**Graphical user interface, text

Description automatically generated with medium confidence**

**Ricart-Agrawala Algorithm**

By:

Revanth Sakthivelan

[sakthr00@jcu.cz](mailto:sakthr00@jcu.cz)

&

Mihir Rajnikant Kotecha

kotecm00@jcu.cz

Table of Contents

[Introduction: 3](#_Toc123766601)

[Algorithm: 3](#_Toc123766602)

[Message Complexity: 4](#_Toc123766603)

[Code Description: 4](#_Toc123766604)

[ RicartAgrawala.java: 4](#_Toc123766605)

[ RicartAgrawalaClient.java: 5](#_Toc123766606)

[ RicartAgrawalaServer.java: 6](#_Toc123766607)

[ RicartAgrawalaServerImpl.java: 7](#_Toc123766608)

[Advantages: 9](#_Toc123766609)

[Disadvantages: 10](#_Toc123766610)

# **Introduction:**

The Ricart-Agrawala algorithm is used for Mutual exclusion that allows multiple processes in a distributed system to access a shared resource without conflict. It was proposed by Glenn Ricart and Ashok Agrawala in their 1979 paper **"An optimal algorithm for mutual exclusion in computer networks"**. It 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.

# **Algorithm:**

Diagram

Description automatically generated

In this algorithm, there are two types of messages **REQUEST** and **REPLY** are used. The communication channels must follow **FIFO** order in this.

* A process P1 sends a **REQUEST** message to all other processes to enter critical section.
* Another process P2 responds to the **REQUEST** message with a **REPLY** message to the process to give its permission to enter the critical section.
* A **timestamp** is given to the **REQUEST** using Lamport’s logical clock. It is used to determine priority of critical section requests. The smaller timestamp gets higher priority.
* When a process P2 receives a **REQUEST** message from process P1, it sends a **REPLY** message to process P1 if and only if
  + Process P2 is neither requesting nor currently executing the critical section.
  + In case process P2 is requesting, the timestamp of process P1’s request is smaller than its own request.
* Otherwise, the request is deferred by process P2.
* For a process P1 to enter critical section, it must get a **REPLY** message from all the other processes.
* After all the work is done, process P1 exits the critical section and sends a **REPLY** message to all the deferred requests.

# **Message Complexity:**

According to the Ricart-Agrawala algorithm, executing a critical section requires the invocation of 2(N-1) messages. These 2(N–1) messages involves:

* (N–1) **REQUEST** messages
* (N–1) **REPLY** messages

# **Code Description:**

# **RicartAgrawala.java:**

* import java.rmi.Remote;
* import java.rmi.RemoteException;
* public interface RicartAgrawala extends Remote {
* public String Process1() throws RemoteException;
* public String Process2() throws RemoteException;
* public String Process3() throws RemoteException;
* public String Process4() throws RemoteException;
* }
* The given code is an interface in Java for the Ricart-Agrawala algorithm, which is a distributed mutual exclusion algorithm used to ensure that only one process can access a shared resource at a time in a distributed system.
* The interface defines four methods: Process1, Process2, Process3, and Process4. All of these methods throw a RemoteException, which is a checked exception in Java that indicates a communication-related failure in a remote method call.
* The Remote interface, which is imported at the top of the code, is a marker interface in Java that is used to identify interfaces whose methods may be invoked remotely.

# **RicartAgrawalaClient.java:**

* import java.rmi.registry.LocateRegistry;
* import java.rmi.registry.Registry;
* public class RicartAgrawalaClient {
* public static void main(String[] args) throws Exception {
* String ip = "192.168.178.49";
* int port = 1100;
* try {
* Registry registry = LocateRegistry.getRegistry(ip, port);
* RicartAgrawala stub = (RicartAgrawala) registry.lookup("RicartAgrawalaServer");
* String response1 = stub.Process1();
* String response2 = stub.Process2();
* String response3 = stub.Process3();
* String response4 = stub.Process4();
* System.out.println("Response 3: " + response3);
* System.out.println("Response 1: " + response1);
* System.out.println("Response 4: " + response4);
* System.out.println("Response 2: " + response2);
* } catch (Exception e) {
* System.out.println("Error: " + e.toString());
* }
* }
* }
* The client in this code uses the LocateRegistry and Registry classes to locate the remote object registry on a specified IP address and port.
* It obtains a reference to the remote object (the server) through the lookup method of the Registry class, and casts it to the RicartAgrawala interface.
* The client then invokes the Process1, Process2, Process3, and Process4 methods of the server and prints the responses to the console.
* If an exception is thrown during this process, it is caught and the error message is printed to the console.

# **RicartAgrawalaServer.java:**

* import java.rmi.RemoteException;
* import java.rmi.registry.\*;
* import java.rmi.server.\*;
* public class RicartAgrawalaServer {
* public static void main(String args[]) throws RemoteException {
* // important for the rmi registry location
* System.setProperty("java.rmi.server.hostname", "192.168.178.49");
* Registry reg = null;
* int port = 1100;
* try {
* reg = LocateRegistry.createRegistry(port);
* } catch (Exception ce) {
* System.out.println("Error: " + ce.toString());
* }
* // creating instance of implemented class
* RicartAgrawalaServerImpl ricartAgrawalaImpl = new RicartAgrawalaServerImpl();
* try {
* RicartAgrawala stub = (RicartAgrawala) UnicastRemoteObject.exportObject(ricartAgrawalaImpl, 1100);
* // bind token "Hello" and the created instance
* reg.rebind("RicartAgrawalaServer", stub);
* System.out.println("Server has been started.");
* } catch (Exception e) {
* System.out.println("Error: " + e.toString());
* }
* return;
* }
* }
* The server sets the system property java.rmi.server.hostname to a specified IP address and creates a registry on a specified port using the LocateRegistry.createRegistry method.
* It creates an instance of the RicartAgrawalaServerImpl class, which implements the RicartAgrawala interface.
* The server exports the ricartAgrawalaImpl object as a remote object using the UnicastRemoteObject.exportObject method and binds it to the registry using the rebind method.
* It prints a message to the console indicating that the server has been started.
* If any exceptions are thrown during this process, they are caught and the error message is printed to the console.

# **RicartAgrawalaServerImpl.java:**

* import java.rmi.RemoteException;
* public class RicartAgrawalaServerImpl implements RicartAgrawala {
* private boolean inCS; // flag to indicate if server is in critical section
* private int numRequests; // number of pending requests
* public RicartAgrawalaServerImpl() {
* inCS = false;
* numRequests = 0;
* }
* public String Process1() {
* numRequests++; // increment number of pending requests
* while (inCS || numRequests > 1) {
* // If server is in critical section or there are pending requests, wait
* try {
* Thread.sleep(10);
* } catch (InterruptedException e) {
* e.printStackTrace();
* }
* }
* inCS = true; // enter critical section
* numRequests--; // decrement number of pending requests
* String response = "Process 1 is implemented";
* inCS = false; // exit critical section
* return response;
* }
* @Override
* public String Process2() throws RemoteException {
* numRequests++; // increment number of pending requests
* while (inCS || numRequests > 1) {
* // If server is in critical section or there are pending requests, wait
* try {
* Thread.sleep(10);
* } catch (InterruptedException e) {
* e.printStackTrace();
* }
* }
* inCS = true; // enter critical section
* numRequests--; // decrement number of pending requests
* String response = "Process 2 is implemented";
* inCS = false; // exit critical section
* return response;
* }
* @Override
* public String Process3() throws RemoteException {
* numRequests++; // increment number of pending requests
* while (inCS || numRequests > 1) {
* // If server is in critical section or there are pending requests, wait
* try {
* Thread.sleep(10);
* } catch (InterruptedException e) {
* e.printStackTrace();
* }
* }
* inCS = true; // enter critical section
* numRequests--; // decrement number of pending requests
* String response = "Process 3 is implemented";
* inCS = false; // exit critical section
* return response;
* }
* @Override
* public String Process4() throws RemoteException {
* numRequests++; // increment number of pending requests
* while (inCS || numRequests > 1) {
* // If server is in critical section or there are pending requests, wait
* try {
* Thread.sleep(10);
* } catch (InterruptedException e) {
* e.printStackTrace();
* }
* }
* inCS = true; // enter critical section
* numRequests--; // decrement number of pending requests
* String response = "Process 4 is implemented";
* inCS = false; // exit critical section
* return response;
* }
* }
* The class has two instance variables: inCS, a boolean flag that indicates whether the server is in the critical section, and numRequests, an integer that keeps track of the number of pending requests. The class has a default constructor that initializes these variables to false and 0, respectively.
* The class defines four methods: Process1, Process2, Process3, and Process4. These methods are synchronized and use the Ricart-Agrawala algorithm to ensure that only one process can access the critical section at a time.

**The algorithm works as follows:**

* The method increments the number of pending requests and enters a loop.
* If the server is in the critical section or there are pending requests, the method waits for a short period of time before checking again.
* If the server is not in the critical section and there are no pending requests, the method enters the critical section by setting the inCS flag to true.
* The method decrements the number of pending requests and performs the necessary actions.
* The method exits the critical section by setting the inCS flag to false and returns a response string to the client indicating that the process has been implemented.
* If any InterruptedExceptions are thrown during the waiting process, they are caught and the stack trace is printed.

# **Output:**

Text

Description automatically generated

Text

Description automatically generated

* Client 1 and Client 2 are processes that want to access the shared resource, which is protected by the Server.
* Client 1 sends a request to the Server to access the critical section.
* Client 2 also sends a request to the Server to access the critical section.
* The Server implements the Ricart-Agrawala algorithm to ensure that only one process can access the critical section at a time. It grants access to Client 1 and sends a response to Client 1 indicating that it has received the grant.
* Client 1 receives the grant from the Server and enters the critical section. It performs its operation on the shared resource and then exits the critical section.
* The Server grants access to Client 2 and sends a response to Client 2 indicating that it has received the grant.
* Client 2 receives the grant from the Server and enters the critical section. It performs its operation on the shared resource and then exits the critical section.

**UML Diagram:**

**Graphical user interface, text

Description automatically generated with medium confidence**

* The RicartAgrawalaClient class represents a client process in a distributed system. It sends requests to the Server to access the shared resource.
* The RicartAgrawalaServer class represents the server process that controls access to the shared resource. It implements the RicartAgrawala interface, which defines the methods for accessing the critical section.
* The RicartAgrawala interface is a marker interface that identifies interfaces whose methods may be invoked remotely. It defines four methods: Process1, Process2, Process3, and Process4, which are implemented by the Server class.
* The RicartAgrawalaServerImpl class represents the remote object registry, which is a remote server that maintains a registry of remote object references bound to human-readable names.

# **Advantages:**

* **Simple**: The algorithm is relatively simple and easy to understand, making it suitable for use in a wide range of distributed systems.
* **Efficient**: The algorithm is efficient and has a low overhead, as it requires only a small number of messages to be exchanged between processes.
* **Scalable**: The algorithm scales well with the number of processes in the system, making it suitable for use in large distributed systems.

# **Disadvantages:**

* **Deadlock:** The system of nodes is said to be deadlocked when no node is in its critical section and no requesting node can ever proceed to its own critical section.
* **Starvation:** Starvation can be an issue in the Ricart-Agrawala algorithm when a process is frequently denied access to the shared resource due to having a low priority and being constantly interrupted by processes with higher priorities.
* **Unreliable approach**: Failure of any one of node in the system can halt the progress of the system.

**References:**

* Ricart, G., & Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM, 24(1), 9-17.

<https://www.cs.ucf.edu/courses/cop6614/fall2005/Ricart-Agrawala.pdf>

* The article was written by various contributors and is available at the following link: <https://www.geeksforgeeks.org/ricart-agrawala-algorithm-in-mutual-exclusion-in-distributed-system/>
* Analysis of Distributed Mutual Exclusion Algorithms https://media.springernature.com/lw685/springer-static/image/chp%3A10.1007%2F978-981-10-5547-8\_50/MediaObjects/431837\_1\_En\_50\_Fig5\_HTML.gif