Experiment No. 07

Semester	B.E. Semester VIII – Computer Engineering
Subject	Distributed Computing Lab
Subject Professor In-charge	Dr. Umesh Kulkarni
Assisting Professor	Prof. Prakash Parmar
Academic Year	2024-25

Student Name	Deep Salunkhe
Roll Number	21102A0014

Title: Mutual Exclusion in Distributed Systems

Introduction

Mutual exclusion is a fundamental problem in distributed computing that ensures only one process accesses a critical section at a time, preventing conflicts and inconsistencies. This report discusses the implementation of mutual exclusion using the Ricart-Agrawala algorithm in a distributed system.

Concept of Mutual Exclusion

In a distributed system, multiple processes execute independently, and a shared resource may require exclusive access to avoid data inconsistency. Mutual exclusion mechanisms ensure that at any given time, only one process can enter its critical section, maintaining consistency and synchronization.

Ricart-Agrawala Algorithm

The Ricart-Agrawala algorithm is a message-passing based approach for achieving mutual exclusion in distributed systems. It eliminates the need for a central coordinator by using a fully distributed approach, where nodes communicate with each other to request and grant access to the critical section.

Working Principle

1. **Requesting the Critical Section**: When a process wants to enter the critical section, it sends a request message to all other processes.

2. **Receiving Requests**: Other processes respond with a reply message if they are not currently in the critical section or if they have a lower priority request.

3. **Entering the Critical Section**: The process enters the critical section only after receiving replies from all other processes.

4. **Releasing the Critical Section**: After execution, the process sends reply messages to all queued requests, allowing other processes to proceed.

Advantages of Ricart-Agrawala Algorithm

- **Fully distributed**: No central coordinator, reducing the risk of single points of failure.
- **Less message overhead**: Requires only (N-1) messages per request in contrast to token-based approaches.
- **Fairness**: Requests are handled based on timestamps, ensuring fairness in granting access.

Challenges and Considerations

- **Message Delays**: Network latency can affect response times and access to the critical section.
- **Node Failures**: If a process crashes before replying, other processes may get blocked.
- **Scalability**: As the number of processes increases, message complexity also increases.

Applications of Mutual Exclusion in Distributed Systems

- **File locking in distributed databases**
- **Concurrent access control in cloud computing**
- **Resource allocation in cluster computing**
- **Process synchronization in operating systems**

Conclusion

Mutual exclusion is essential in distributed computing to maintain consistency and prevent race conditions. The Ricart-Agrawala algorithm provides an efficient, distributed solution by utilizing message-passing techniques to synchronize access to shared resources. Understanding and implementing such algorithms is crucial for developing reliable distributed systems.

Code:

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

class MutualExclusionNode {
    private int nodeId;
    private int totalNodes;
    private boolean requestingCS;
    private Set<Integer> repliesReceived;
    private ServerSocket serverSocket;
    private List<Integer> otherNodes;

    public MutualExclusionNode(int nodeId, int totalNodes) throws IOException {
        this.nodeId = nodeId;
        this.totalNodes = totalNodes;
        this.requestingCS = false;
        this.repliesReceived = new HashSet<>();
        this.otherNodes = new ArrayList<>();
        this.serverSocket = new ServerSocket(5000 + nodeId);

        for (int i = 0; i < totalNodes; i++) {
            if (i != nodeId) {
                otherNodes.add(i);
            }
        }
    }

    public void startListener() {
        new Thread(() -> {
            try {
                while (true) {
                    Socket socket = serverSocket.accept();
                    BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
                    String message = in.readLine();
                    handleMessage(message);
                    socket.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }).start();
    }

    private void handleMessage(String message) {
        String[] parts = message.split(" ");
```

```
        String type = parts[0];
        int senderId = Integer.parseInt(parts[1]);

        if (type.equals("REQUEST")) {
            sendReply(senderId);
        } else if (type.equals("REPLY")) {
            repliesReceived.add(senderId);
        }
    }

    private void sendReply(int targetId) {
        sendMessage(targetId, "REPLY " + nodeId);
    }

    private void sendMessage(int targetId, String message) {
        try {
            Socket socket = new Socket("localhost", 5000 + targetId);
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            out.println(message);
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void requestCriticalSection() {
        requestingCS = true;
        repliesReceived.clear();

        for (int node : otherNodes) {
            sendMessage(node, "REQUEST " + nodeId);
        }

        while (repliesReceived.size() < totalNodes - 1) {
            // Waiting for replies
        }

        enterCriticalSection();
    }

    private void enterCriticalSection() {
        System.out.println("Node " + nodeId + " is entering the critical section.");
        try {
            Thread.sleep(2000); // Simulate work in the critical section
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Node " + nodeId + " is leaving the critical section.");
```

```
        requestingCS = false;
    }

    public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            System.out.println("Usage: java MutualExclusionNode <nodeId>
<totalNodes>");
            return;
        }

        int nodeId = Integer.parseInt(args[0]);
        int totalNodes = Integer.parseInt(args[1]);

        MutualExclusionNode node = new MutualExclusionNode(nodeId, totalNodes);
        node.startListener();

        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("Press Enter to request critical section...");
            scanner.nextLine();
            node.requestCriticalSection();
        }
    }
}
```

Output: