Gandaki College of Engineering and Science

Distributed System

Lab Experiment: Distributed Mutual Exclusion using Ricart and Agrawala Algorithm

Lab: 05

Submitted By:	**Submitted To:**
Name: Dipendra Raut Kurmi Roll No: 18 Subject: DS	Er. Amrit Poudel Lecturer at Gandaki College of Engineering and Science

1.Objective:

To implement and demonstrate the Ricart and Agrawala distributed mutual exclusion algorithm using Java socket programming:
✔ Mutual Exclusion (only one process in CS at a time).
✔ Deadlock Freedom (no indefinite waiting).
✔ Fairness (requests granted in order of timestamps).

2. Problem Statement

Design a multi-process simulation where processes coordinate access to a critical section (CS) using:
- REQUEST messages (before entering CS).
-REPLY messages (granting permission).
-Logical clocks (Lamport timestamps for ordering).

3.Tools & Technologies

- Java JDK 8+ (Socket Programming , Multithreading)
- Terminal (For code Execution).
- VS code (Code Editing)

4.Implementation

4.1 Key Components:

Components	Description
RicartAgrawalaProcess	Main class handling message passing and CS access
ServerSocket	Listening for incoming messages
Lamport Clock	Logical clock for event ordering
REQUEST/REPLY	Message type for mutual exclusion
Deferred Queue	Hold pending requests when CS is occupied

4.2 Algorithm Steps:

1) Requesting CS:
    - Increment logical clock
    - Send REQUEST(ts, pid) to all processes.

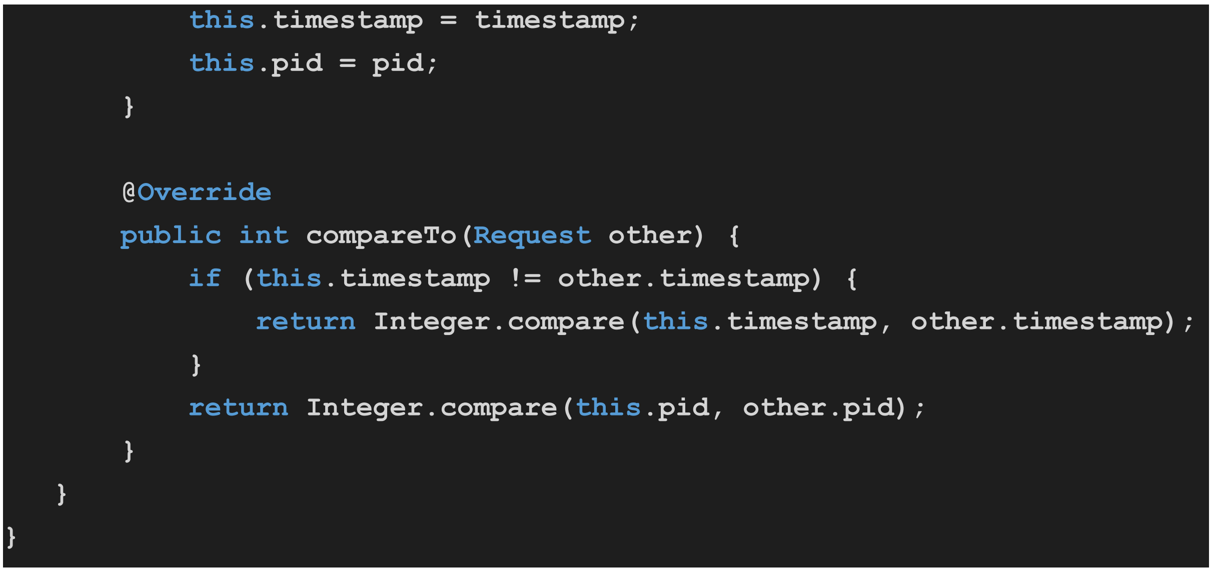
2) Granting CS:
    - Wait for reply from all processes.
    - Enter CS if all replies are received.

3) Releasing CS
    - Exit CS and REPLY to deferred requests.

4.2 Code:

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

public class RicartAgrawalaProcess {
    private int pid;
    private int clock;
    private List<Integer> otherPorts;
    private ServerSocket serverSocket;
    private boolean wantCS = false;
    private int repliesNeeded;
    private PriorityQueue<Request> deferredRequests = new
PriorityQueue<>();

    // For tracking replies received
    private Set<Integer> repliesReceived =
ConcurrentHashMap.newKeySet();

    public RicartAgrawalaProcess(int pid, int clock, List<Integer>
otherPorts) {
        this.pid = pid;
        this.clock = clock;
        this.otherPorts = otherPorts;
        this.repliesNeeded = otherPorts.size();
    }

    public void start() {
        try {
            // Start server socket
            int port = 5000 + pid;
            serverSocket = new ServerSocket(port);
            System.out.println("Process " + pid + " listening on port "
+ port);

            // Start message listener thread
            new Thread(this::listenForMessages).start();

            // Start periodic CS requests
            new Thread(this::periodicallyRequestCS).start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void periodicallyRequestCS() {
        Random random = new Random();
        while (true) {
            try {
                // Random delay between CS requests (5-15 seconds)
                Thread.sleep(5000 + random.nextInt(10000));

                // Request CS
                requestCriticalSection();

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private void requestCriticalSection() {
        wantCS = true;
        clock++;
        repliesReceived.clear();

        System.out.println("Process " + pid + " requesting CS at time "
+ clock);

        // Send request to all other processes
        for (int port : otherPorts) {
            sendMessage(port, "REQUEST:" + clock + ":" + pid);
        }
    }

    private void listenForMessages() {
        try {
            while (true) {
                Socket clientSocket = serverSocket.accept();
                new Thread(() ->
handleClientConnection(clientSocket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void handleClientConnection(Socket clientSocket) {
        try (BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()))) {
            String message = in.readLine();
            if (message != null) {
                processMessage(message);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void processMessage(String message) {
        String[] parts = message.split(":");
        String type = parts[0];
        int timestamp = Integer.parseInt(parts[1]);
        int senderPid = Integer.parseInt(parts[2]);

        // Update clock
        clock = Math.max(clock, timestamp) + 1;

        switch (type) {
            case "REQUEST":
                handleRequest(timestamp, senderPid);
                break;
            case "REPLY":
                handleReply(senderPid);
                break;
        }
    }

    private void handleRequest(int timestamp, int senderPid) {
        boolean shouldDefer = wantCS && (timestamp > clock || (timestamp
== clock && senderPid > pid));

        if (shouldDefer) {
            // Defer the reply
            System.out.println("Process " + pid + " deferring reply to "
+ senderPid);
            deferredRequests.add(new Request(timestamp, senderPid));
        } else {
            // Reply immediately
            sendMessage(5000 + senderPid, "REPLY:" + clock + ":" + pid);
            System.out.println("Process " + pid + " sent reply to " +
senderPid);
        }
    }

    private void handleReply(int senderPid) {
        System.out.println("Process " + pid + " received reply from " +
senderPid);
        repliesReceived.add(senderPid);

        if (repliesReceived.size() == repliesNeeded && wantCS) {
            enterCriticalSection();
        }
    }

    private void enterCriticalSection() {
        wantCS = false;
        System.out.println("Process " + pid + " ENTERING critical
section.");

        try {
            // Simulate CS work (3-5 seconds)
            Thread.sleep(3000 + new Random().nextInt(2000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        exitCriticalSection();
    }

    private void exitCriticalSection() {
        System.out.println("Process " + pid + " EXITING critical
section.");


        // Reply to all deferred requests
        while (!deferredRequests.isEmpty()) {
```

```java
            Request req = deferredRequests.poll();
            sendMessage(5000 + req.pid, "REPLY:" + clock + ":" + pid);
            System.out.println("Process " + pid + " sent deferred reply
to " + req.pid);
        }
    }

    private void sendMessage(int port, String message) {
        try (Socket socket = new Socket("localhost", port);
             PrintWriter out = new PrintWriter(socket.getOutputStream(),
true)) {
            out.println(message);
        } catch (IOException e) {
            System.err.println("Process " + pid + " failed to send
message to port " + port);
        }
    }

    public static void main(String[] args) {
        if (args.length < 3) {
            System.err.println("Usage: java RicartAgrawalaProcess <pid>
<clock> <port1> <port2> ...");
            System.exit(1);
        }

        int pid = Integer.parseInt(args[0]);
        int clock = Integer.parseInt(args[1]);
        List<Integer> otherPorts = new ArrayList<>();

        for (int i = 2; i < args.length; i++) {
            otherPorts.add(Integer.parseInt(args[i]));
        }

        RicartAgrawalaProcess process = new RicartAgrawalaProcess(pid,
clock, otherPorts);
        process.start();
    }

    private static class Request implements Comparable<Request> {
        int timestamp;
        int pid;

        Request(int timestamp, int pid) {
```



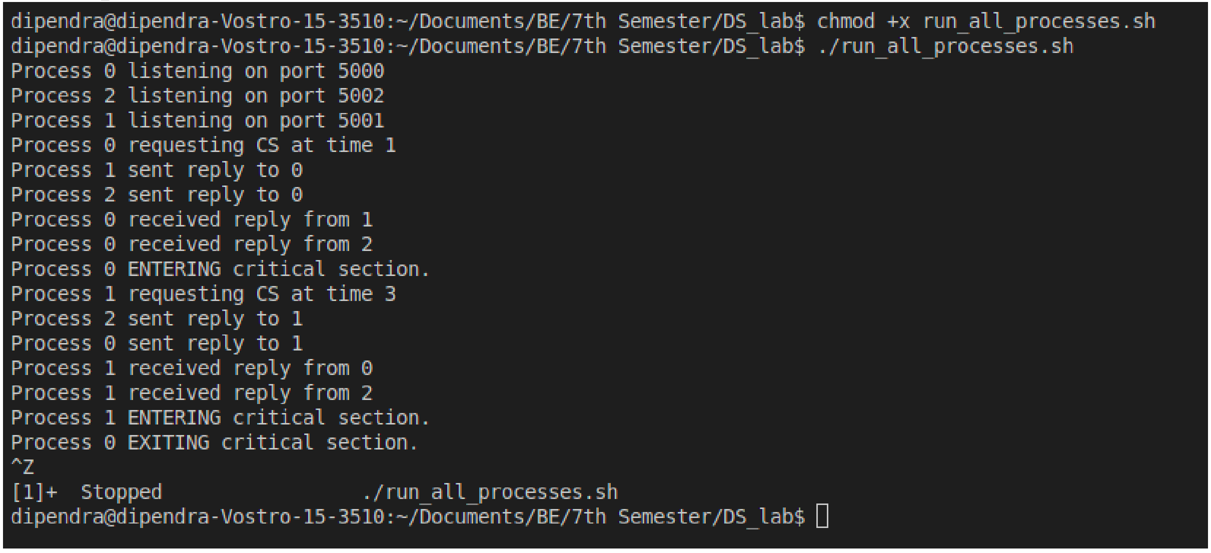
```java
            this.timestamp = timestamp;
            this.pid = pid;
        }

        @Override
        public int compareTo(Request other) {
            if (this.timestamp != other.timestamp) {
                return Integer.compare(this.timestamp, other.timestamp);
            }
            return Integer.compare(this.pid, other.pid);
        }
    }
}
```

5. Experiments and Result:

5.1 Setup

Processes: 3 (P0, P1, P2).
Ports: 5000, 5001, 5002.
Test Case: All processes randomly request CS.

5.2 Output



```
dipendra@dipendra-Vostro-15-3510:~/Documents/BE/7th Semester/DS_lab$ chmod +x run_all_processes.sh
dipendra@dipendra-Vostro-15-3510:~/Documents/BE/7th Semester/DS_lab$ ./run_all_processes.sh
Process 0 listening on port 5000
Process 2 listening on port 5002
Process 1 listening on port 5001
Process 0 requesting CS at time 1
Process 1 sent reply to 0
Process 2 sent reply to 0
Process 0 received reply from 1
Process 0 received reply from 2
Process 0 ENTERING critical section.
Process 1 requesting CS at time 3
Process 2 sent reply to 1
Process 0 sent reply to 1
Process 1 received reply from 0
Process 1 received reply from 2
Process 1 ENTERING critical section.
Process 0 EXITING critical section.
^Z
[1]+  Stopped                 ./run_all_processes.sh
dipendra@dipendra-Vostro-15-3510:~/Documents/BE/7th Semester/DS_lab$ 
```

6. Conclusion

Hence, The Ricart-Agrawala algorithm successfully enforces distributed mutual exclusion.