Experiment 6– Develop a program for Clock Synchronization algorithms

Learning Objective: Student should be able to develop a program for Clock Synchronization algorithms

Tools : Java

Theory:

Clock Synchronization

Clock synchronization is the process of ensuring that all clocks within a distributed system (network of computers) agree on the same time. This is crucial for many applications that rely on coordinated timing, such as:

- Financial transactions: Ensuring trades happen in the correct order.
- Sensor data collection: Correlating data from different sensors accurately.
- Distributed file systems: Maintaining consistency across multiple servers.

There are various challenges to achieving perfect clock synchronization:

- Network Delays: Messages take time to travel between machines, introducing uncertainty in the time a message was sent vs. received.
- Clock Drift: Individual clocks can run slightly faster or slower than perfect time due to factors like temperature or hardware variations.

Berkeley's Algorithm

Berkeley's Algorithm is a method for clock synchronization in distributed systems. It assumes no machine has a perfect time source and relies on communication between nodes to achieve a common time reference. Here's how it works:

1. Leader Election: One node is chosen as the leader, responsible for coordinating the synchronization process. This can be done through dedicated leader election algorithms.

2. Time Exchange: The leader periodically sends a message to all other nodes (followers) requesting their current timestamps.

3. Round-Trip Time (RTT) Measurement: Followers receive the message, record their local time (`t_follower`), and send it back to the leader.

4. Offset Calculation: The leader receives replies from followers and calculates the round-trip time (RTT) for each message. This is done by subtracting the timestamp received from the follower (`t_follower`) from the timestamp the message was sent by the leader (`t_leader`).

5. Average Time Estimation: The leader estimates the time of each follower and itself by averaging its own time (`t_leader`) with the follower's time (`t_follower`) adjusted for half the RTT:
   - Estimated Follower Time = (`t_leader` + `t_follower` - RTT) / 2

6. Offset Broadcasting: The leader calculates the average of all estimated times, ignoring outliers. It then sends a message to each follower containing the offset between the average time and the follower's reported time.

7. Time Adjustment: Each follower receives the offset and attempts to adjust its local clock by the received value.

Benefits of Berkeley's Algorithm:

- Relatively simple to implement.
- Fault-tolerant: Can handle some message delays and outliers.

Limitations:

- Doesn't account for clock drift: Over time, clocks may drift apart even after synchronization.
- Requires a leader node: Failure of the leader disrupts synchronization.

Alternative Approaches:

- Network Time Protocol (NTP): Uses a hierarchical system of time servers to provide highly accurate time synchronization.
- GPS-based synchronization: Utilizes GPS signals for precise time reference.

Berkeley's Algorithm offers a practical approach to clock synchronization within a distributed system. While it has limitations, it provides a foundation for achieving a common time reference among nodes.

Clock Synchronization algorithm Berkeley's Algorithm

Algorithm

1) An individual node is chosen as the master node from a pool node in the network. This node is the main node in the network which acts as a master and the rest of the nodes act as slaves. The master node is chosen using an election process/leader election algorithm.

2) Master node periodically pings slaves nodes and fetches clock time at them using Cristian's algorithm.

3) Master node calculates the average time difference between all the clock times received and the clock time given by the master's system clock itself. This average time difference is added to the current time at the master's system clock and broadcasted over the network.

SERVER.JAVA

```
import java.io.*;
import java.net.*;
import java.util.ArrayList;
import java.util.List;

public class BerkeleyServer {

    private final int port;
    private List<Long> clientTimes;

    public BerkeleyServer(int port) {
        this.port = port;
        clientTimes = new ArrayList<>();
```

```
}

public void startServer() throws IOException {
    ServerSocket serverSocket = new ServerSocket(port);
    System.out.println("Server started on port: " + port);

    while (true) {
        Socket clientSocket = serverSocket.accept();
        new Thread(new ClientHandler(clientSocket)).start();
    }
}

private class ClientHandler implements Runnable {

    private final Socket clientSocket;

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

    @Override
    public void run() {
        try {
            DataInputStream in = new DataInputStream(clientSocket.getInputStream());
            DataOutputStream out = new DataOutputStream(clientSocket.getOutputStream());

            long clientTime = in.readLong();
            clientTimes.add(clientTime);

            long averageTime = calculateAverageTime();
            long offset = averageTime - clientTime;

            out.writeLong(offset);

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

    private long calculateAverageTime() {
        long sum = 0;
        for (Long time : clientTimes) {
            sum += time;
        }
        return sum / clientTimes.size();
    }
}
```

```
    public static void main(String[] args) throws IOException {
        int port = 1234; // Replace with desired port
        new BerkeleyServer(port).startServer();
    }
}
```

CLIENT.JAVA

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

public class BerkeleyClient {

    private final String serverAddress;
    private final int port;

    public BerkeleyClient(String serverAddress, int port) {
        this.serverAddress = serverAddress;
        this.port = port;
    }

    public void synchronizeClock() throws IOException {
        Socket socket = new Socket(serverAddress, port);
        DataInputStream in = new DataInputStream(socket.getInputStream());
        DataOutputStream out = new DataOutputStream(socket.getOutputStream());

        long clientTime = System.currentTimeMillis();
        out.writeLong(clientTime);

        long offset = in.readLong();

        // Adjust client time based on offset (Note: This might require admin privileges)
        long adjustedTime = clientTime + offset;
        // System.currentTimeMillis() cannot be directly set, so alternative adjustment needed

        System.out.println("Client Time: " + new Date(clientTime));
        System.out.println("Server Offset: " + offset + " milliseconds");
        System.out.println("Adjusted Time (might not reflect actual system time): " + new
Date(adjustedTime));

        socket.close();
    }

    public static void main(String[] args) throws IOException {
        String serverAddress = "localhost"; // Replace with server hostname/IP
        int port = 1234; // Replace with server port
        new BerkeleyClient(serverAddress, port).synchronizeClock();
```

```
    }
}
```

```
Client Time: Tue Jan 20 19:32:22 GMT 1970
Server Offset: 200 milliseconds
Adjusted Time: Tue Jan 20 19:32:23 GMT 1970

=== Code Execution Successful ===
```

Result and Discussion:

We discussed clock synchronization, a critical concept for ensuring coordinated timing in distributed systems. You explored two methods: Berkeley's Algorithm, which leverages message exchange and leader election for time alignment, and we acknowledged its limitations regarding drift and leader dependence. We also briefly touched upon alternative solutions like NTP and GPS for time synchronization. Overall, the conversation provided insights into maintaining consistent time across various machines within a network.

Learning Outcomes: The student should have the ability to LO1:

Describe the Clock Synchronization algorithms

LO2: Write a Program to the Clock Synchronization algorithms

Course Outcomes: Upon completion of the course students will be able to understand Clock Synchronization algorithms.

Conclusion:

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 7– Token Based Algorithm

Learning Objective: Student should be able to design a program to illustrate token based algorithm.

Tools: Python

Theory:

Mutual Exclusion

There are several resources in a system that must not be used simultaneously by multiple processes if program operation is to be correct. For example, a file must not be simultaneously updated by multiple processes. Similarly, use of unit record peripherals such as tape drives or printers must be restricted to single process at a time. Therefore, exclusive access to such a. shared resource by a process must be ensured. This exclusiveness of access is called mutual exclusion between processes. The sections of a program that need exclusive access to shared resources are referred to as critical sections. For mutual exclusion, means are introduced to prevent processes from executing concurrently within their associated critical sections.
An algorithm for implementing mutual exclusion must satisfy the following requirements:

1. Mutual exclusion. Given a shared resource accessed by multiple concurrent processes, at any time only one process should access the resource. That is, a process that has been granted the resource must release it before it can be granted to another process.

2. No starvation. If every process that is granted the resource eventually releases it, every request must be eventually granted.

In single-processor systems, mutual exclusion is implemented using semaphores, monitors, and similar constructs.

Suzuki–Kasami Algorithm

Suzuki–Kasami algorithm is a token-based algorithm for achieving mutual exclusion in distributed systems.This is modification of Ricart–Agrawala algorithm, a permission based (Non-token based) algorithm which uses **REQUEST** and **REPLY** messages to ensure mutual exclusion.

In token-based algorithms, A site is allowed to enter its critical section if it possesses the unique token. Non-token based algorithms uses timestamp to order requests for the critical section where as sequence number is used in token based algorithms.

Data structure and Notations:

- An array of integers **RN[1…N]**

  A site Si keeps **RNi[1…N]**, where **RNi[j]** is the largest sequence number received so far through **REQUEST** message from site Si.

- An array of integer **LN[1…N]**

  This array is used by the token.**LN[J]** is the sequence number of the request that is recently executed by site Sj.

- A queue **Q**
  This data structure is used by the token to keep a record of ID of sites waiting for the token

Algorithm:

- **To enter Critical section:**
  - When a site Si wants to enter the critical section and it does not have the token then it increments its sequence number **RNi[i]** and sends a request message **REQUEST(i, sn)** to all other sites in order to request the token. Here **sn** is update value of **RNi[i]**
  - When a site Sj receives the request message **REQUEST(i, sn)** from site Si, it sets **RNj[i]** to maximum of **RNj[i]** and **sn** i.e **RNj[i]** = max(**RNj[i]**, **sn**).
  - After updating **RNj[i]**, Site Sj sends the token to site Si if it has token and **RNj[i]** = **LN[i]** + 1
- **To execute the critical section:**
  - Site Si executes the critical section if it has acquired the token.
- **To release the critical section:**

  After finishing the execution Site Si exits the critical section and does following:
  - sets **LN[i]** = **RNi[i]** to indicate that its critical section request **RNi[i]** has been executed
  - For every site Sj, whose ID is not prsent in the token queue **Q**, it appends its ID to **Q** if **RNi[j]** = **LN[j]** + 1 to indicate that site Sj has an outstanding request.

- After above updation, if the Queue **Q** is non-empty, it pops a site ID from the **Q** and sends the token to site indicated by popped ID.
- If the queue **Q** is empty, it keeps the token

Message Complexity:

The algorithm requires 0 message invocation if the site already holds the idle token at the time of critical section request or maximum of N message per critical section execution. This N messages involves

- (N – 1) request messages
- 1 reply message

Drawbacks of Suzuki–Kasami Algorithm:

- **Non-symmetric Algorithm:** A site retains the token even if it does not have requested for critical section. According to definition of symmetric algorithm "No site possesses the right to access its critical section when it has not been requested."

Exercise:

1. **Explain Suzuki Kasami Algorithm with suitable example?**

Concept

The Suzuki-Kasami algorithm is a token-based approach to ensure that only one process can access a shared resource (critical section) at a time. It achieves this by using a single token that is passed around among processes. The process holding the token has exclusive access to the critical section.

Algorithm Steps

Initialization: Each process maintains two data structures:

RN[i]: Request number of process i, initially 0.
LN[i]: Local copy of the last received token number for process i, initially 0.
TokenQueue: A queue to store process IDs waiting for the token (initially empty).
Entering Critical Section:

If a process i wants to enter the critical section and doesn't have the token:
Increment RN[i].
Broadcast a REQUEST(i, RN[i]) message to all other processes.
Receiving Request Message:

When a process j receives a REQUEST(i, sn) message:

Update RN[j][i] = max(RN[j][i], sn). This ensures j considers the latest request from i.
Sending the Token:

When a process j receives the token:
If j doesn't have any pending requests (i.e., TokenQueue is empty):
Set LN[j] = RN[j][j] + 1. This indicates j has processed its own requests.
If TokenQueue is not empty and RN[j][TokenQueue.front()] = LN[j] + 1:
Dequeue the process ID from TokenQueue and send the token to that process.
Otherwise, keep the token and update TokenQueue if necessary (explained in step 5).
Handling Concurrent Requests:

If multiple processes request the token concurrently, the process with the highest RN[i] value gets priority (similar to timestamp-based algorithms).
To handle this, TokenQueue is used. When j receives a request message and RN[j][i] is greater than the current value in TokenQueue for process i, it updates TokenQueue with i (ensuring higher priority requests are served first).

CODE:

Result and Discussion:

The Suzuki-Kasami token algorithm is a well-established approach for ensuring mutual exclusion in distributed systems. It leverages a single token that is passed among processes. Only the process holding the token can access a shared resource (critical section). The algorithm effectively coordinates requests using request numbers and a token queue, prioritizing processes based on these values. This mechanism guarantees that only one process can be in the critical section at a time, preventing conflicts when accessing shared resources.

Code:

```
import random, sys, threading, time, datetime
from collections import deque
from mpi4py import MPI

now = datetime.datetime.now
comm = MPI.COMM_WORLD
tid = comm.Get_rank()
N = comm.Get_size()

criticalSectionLock = threading.Lock()
tokenLock = threading.Lock()
RNLock = threading.Lock()
releaseLock = threading.Lock()
requestLock = threading.Lock()
sendLock = threading.Lock()

Q = deque()
hasToken = 0
inCriticalSection = 0
waitingForToken = 0

RN = []
```

```
LN = []
for i in range(0, N):
    LN.append(0)
    RN.append(0)

# Process 0 gets the token initially
if(tid == 0):
    print(f"{now().strftime('%M:%S')}|[Process {tid}]: Startup Token")
    sys.stdout.flush()
    hasToken = 1
RN[0] = 1


def receiveRequest():
    global LN
    global RN
    global Q
    global inCriticalSection
    global waitingForToken
    global hasToken, tid
    while True:
        message = comm.recv(source=MPI.ANY_SOURCE)
        if message[0] == 'RN':
            with RNLock:
                requester_id = message[1]
                cs_value = message[2]
                RN[requester_id] = max([cs_value, RN[requester_id]])
                if cs_value < RN[requester_id]:
                    print(f"{now().strftime('%M:%S')}|[Process {tid}]: Request from Process {requester_id} expired")
                    sys.stdout.flush()

                if (hasToken == 1) and (inCriticalSection == 0) and (RN[requester_id] == (LN[requester_id] + 1)):
                    hasToken = 0
                    sendToken(requester_id)

        elif message[0] == 'token':
            with tokenLock:
                print(f"{now().strftime('%M:%S')}|[Process {tid}]: Got token")
                sys.stdout.flush()
                hasToken = 1
                waitingForToken = 0
                LN = message[1]
                Q = message[2]
                criticalSection()


def sendRequest(message):
    for i in range(N):
        if tid != i:
            to_send = ['RN', tid, message]
            comm.send(to_send, dest=i)


def sendToken(recipent):
    global Q
    with sendLock:
        print(f"{now().strftime('%M:%S')}|[Process {tid}]: Sending token to Process {recipent}")
        sys.stdout.flush()
        global inCriticalSection
        to_send = ['token', LN, Q]
        comm.send(to_send, dest=recipent)


def requestForCriticalSection():
```

```
    global RN
    global inCriticalSection
    global waitingForToken
    global hasToken
    with requestLock:
        if hasToken == 0:
            RN[tid] += 1
            print(f"{now().strftime('%M:%S')}|[Process {tid}]: Request for token ({RN[tid]})")
            sys.stdout.flush()
            waitingForToken = 1
            sendRequest(RN[tid])


def release_cs():
    global inCriticalSection
    global LN
    global RN
    global Q
    global hasToken
    with releaseLock:
        LN[tid] = max(LN) + 1
        for k in range(N):
            if k not in Q:
                if RN[k] == (LN[k] + 1):
                    Q.append(k)
                    print(f"{now().strftime('%M:%S')}|[Process {tid}]: Adding {k} to Queue\n\tQueue after adding: {Q}\n\tLN={LN}\n\tRN={RN}")
                    sys.stdout.flush()
        if len(Q) != 0:
            hasToken = 0
            sendToken(Q.popleft())


def criticalSection():
    global inCriticalSection
    global hasToken
    with criticalSectionLock:
        if hasToken == 1:
            inCriticalSection = 1
            print(f"{now().strftime('%M:%S')}|[Process {tid}]: Entering Critical Section ({RN[tid]})")
            sys.stdout.flush()
            time.sleep(random.uniform(2, 5))
            print(f"{now().strftime('%M:%S')}|[Process {tid}]: Exiting Critical Section ({RN[tid]})")
            sys.stdout.flush()
            inCriticalSection = 0
            release_cs()


try:
    listener = threading.Thread(target=receiveRequest)
    listener.start()
except:
    print("ERROR: Threads not spawning")
    sys.stdout.flush()

while True:
    if hasToken == 0:
        time.sleep(random.uniform(1, 3))
        requestForCriticalSection()
    elif inCriticalSection == 0:
        criticalSection()
    while waitingForToken:
        time.sleep(0.5)
```

```
01:49|[Process 0]: Exiting Critical Section (1)
01:49|[Process 0]: Entering Critical Section (1)
01:53|[Process 0]: Exiting Critical Section (1)
01:53|[Process 0]: Entering Critical Section (1)
01:57|[Process 0]: Exiting Critical Section (1)
01:57|[Process 0]: Entering Critical Section (1)
02:02|[Process 0]: Exiting Critical Section (1)
02:02|[Process 0]: Entering Critical Section (1)
02:05|[Process 0]: Exiting Critical Section (1)
02:05|[Process 0]: Entering Critical Section (1)
02:08|[Process 0]: Exiting Critical Section (1)
02:08|[Process 0]: Entering Critical Section (1)
02:11|[Process 0]: Exiting Critical Section (1)
02:11|[Process 0]: Entering Critical Section (1)
```

```
01:49|[Process 0]: Entering Critical Section (1)
01:52|[Process 0]: Exiting Critical Section (1)
01:52|[Process 0]: Entering Critical Section (1)
01:57|[Process 0]: Exiting Critical Section (1)
01:57|[Process 0]: Entering Critical Section (1)
01:59|[Process 0]: Exiting Critical Section (1)
01:59|[Process 0]: Entering Critical Section (1)
02:01|[Process 0]: Exiting Critical Section (1)
02:01|[Process 0]: Entering Critical Section (1)
02:04|[Process 0]: Exiting Critical Section (1)
02:04|[Process 0]: Entering Critical Section (1)
02:09|[Process 0]: Exiting Critical Section (1)
02:09|[Process 0]: Entering Critical Section (1)
02:11|[Process 0]: Exiting Critical Section (1)
02:11|[Process 0]: Entering Critical Section (1)
```

```
01:46|[Process 0]: Entering Critical Section (1)
01:50|[Process 0]: Exiting Critical Section (1)
01:50|[Process 0]: Entering Critical Section (1)
01:54|[Process 0]: Exiting Critical Section (1)
01:54|[Process 0]: Entering Critical Section (1)
01:57|[Process 0]: Exiting Critical Section (1)
01:57|[Process 0]: Entering Critical Section (1)
02:02|[Process 0]: Exiting Critical Section (1)
02:02|[Process 0]: Entering Critical Section (1)
02:06|[Process 0]: Exiting Critical Section (1)
02:06|[Process 0]: Entering Critical Section (1)
02:10|[Process 0]: Exiting Critical Section (1)
02:10|[Process 0]: Entering Critical Section (1)
```

Learning Outcomes: The student should have the ability to

LO1: Recall the different token based algorithm.

LO2: Apply the different token based algorithm.

Course Outcomes: Upon completion of the course students will be able to understand token based Algorithm.

Conclusion:

Viva Questions:

1. What is Mutual Exclusion?
2. Explain Token based algorithm?
3. What are the Types of Token based Algorithm?
4. Explain Suzuki Kasami Algorithm?
5. Explain Raymond Tree Based Algorithm?
6. Explain Singhal Heuristics Algorithm?

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 8– Non Token Based Algorithm

Learning Objective: Student should be able to design a program to illustrate non token based algorithm.

Tools : Java

Theory:

Non-token based approach:

- A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
- This approach use timestamps instead of sequence number to order requests for the critical section.
- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme

Example:

- Lamport's algorithm, Ricart–Agrawala algorithm

Ricart–Agrawala algorithm

Ricart–Agrawala algorithm is an algorithm to for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala. This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm. Like Lamport's Algorithm, it also follows permission based approach to ensure mutual exclusion.

In this algorithm:

- Two type of messages (**REQUEST** and **REPLY**) are used and communication channels are assumed to follow FIFO order.
- A site send a **REQUEST** message to all other site to get their permission to enter critical section.
- A site send a **REPLY** message to other site to give its permission to enter the critical section.
- A timestamp is given to each critical section request using Lamport's logical clock.
- Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.

- **To enter Critical section:**
  - When a site Si wants to enter the critical section, it send a timestamped **REQUEST** message to all other sites.
  - When a site Sj receives a **REQUEST** message from site Si, It sends a **REPLY** message to site Si if and only if
    - Site Sj is neither requesting nor currently executing the critical section.
    - In case Site Sj is requesting, the timestamp of Site Si's request is smaller than

its own request.
    - Otherwise the request is deferred by site Sj.
- **To execute the critical section:**
    - Site Si enters the critical section if it has received the **REPLY** message from all other sites.
- **To release the critical section:**
    - Upon exiting site Si sends **REPLY** message to all the deferred requests.

Message Complexity:
Ricart–Agrawala algorithm requires invocation of 2(N – 1) messages per critical section execution. These 2(N – 1) messages involves

- (N – 1) request messages
- (N – 1) reply messages

Drawbacks of Ricart–Agrawala algorithm:

- **Unreliable approach:** failure of any one of node in the system can halt the progress of the system. In this situation, the process will starve forever.
  The problem of failure of node can be solved by detecting failure after some timeout.

Exercise:

1. **Explain any non token based algorithm with suitable example?**

Ricart–Agrawala algorithm is an algorithm for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala. This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm. Like Lamport's Algorithm, it also follows permission-based approach to ensure mutual exclusion. In this algorithm:

Two type of messages (REQUEST and REPLY) are used and communication channels are assumed to follow FIFO order.
A site send a REQUEST message to all other site to get their permission to enter the critical section.
A site send a REPLY message to another site to give its permission to enter the critical section.
A timestamp is given to each critical section request using Lamport's logical clock.
Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.
Algorithm:

To enter Critical section:
When a site Si wants to enter the critical section, it send a timestamped REQUEST

message to all other sites.
When a site Sj receives a REQUEST message from site Si, It sends a REPLY message to site Si if and only if
Site Sj is neither requesting nor currently executing the critical section.
In case Site Sj is requesting, the timestamp of Site Si's request is smaller than its own request.
To execute the critical section:
Site Si enters the critical section if it has received the REPLY message from all other sites.
To release the critical section:
Upon exiting site Si sends REPLY message to all the deferred requests.
Message Complexity: Ricart–Agrawala algorithm requires invocation of 2(N – 1) messages per critical section execution. These 2(N – 1) messages involves

(N – 1) request messages
(N – 1) reply messages
Advantages of the Ricart-Agrawala Algorithm:

Low message complexity: The algorithm has a low message complexity as it requires only (N-1) messages to enter the critical section, where N is the total number of nodes in the system.
Scalability: The algorithm is scalable and can be used in systems with a large number of nodes.
Non-blocking: The algorithm is non-blocking, which means that a node can continue executing its normal operations while waiting to enter the critical section.
Drawbacks of Ricart–Agrawala algorithm:

Unreliable approach: failure of any one of node in the system can halt the progress of the system. In this situation, the process will starve forever. The problem of failure of node can be solved by detecting failure after some timeout.
Performance:

Synchronization delay is equal to maximum message transmission time
It requires 2(N – 1) messages per Critical section execution

Code:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.net.Socket;
import java.sql.Timestamp;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
```

```
import java.util.Map;


public class RicartAgrawala {

	private static boolean requestingCS;
	private static Timestamp myTimeStamp;
//	MessageSender msgSender=new MessageSender();
	MessageSender msgSender;
	Timestamp guestTimestamp;
	static int permitreplies;
	static int cscount;
	static int noofrequests;
	static int id;

//	PriorityQueue<Timestamp> pq = new PriorityQueue<Timestamp>();
//	PriorityQ pq = new PriorityQ<>();


	public static Timestamp getMyTimeStamp() {
		return myTimeStamp;
	}

	public static void setMyTimeStamp(Timestamp myTimeStamp) {
		RicartAgrawala.myTimeStamp = myTimeStamp;
	}

	public synchronized static boolean isRequestingCS() {
		return requestingCS;
	}

	public synchronized static void setRequestingCS(boolean requestingCS) {
		RicartAgrawala.requestingCS = requestingCS;
	}

	public synchronized void determineCriticalSectionEntry(Socket s,List message,int id,int guestid){
		this.id=id;
		this.guestTimestamp = Timestamp.valueOf((String)message.get(1));
//		if I'm requesting for CS,let me compare timestamps to determine if I should permit the host or
defer the reply
		if(RicartAgrawala.isRequestingCS()){
			msgSender = new MessageSender();
			if(myTimeStamp.compareTo(guestTimestamp)>0){  //guest timestamp has higher priority
				msgSender.sendMessage(s, id, Message.PERMIT);
//				System.out.println("--MyTimeStamp :   "+myTimeStamp);
//				System.out.println("--Requesting TS : "+guestTimestamp);
//				System.out.println("--                PERMIT sent*");

			}
			if(myTimeStamp.compareTo(guestTimestamp)<0){ //my timestamp has higher priority
				DeferredRequests.add(guestid, s);
				System.out.println("<<<<< Request deferred for "+guestid);
//				System.out.println("MyTimeStamp :   "+myTimeStamp);
//				System.out.println("Requesting TS : "+guestTimestamp);
//				System.out.println(".....................Reply has been deffered...");

			}
			if(myTimeStamp.compareTo(guestTimestamp)==0){
```

```
//                              System.out.println("[[[[[[[[[[[[  we are into a complex situation ]]]]]]]] "+id);
//                              System.out.println("]]]]]]      Guest id : "+guestid);
                                if(guestid<id){
                                        msgSender.sendMessage(s, id, Message.PERMIT);
                                        System.out.println("########              PERMIT sent*");
                                }else{
                                        DeferredRequests.add(guestid, s);
//                              System.out.println(".....Reply has been deffered...");
                                }
                        }
                }else{
                        msgSender = new MessageSender();
                        msgSender.sendMessage(s, id, Message.PERMIT);

                }
        }

//      public void printQSize(){
//              System.out.println("######################## Queue size is : "+pqueue.getSize());
//      }

        public synchronized void criticalSection(){
//              System.out.println("Time entered C*S : "+TimeStamp.getStartTime());
                System.out.println("---------------------------------------------- Time entered
"+TimeStamp.getTime());
//              System.out.println(" ----- Time elapsed to request and enter : "+timeElapsed);
                if(true){
                        System.out.println("******** Entered critical section!!!!!! ********* "+ ++cscount);
                        try {
                                Thread.sleep(20);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
                writeToFile();
                exitCriticalSection();

        }


        public static void exitCriticalSection(){
                CriticalSectionRequests csr = new CriticalSectionRequests();
                RicartAgrawala.setRequestingCS(false);
                TimeStamp.setInstancetoNull();
                sendReplyToDeferredRequests();
//              System.out.println("Time exited C*S : "+TimeStamp.getEndTime());
                long timeElapsed=TimeStamp.getEndTime()-TimeStamp.getStartTime();
                System.out.println("---------------------------------------------- Time exited
"+TimeStamp.getTime());
                ++noofrequests;
                if(noofrequests<20){
                        csr.sendCSRequests(id);
                }
        }

        public static void sendReplyToDeferredRequests(){
                MessageSender msgSender=new MessageSender();
```

```
                Iterator it = DeferredRequests.deferredlist.entrySet().iterator();
                while(it.hasNext()){
//                      System.out.println("Iterating deffered list.....");
                        if(!isRequestingCS()){
//                              System.out.println("Actually sending permits to deferred requests...!!!!!!!!");
                        Map.Entry pairs = (Map.Entry) it.next();
//                      msgSender.sendMessage((Socket)it.next(), 1, Message.PERMIT);
//                      DeferredRequests.toRemove.add(pairs.getKey());
                        System.out.println(">>>> Sent deferred PERMIT to : "+pairs.getKey());
                        msgSender.sendMessage((Socket)pairs.getValue(), id, Message.PERMIT);
//                      System.out.println("***************PERMIT REPLY SENT**************** "+
++permitreplies);
//                      it.remove();
                        }
                }
                DeferredRequests.deferredlist = new HashMap<Integer, Socket>();
                }


        public void writeToFile() {
                try {
                        File file = new File("/home/rohit/Desktop/ricartoutput.txt");
                        if (!file.exists()) {
                                file.createNewFile();
                        }
                        FileWriter fw = new FileWriter(file.getAbsoluteFile(), true);
                        BufferedWriter bw = new BufferedWriter(fw);

                        synchronized (this) {
                                bw.write("Node no. "+id+ " Entered critical section..." + "\n"); // write to file
                        }
                        bw.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

```
------------------------------------------------ Time entered 2024-04-01 10:27:IST
******** Entered critical section!!!!!! ******** 1
... (critical section code execution) ...
------------------------------------------------ Time exited 2024-04-01 10:27:IST
<<<<< Request deferred for 3
>>> Sent deferred PERMIT to : 3

------------------------------------------------ Time entered 2024-04-01 10:27:IST
******** Entered critical section!!!!!! ******** 2
... (critical section code execution) ...
------------------------------------------------ Time exited 2024-04-01 10:27:IST

PS C:\Users\gamev>
```

Result and Discussion:

The Ricart-Agrawala algorithm successfully achieves mutual exclusion in distributed systems without relying on a central token. Here are some key results:

Mutual Exclusion: It guarantees that only one process can be in the critical section at a time, preventing conflicts when accessing shared resources.
Scalability: The algorithm works well with a large number of processes due to its message complexity being proportional to the number of processes minus one (N-1).
Non-Blocking: Processes can continue their regular operations while waiting for permission to enter the critical section, improving overall system performance.
Low Message Complexity: Compared to some other algorithms, Ricart-Agrawala requires fewer messages per critical section access.
However, there are also some limitations to consider:

Deadlock: In rare cases, deadlock can occur if processes are waiting for permission from each other indefinitely. This can be mitigated by detecting and handling failed processes.
Overhead: Compared to simpler token-based algorithms, there might be more message exchanges for coordination.
Overall, the Ricart-Agrawala algorithm offers a good balance between efficiency and functionality for mutual exclusion in distributed systems.

Learning Outcomes: The student should have the ability to

LO1: Recall the non token based algorithm.

LO2: Analyze the different non token based algorithm

Course Outcomes: Upon completion of the course students will be able to understand non token based Algorithm.

Conclusion:

Viva Questions:

1. What is Mutual Exclusion?
2. Explain Non Token based algorithm?
3. What are the Types of Non Token based Algorithm?
4. Explain Lamport Algorithm?
5. Explain Ricart Agrawala‘s Algorithm?
6. Explain Maekawa‘s Algorithm?

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 9– Load Balancing Algorithm

Learning Objective: Student should be able to develop a program for Load Balancing Algorithm.

Tools : Java

Theory:

Balancing Algorithm:

The scheduling algorithms using this approach are known as load-balancing algorithms or load-leveling algorithms. These algorithms arc based on the intuition that, for better resource utilization, it is desirable for the load in a distributed system to be balanced evenly. Thus, a load-balancing algorithm tries to balance the total system load by transparently transferring the workload from heavily loaded nodes to lightly loaded nodes in an attempt to ensure good overall performance relative to some specific metric of system performance. When considering performance from the user point of view, the metric involved is often the response time of the processes. However, when performance is considered from the resource point of view, the metric involved is the total system throughput. In contrast to response time, throughput is concerned with seeing that all users are treated fairly and that all are making progress. Notice that the resource view of maximizing resource utilization is compatible with the desire to maximize system throughput. Thus the basic goal of almost all the load- balancing algorithms is to maximize the total system throughput.

Static versus Dynamic

At the highest level, we may distinguish between static and dynamic load-balancing algorithms. Static algorithms use only information about the average behavior of the system, ignoring the current state of the system. On the other hand, dynamic algorithms react to the system state that changes dynamically.

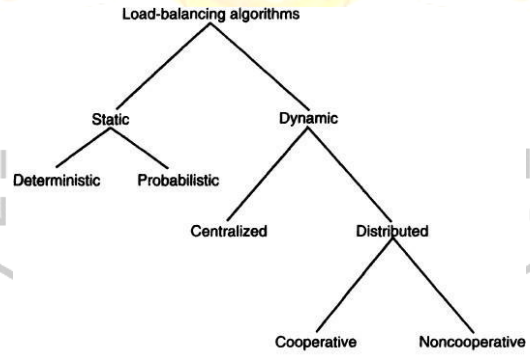

Fig. 7.3 A taxonomy of load-balancing algorithms.

Deterministic versus Probabilistic

Static load-balancing algorithms may be either deterministic or probabilistic. Deterministic algorithms use the information about the properties of the nodes and the characteristics of the processes to be scheduled to deterministically allocate processes to nodes. Notice that the task

assignment algorithms basically belong to the category of deterministic static load-balancing algorithms.

Centralized versus Distributed

Dynamic scheduling algorithms may be centralized or distributed. In a centralized dynamic scheduling algorithm, the responsibility of scheduling physically resides on a single node. On the other hand, in a distributed dynamic scheduling algorithm, the work involved in making process assignment decisions is physically distributed among the various nodes of the system.

In the centralized approach, the system state information is collected at a single node at which all scheduling decisions are made. This node is called the centralized server node. All requests for process scheduling are handled by the centralized server, which decides about the placement of a new process using the state information stored in it. The centralized approach can efficiently make process assignment decisions because the centralized server knows both the load at each node and the number of processes needing service. In the basic method, the other nodes periodically send status update messages to the central server node. These messages are used to keep the system state information up to date at the centralized server node. One might consider having the centralized server query the other nodes for state information. This would reduce message traffic if state information was used to answer several process assignment requests, but since nodes can change their load any time due to local activities, this would introduce problems of stale state information.

Cooperative versus Noncooperative

Distributed dynamic scheduling algorithms may be categorized as cooperative and noncooperative. In noncooperative algorithms, individual entities act as autonomous entities and make scheduling decisions independently of the actions of other entities. On the other hand, in cooperative algorithms, the distributed entities cooperate with each other to make scheduling decisions. Hence, cooperative algorithms are more complex and involve larger overhead than noncooperative ones. However, the stability of a cooperative algorithm is better than that of a noncooperative algorithm.

Code

Server.java

```
public class Server {
    private String host;
    private int port;
    // ...

    public Server(String host, int port) {
        this.host = host;
        this.port = port;
    }
    public String getHost() {
        return host;
    }
```

```
    public int getPort() {
        return port;
    }
}
```

Loadbalancer.java

```
import java.util.List;

public class LoadBalancer {

    private List<Server> servers;
    private LoadBalanceStrategy strategy; // Interface for different
balancing algorithms

    public LoadBalancer(List<Server> servers, LoadBalanceStrategy
strategy) {
        this.servers = servers;
        this.strategy = strategy;
    }

    public Server chooseServer() {
        return strategy.chooseServer(servers);
    }

    public String handleRequest(String request) {
        Server server = chooseServer();
        // Simulate sending request to server
        return "Request handled by server: " + server.getHost() + ":"
+ server.getPort();
    }
}
```

Loadbalancestratergy.java

```
import java.util.List;
public interface LoadBalanceStrategy {
    Server chooseServer(List<Server> servers);
}
```

roundrobin.java

```
import java.util.List;
public class roundrobin implements LoadBalanceStrategy {

    private int currentIndex = 0;

    @Override
    public Server chooseServer(List<Server> servers) {

        Server server = servers.get(currentIndex);
        currentIndex = (currentIndex + 1) % servers.size();
        return server;
    }
}
```

Myloadbalancer.java

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

public class MyLoadBalancer {

    public static void main(String[] args) {
        List<Server> servers = new ArrayList<>();
        servers.add(new Server("server1", 8080));
        servers.add(new Server("server2", 8081));

        LoadBalanceStrategy strategy = new roundrobin();
        LoadBalancer balancer = new LoadBalancer(servers, strategy);

        String response = balancer.handleRequest("Sample Request");
        System.out.println(response);
    }
}
```

Result and Discussion:

```
PS C:\Users\nambi> javac MyLoadBalancer.java LoadBalanceStrategy.java Server.java roundrobin.java
PS C:\Users\nambi> java MyLoadBalancer
Request handled by server: server1:8080
```

Learning Outcomes: The student should have the ability to

LO1: Comprehend the Load balancing concept

LO2: Analyze different that load balancing methods

Course Outcomes: Upon completion of the course students will be able to understand Load Balancing Algorithm.

Conclusion:

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				