Assignment 2

1. Problem Statement

1.1. Develop a program to implement Ricart Agrawala algorithm.

2. Source Code

2.1. >> Main.java

```
// Here we implement Ricart–Agrawala Algorithm in Mutual Exclusion in
Distributed System
// with Lamport's logical clock model using Thread

// Here Threads are shown as different system which are distributed and can request
for critical state
//without any external interference.


// Input File Format
/* node1,node2,node3,node4 */
//or
/* node1,node2
   node3,node4
 */

import java.awt.*;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class Main {
   public static void main(String[] args) {
      List<String> nodeNames = new ArrayList<>();  // List to store node names
from file
      int id = 1; // ID of the first node
      String filename = openFileDialog();  // Get file name from gui component
      if (filename == null) {
         System.out.println("No file selected.");
         return;
      }
```

```
        try {
            Scanner scanner = new Scanner(new File(filename));
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                String[] tokens = line.split(",");  // Split line by commas
                nodeNames.addAll(List.of(tokens)); // Store node name
            }
        } catch (FileNotFoundException e) {
            System.out.println("Error: File not found - nodes.txt");
            // Handle the exception if the file is not found
        }
        Message messageSystem = new Message(nodeNames); // Message system
which handel the message passing

        List<Node> nodes = new ArrayList<>(); // Store node as a Node component

        for (String nodeName : nodeNames) {
            nodes.add(new Node(String.valueOf(id), nodeName, messageSystem,
nodes));
            id++;
        }

        for (Node node : nodes) {
            new Thread(node).start(); // start each node
        }
    }

    private static String openFileDialog() {
        FileDialog fd = new FileDialog((Frame) null, "Open", FileDialog.LOAD);
        fd.setVisible(true);
        String filename = fd.getFile();
        // Validate file extension
        if (filename != null) {
            return fd.getDirectory() + filename;
        }
        return null;
    }

}
```

2.2. >> Node.java

```
import java.util.List;
import java.util.Random;
import java.util.Set;
import java.util.concurrent.CopyOnWriteArraySet;

public class Node implements Runnable {
    // Objects from another class
    private final Message messageSystem;
    private final Random random = new Random();
    private Status status = Status.None;
    private MessageType messageType;

    // Used Variables
    private String id; //Store ID of a node
    private String name; // Store name of a node
    private String sMessage; // Message to be sent
    private int logicalClock = 0; // logical clock
    private int clockAfterCriticalState = 0; // Clock after critical state
    private int clockInCriticalState = 0; // clock in critical state
    private boolean isRequested = false;  // If a node  requested for critical state or not
    private int requestLogicalClock; // Timestamp while requesting for critical state

    //Used Structures
    private List<Node> nodes;       // Information of all the node in the system
    private Set<String> AcceptedCriticalState = new CopyOnWriteArraySet<>();  // To
track who accepted our critical state request
    private Set<String> DeferredWhileCriticalState = new CopyOnWriteArraySet<>(); //
To track requests while in critical state

    public Node(String id, String name, Message messageSystem, List<Node> nodes) {
        this.id = id;
        this.name = name;
        this.messageSystem = messageSystem;
        this.nodes = nodes;
    }

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            //After Node exited critical state and can request again for critical state
```

```
            if (clockAfterCriticalState == 0) {
                if (status != Status.Requesting) {
                    status = Status.None;
                }
            }
            setStatus(); // To set the status of node
            checkClock(); // To check the different clocks
            sendRequestMessage(); // Sending request critical state to everyone
            String receivedMessage = messageSystem.getMessage(name); // Receive
message from other nodes
            if (receivedMessage == null) {
                System.out.println("Received Message [ " + status + " ] :: Name: " + name + "
Received Message: No message to receive");
                // If no message is received but some already requested for critical state
earlier
                if (status == Status.AfterCriticalState) {
                    if (!DeferredWhileCriticalState.isEmpty()) {
                        System.out.println(name + " exiting critical state.");
                        sendGO_AHEADMessage(); // Send those nodes GO_AHEAD
                        DeferredWhileCriticalState.clear(); // Clear the deferred List while is
populated while in critical state
                    }
                }
            } else { // If some message is received
                logicalClock++; // Increments logical clock
                String receivedMessageParts[] = receivedMessage.split(":");
                System.out.println("Received Message[ " + status + " ]:: Name: " + name + "
Received Message: " + receivedMessage +
                        " Received from : " + receivedMessageParts[0]);
                System.out.println();
                if (receivedMessageParts[2].equalsIgnoreCase("GO_AHEAD")) {
                    // If received message is GO_AHEAD them that node is permitting this
node to go in critical state
                    AcceptedCriticalState.add(receivedMessageParts[0]);
                }

                if (AcceptedCriticalState.size() == nodes.size() - 1) {
                    //If all other nodes are agree for critical state
                    System.out.println(name + " is in Critical State");
                    status = Status.InCriticalState; // Changing state to critical state
                    clockInCriticalState = random.nextInt((5 - 1) + 1) + 1; // Stay in critical
state for 1 tick to 5 tick
                    clockAfterCriticalState = clockInCriticalState + 4; // can't request to be in
critical state for next 10 more ticks
```

```
                AcceptedCriticalState.clear(); // Remove everyone who accepted request
            }
            // Behavior of a node according to it's status
            if (status == Status.None || status == Status.AfterCriticalState) {
                messageType = MessageType.GO_AHEAD;
            }
            if (status == Status.Requesting) {
                if (Integer.parseInt(receivedMessageParts[3]) < requestLogicalClock) {
                    messageType = MessageType.GO_AHEAD;

                } else if (Integer.parseInt(receivedMessageParts[3]) ==
requestLogicalClock) {
                    if (Integer.parseInt(receivedMessageParts[1]) < Integer.parseInt(id)) {
                        messageType = MessageType.GO_AHEAD;
                    } else {
                        DeferredWhileCriticalState.add(receivedMessageParts[0]);
                        messageType = null;
                    }
                } else {
                    logicalClock = Math.max(logicalClock,
Integer.parseInt(receivedMessageParts[3])) + 1;
                    DeferredWhileCriticalState.add(receivedMessageParts[0]);
                    messageType = null;
                }

            }
            if (status == Status.InCriticalState) {
                DeferredWhileCriticalState.add(receivedMessageParts[0]);
            } else {
                if (status == Status.AfterCriticalState) {
                    if (!DeferredWhileCriticalState.isEmpty()) {
                        System.out.println(name + " exiting critical state.");
                        sendGO_AHEADMessage();
                        DeferredWhileCriticalState.clear();
                    }
                } else if (messageType != null) {
                    sendResponseMessage(receivedMessageParts[0]);
                }
            }
        }

        try {
            Thread.sleep(1000); // Sleep for 1 second
        } catch (InterruptedException e) {
```

```
                Thread.currentThread().interrupt();
            }
        }
    }


    private void setStatus() {
        int number = random.nextInt(22); // Generate a random number between 0 and 20
        if (number >= 13 && number < 21) {
            // A node has 38.10% chance to change its status to Requesting
            this.status = Status.Requesting;
        }
    }

    private void checkClock() {
        if (clockAfterCriticalState != 0) {
            clockAfterCriticalState--;
        }
        if (clockInCriticalState != 0) {
            //If still in critical state
            status = Status.InCriticalState;
            clockInCriticalState--;
        }
        if (clockInCriticalState == 0 && clockAfterCriticalState != 0) {
            //If just came out from critical state but still cannot request
            status = Status.AfterCriticalState;
            isRequested = false;
        }
    }

    private void sendResponseMessage(String recipientNodeName) {
        //Send any response message to specific recipient
        sMessage = name + ":" + id + ":" + messageType + ":" + logicalClock;
        messageSystem.setMessage(sMessage, recipientNodeName);
        System.out.println("Send Message :: Name: " + name + " id: " + id + " status: " +
status + " message sent: "
                + sMessage + " to: " + recipientNodeName + " at: " + logicalClock);
        logicalClock++;
    }

    private void sendRequestMessage() {
        // To send request message for critical state
        if (status == Status.Requesting && !isRequested) {
            for (Node sendNode : nodes) {
```

```
            if (!sendNode.name.equalsIgnoreCase(name)) {
                String rMessage = name + ":" + id + ":" + MessageType.REQUEST + ":" +
logicalClock;
                messageSystem.setMessage(rMessage, sendNode.name);
                System.out.println("Request Message :: Name: " + name + " id: " + id + "
status: " + status + " message sent: "
                        + rMessage + " to: " + sendNode.name + " at: " + logicalClock);
            }
        }
        requestLogicalClock = logicalClock; // Storing the actual timestamp of
requesting
        isRequested = true;
      }
   }
   private void sendGO_AHEADMessage() {
      // To send agree to all the node who requested while this node is in critical state
      for (String sendNode : DeferredWhileCriticalState) {
        String rMessage = name + ":" + id + ":" + MessageType.GO_AHEAD + ":" +
logicalClock;
        System.out.println("Send Message :: Name: " + name + " id: " + id + " status: "
+ status + " message sent: "
                + rMessage + " to: " + sendNode + " at: " + logicalClock);
        messageSystem.setMessage(rMessage, sendNode);
        logicalClock++;
      }
   }
}
```

2.3. >> MessageType.java

```
public enum MessageType {
   GO_AHEAD,
   REQUEST
}
```

2.4. >> Status.java

```
public enum Status {
   InCriticalState, // While in critical state
   AfterCriticalState, // just after critical state but cannot request right away
   None, // Default
   Requesting // Requesting for critical state
}
```

2.5. >> Message.java

```
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.LinkedBlockingQueue;

public class Message {
   private ConcurrentHashMap<String, BlockingQueue<String>> messageQueues =
new ConcurrentHashMap<>();
   // Store message for specific recipient



   public Message(List<String> nodeNames) {
      for (String nodeName : nodeNames) {
         messageQueues.put(nodeName, new LinkedBlockingQueue<>());
      }
   }

   public void setMessage(String message, String recipientName) {
      // To set messages in hashmap
      BlockingQueue<String> recipientQueue =
messageQueues.get(recipientName);
      if (recipientQueue != null) {
         try {
            recipientQueue.put(message);
         } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
         }
      }
   }

   public String getMessage(String name) {
      // When a recipient want to get intended message
      BlockingQueue<String> queue = messageQueues.get(name);
      if (queue != null) {
         try {
            return queue.poll();
         } catch (Exception e) {
            Thread.currentThread().interrupt();
         }
      }
      return null;
   }
}
```

3. Pre-requisites & Assumptions

3.1. Concurrency Handling:

The code uses Java's concurrency utilities like ConcurrentHashMap, BlockingQueue, and CopyOnWriteArraySet to handle concurrent operations and ensure thread safety.

3.2. Message Handling:

The Message class handles message passing between nodes using a concurrent map of blocking queues.

3.3. Node Behavior:

Each Node simulates behavior in a distributed system, including requesting, entering, and exiting critical states using the Ricart–Agrawala algorithm.

3.4. Randomized Status Changes:

The Node class randomly changes its status to simulate requests for entering the critical section. Each node 38.10% chance for requesting state.

3.5. File Structure

Ensure you have a text file containing the node names separated by commas.

The file should look like this:

node1, node2, node3, node4

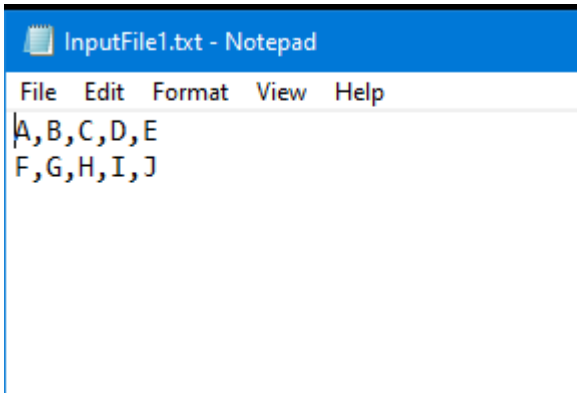
The file can contain multiple lines if you have many nodes:

node1, node2

node3, node4

4. Output

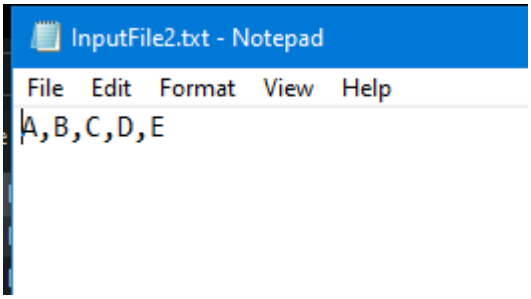
1. **Input file 1:**



2. **Output PDF 1:**

..\Output\output1.pdf

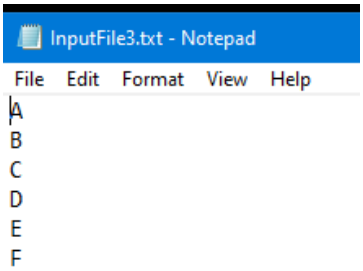
3. **Input file 2:**



4. **Output PDF 2:**

..\Output\output2.pdf

5. **Input file 3:**



6. **Output PDF 3:**

..\Output\output3.pdf

5. Remarks

5.1. Concurrency and Synchronization

The implementation makes use of Java's concurrency utilities such as ConcurrentHashMap, BlockingQueue, and CopyOnWriteArraySet to manage concurrent access to shared resources. This ensures thread safety and helps in achieving mutual exclusion without deadlocks or race conditions.

5.2. Message Passing

The Message class efficiently handles message passing between nodes by maintaining a map of blocking queues. Each node has its own queue for receiving messages, facilitating non-blocking communication and enabling asynchronous operations.

5.3. Node Behavior and State Management

Each Node instance runs in its own thread and follows a lifecycle of states (None, Requesting, InCriticalState, AfterCriticalState). The state transitions are governed by the Ricart–Agrawala algorithm, which ensures mutual exclusion by exchanging request and go-ahead messages.

5.4. Logical Clock Management

The implementation uses Lamport's logical clock to manage the ordering of events across the distributed system. The logical clock is incremented based on internal and received events, helping in maintaining a consistent view of the system's state.

5.5. Randomized Status Changes

The node status changes are driven by a random number generator, simulating real-world scenarios where requests for critical sections occur unpredictably. This randomness adds robustness to the simulation by testing the algorithm under varying conditions.

5.6. Deferred Requests Handling

Nodes maintain a list of deferred requests, ensuring that any requests received while in the critical state are appropriately handled once the critical state is exited. This prevents starvation and ensures that all nodes eventually gain access to the critical section.

5.7. File-Based Node Initialization

The nodes are initialized based on a configuration file, allowing flexibility in defining different network topologies and node setups. The file dialog interface provides a user-friendly way to select the configuration file.

5.8. Exception Handling and Robustness

The code includes appropriate exception handling mechanisms, such as dealing with interrupted exceptions in blocking operations and file not found exceptions during node initialization. This enhances the robustness and stability of the system.

5.9. Simulation and Debugging

The use of extensive print statements for logging the internal states and message exchanges between nodes helps in debugging and understanding the flow of the algorithm. These logs provide insights into how nodes interact and transition between states.

5.10. Scalability and Extensibility

The design of the system is modular and can be easily scaled by adding more nodes or modifying the existing ones. The use of Java's concurrent collections and thread management makes it adaptable to larger and more complex distributed systems.