of their physical location. This abstraction allows clients to invoke methods on remote objects without needing to know their network address or implementation details.

3. Interoperability: CORBA promotes interoperability between heterogeneous systems and platforms. It defines a standard interface for communication between distributed objects, enabling components running on different hardware and operating systems to interact with each other.

4. Object-oriented Model: CORBA is based on an object-oriented model, where distributed components are represented as objects with well-defined interfaces. This allows developers to model complex systems using familiar object-oriented concepts such as inheritance,

polymorphism, and encapsulation.

5. Dynamic Invocation and Dynamic Skeleton Interface (DSI): CORBA supports dynamic invocation, which allows clients to dynamically discover and invoke methods on remote objects at runtime. DSI enables servers to handle requests from clients without pre-generated stubs, providing flexibility in system design.

6. Security: CORBA provides built-in security features to protect the integrity and confidentiality of distributed communications. This includes authentication, access control, encryption, and digital signatures to ensure secure interactions between distributed components.

7. Scalability: CORBA is designed to support scalable distributed systems by providing mechanisms for load balancing, fault tolerance, and distributed object management. This allows applications to scale seamlessly to accommodate growing numbers of clients and resources.

8. Transaction Support: CORBA supports distributed transactions, allowing multiple operations to be grouped together and executed as a single atomic unit. This ensures data consistency and integrity across distributed components, even in the presence of failures or concurrency issues.

Advantages of CORBA:

Platform and Language Independence: CORBA is language-agnostic, supporting multiple programming languages like C++, Java, Python, and more. It can operate across diverse platforms.

Scalability: CORBA was designed with scalability in mind, making it suitable for large, enterprise-level applications.

High Level of Abstraction: CORBA allows communication between systems at a high level of abstraction, making it easier for developers to work on distributed systems.

Disadvantages of CORBA:

Complexity: The initial setup and learning curve of CORBA can be quite steep, especially for developers unfamiliar with distributed computing.

Overhead: CORBA can introduce significant overhead, particularly in terms of performance, due to its comprehensive feature set.

Security Concerns: While security standards exist for CORBA, its older versions have been criticized for weak or insufficient security measures concurrency and handling exceptions.

Goals of CORBA:

Interoperability: Enable seamless communication between distributed components regardless of the programming languages, hardware, or operating systems they use.

Scalability: Provide mechanisms for building distributed systems that can handle growing numbers of clients and resources, ensuring they can scale effectively.

Flexibility: Offer a flexible and extensible framework for designing and implementing distributed systems, allowing developers to adapt to evolving requirements and technologies.

References:

https://www.corba.org/

https://www.geeksforgeeks.org/difference-between-corba-and-dcom/

https://www.ibm.com/docs/en/integration-bus/10.0?topic=corba-common-object-request-broker-architecture

https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture

Distributed Computing Experiment -10

Name: Gaven Dcosta

Roll No: 09

Batch: A

Innovative Experiment : Program to demonstrate Deadlock Detection using Path Pushing Algorithm	
Learning Objective:	Ability to solve deadlock detection problem using content beyond method.
Learning Outcome:	Ability to solve deadlock detection problem using content beyond method.
Course Outcome:	**CSL801.4:** Demonstrate mutual exclusion algorithms and deadlock handling.
Program Outcome:	**To Learn about Deadlock detection using path pushing algorithm**
Theory:	Deadlock detection in distributed systems is a complex problem, and the path-pushing algorithm is one approach to tackling it. **Understanding the Basics:** • **Deadlock:** ○ In a distributed system, a deadlock occurs when two or more processes are blocked indefinitely, each waiting for a resource held by another process[1] in the group. • **Wait-For Graph (WFG):** ○ A WFG is a directed graph used to represent the dependencies between processes. Nodes in the graph represent processes, and directed edges represent the "waits-for" relationship. • **Path-Pushing:** ○ Path-pushing algorithms aim to detect deadlocks by propagating information about these "waits-for" dependencies across the distributed system. **How Path-Pushing Works:** The core idea behind path-pushing is to distribute information about the WFG across the participating sites. This is typically done through these general steps:

- **Local WFG Maintenance:**
  - Each site in the distributed system maintains its own local WFG, representing the dependencies between processes within that site.
- **Path Information Exchange:**
  - When a site initiates a deadlock detection process, or when a change occurs in its local WFG, it sends information about the paths in its WFG to other relevant sites.
  - This information "pushes" the path data across the network.
- **Global WFG Construction (Partial):**
  - Each site receives path information from other sites and uses it to construct a partial view of the global WFG.
- **Cycle Detection:**
  - Sites analyze their partial global WFGs to detect cycles. The presence of a cycle indicates a deadlock.

Key Considerations:

- **Message Overhead:**
  - Path-pushing can generate significant message overhead, as path information is exchanged between sites.
- **Synchronization:**
  - Ensuring consistency and accuracy of the global WFG view across all sites can be challenging.
- **Algorithm Variations:**
  - There are various variations of path-pushing algorithms, each with its own strategies for path information exchange and cycle detection.

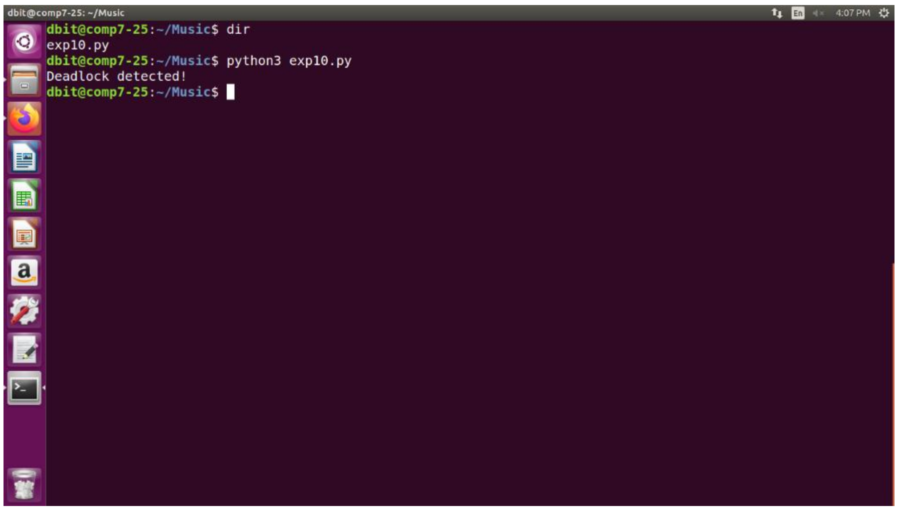
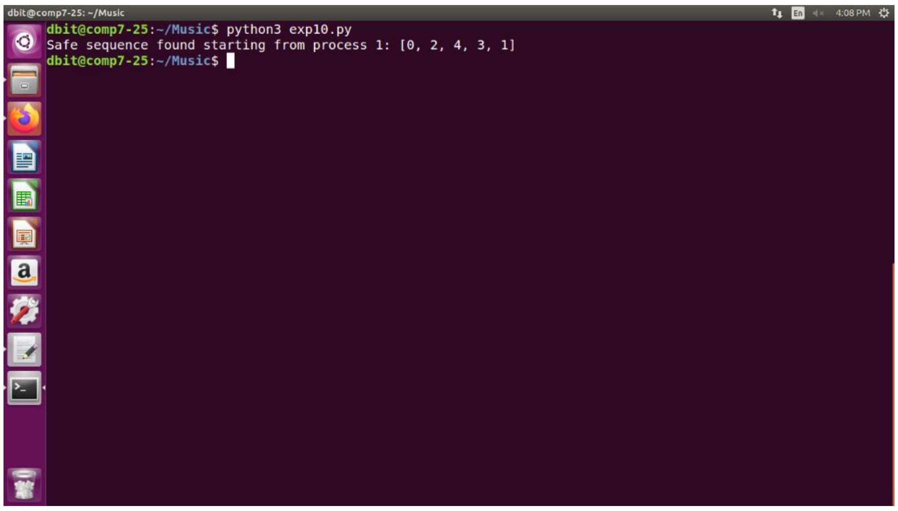
Algorithm :	**Algorithm Steps:** 1. **Local WFG Maintenance:** o Each site maintains a local WFG reflecting the dependencies among processes within its jurisdiction. 2. **Path Information Exchange:** o When a local WFG changes (e.g., a process enters a wait state), the site: ▪ Constructs path messages encoding "waits-for" sequences within its local WFG. ▪ Transmits these messages to other relevant sites. 3. **Partial Global WFG Construction:** o Each site receives path messages from other sites. o It integrates this information into its local WFG view, creating a partial view of the global WFG. 4. **Cycle Detection:** o Each site analyzes its partial global WFG for cycles. ▪ If a cycle is detected, a deadlock is identified.

Program :	
	```
	def detect_deadlock(allocation, request, available):
	  """
	  Detects deadlock using the path-pushing algorithm and shows the execution order.

	  Args:
	    allocation: A list of lists representing the allocated resources.
	    request: A list of lists representing the maximum requested resources.
	    available: A list representing the available resources.

	  Returns:
	    True if deadlock is detected, False otherwise. Also prints execution order.
	  """

	  num_processes = len(allocation)
	  num_resources = len(available)

	  work = available[:]
	  finish = [False] * num_processes
	  need = [[0] * num_resources for _ in range(num_processes)]
	```

```
for i in range(num_processes):
    for j in range(num_resources):
        need[i][j] = request[i][j] - allocation[i][j]

for i in range(num_processes):
    if all(allocation[i][j] == 0 for j in range(num_resources)):
        finish[i] = True

for start_process in range(num_processes):
    process_stack = []
    visited = [False] * num_processes
    order_of_execution = []  # Store the order of execution for printing

    if not finish[start_process]:
        process_stack.append(start_process)
        visited[start_process] = True

        while process_stack:
            current_process = process_stack[-1]

            found = False
            for j in range(num_processes):
                if not finish[j] and not visited[j]:
                    can_satisfy = True
                    for k in range(num_resources):
                        if need[current_process][k] > work[k]:
                            can_satisfy = False
                            break

                    if can_satisfy:
                        process_stack.append(j)
                        visited[j] = True
                        found = True
```

```
                            break

                if not found:
                    for k in range(num_resources):
                        work[k] += allocation[current_process][k]
                    finish[current_process] = True
                    order_of_execution.append(current_process)
                    process_stack.pop()

            if all(finish):
                print("Safe sequence found starting from process {}: {}".format(start_process, order_of_execution))
                return False  # No deadlock detected.
            else:
                # reset the finish array.
                finish = [False] * num_processes
                for i in range(num_processes):
                    if all(allocation[i][j] == 0 for j in range(num_resources)):
                        finish[i] = True
                work = available[:]

    print("Deadlock detected!")
    return True  # Deadlock detected



# Example usage
allocation = [
    [0, 1, 0],
    [2, 0, 0],
    [3, 0, 2],
    [2, 1, 1],
    [0, 0, 2],
]
```

	request = [
	    [9, 5, 3],
	    [3, 2, 2],
	    [9, 0, 2],
	    [2, 2, 2],
	    [4, 3, 3],
	]
	
	available = [3, 2, 3] # for deadlock use [0,0,0] and for safe sequence [3,2,2]
	
	detect_deadlock(allocation, request, available)

Output	Read comment on second last line of code to get both the outputs
	
	
	`dbit@comp7-25:~/Music$ dir`
	`exp10.py`
	`dbit@comp7-25:~/Music$ python3 exp10.py`
	`Deadlock detected!`
	`dbit@comp7-25:~/Music$`
	
	
	`dbit@comp7-25:~/Music$ python3 exp10.py`
	`Safe sequence found starting from process 1: [0, 2, 4, 3, 1]`
	`dbit@comp7-25:~/Music$`

Conclusion :	The above python code implements a deadlock detection algorithm, effectively simulating process execution to find a safe resource allocation sequence. It accurately identifies safe states and deadlocks, printing the execution order when safe. This demonstrates its ability to ensure resource allocation safety in concurrent systems.
References :	https://en.wikipedia.org/wiki/Deadlock_(computer_science) https://www.naukri.com/code360/library/deadlock-detection-algorithm

Rubrics for Assessment

Timely Submission	Submitted after 2 weeks 0	Submitted after deadline 1	On time Submission 2
Understanding	Student is confused about the concept 0	Students has justifiably understood the concept 2	Students is very clear about the concepts 3
Performance	Students has not performed the Experiment 0	Student has performed with help 2	Student has independently performed the experiment 3
Development	Student struggles to write code 0	Student can write code the requirement stated 1	Student can write exceptional code with his own ideas 2