**ASSIGNMENT NO: 10**

**Title:**

Raymond’s tree-based algorithm.

**Problem Statement:**

Implement Raymond’s tree-based algorithm.

**Tools / Environment:**

The following tools and technologies will be used in this project:

Programming language: Java

IDE: Eclipse

Networking: Java Sockets

**Theory:**

**Raymond’s tree based algorithm** is lock based algorithm for mutual exclusion in a distributed system in which a site is allowed to enter the critical section if it has the token. In this algorithm, all sites are arranged as a directed tree such that the edges of the tree are assigned direction towards the site that holds the token. Site which holds the token is also called root of the tree.   
**Data structure and Notations:**

* Every site Si keeps a FIFO queue, called **request\_q**   
  This queue stores the requests of all neighbouring sites that have sent a request for the token to site Si but have not yet been sent token. A non-empty request\_q at any site indicates that the site has sent a REQUEST message to the root node.  
  Every site Si has a local variable, called holder   
  This variable points to an immediate neighbour node on a directed path to the root node.

Algorithm:

To enter Critical section:

When a site Si wants to enter the critical section it sends a REQUEST message to the node along the directed path to the root, provided it does not hold the token and its request\_q is empty. After sending REQUEST message it add its request to its request\_q.  
when a site Sj on the path to the root receives the REQUEST message of site Si, it places the REQUEST in its request\_q and sends the REQUEST message along the directed path to the root, if it has not sent any REQUEST message for previously received REQUEST message.  
When the root site Sr( having token) receives the REQUEST message, it sends the token to the requesting site and sets its holder variable to point at that site.  
On receiving the token, Site Sj deletes the top entry from its request\_q and sends the token to the site indicated by deleted entry. holder variable of Site Sj is set to point at that site.   
After deleting the topmost entry of the request\_q, if it is still non-empty Site Sj sends a REQUEST message to the site indicated by holder variable in order to get token back.  
To execute the critical section:

Site Si executes the critical section if it has received the token and its own entry is at the top of its request\_q.  
To release the critical section:   
After finishing the execution of the critical section, site Si does the following:

If its request\_q is non-empty, then it deletes the top most entry from its <request\_q and then it sends the token to that site indicated by deleted entry and also its holder variable is set to point at that site.  
After performing above action, if the request\_q is still non-empty, then site Si sends a REQUEST message to the site pointed by holder variable in order to get token back

Message Complexity:   
In the worst case, the algorithm requires 2 \* ( Longest path length of the tree ) message invocation per critical section entry. If all nodes are arranged in a straight line then the longest path length will be N – 1 and thus the algorithm will require 2 \* (N -1) message invocation for critical section entry. However, if all nodes generates equal number of REQUEST messages for the privilege, the algorithm will require approximately 2\*N / 3 messages per critical section entry.

Drawbacks of Raymond’s tree based algorithm:

can cause starvation: Raymond’s algorithm uses greedy strategy as a site can executes the critical section on receiving the token even when its request is not on the top of the request queue. This affect the fairness of the algorithm and thus can cause in starvation.

Performance:

Synchronization delay is (T \* log N )/ 2, because the average distance between two sites to successively execute the critical section is (Log N)/2. Here T is maximum message transmission time.  
In heavy load conditions, the synchronization delay become T because a site executes the critical section every time the token is transferred.  
The message complexity of this algorithm is O(log N) as the average distance between any two nodes in a tree with N nodes is log N  
Deadlock is impossible.

**Conclusion:**

Raymond's tree-based algorithm offers a practical and interpretable solution for solving classification and prediction problems. Its versatility, scalability, and interpretability make it a valuable tool in various domains, from healthcare to finance and beyond. However, practitioners should be mindful of its limitations and employ appropriate techniques to overcome potential challenges.

**Code:**

**Client-side code :**

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

import java.util.LinkedList;

import java.util.Queue;

public class Node implements Runnable {

private int id;

private int parentId;

private int numNodes;

private Socket socket;

private ObjectInputStream inputStream;

private ObjectOutputStream outputStream;

private Node parent;

private Queue<Message> messageQueue;

public Node(int id, int parentId) {

this.id = id;

this.parentId = parentId;

this.messageQueue = new LinkedList<>();

}

public void setSocket(Socket socket) {

this.socket = socket;

}

public void setInputStream(ObjectInputStream inputStream) {

this.inputStream = inputStream;

}

public void setOutputStream(ObjectOutputStream outputStream) {

this.outputStream = outputStream;

}

public int getParentId() {

return parentId;

}

public void setNumNodes(int numNodes) {

this.numNodes = numNodes;

}

public void broadcast(String type, String message) {

for (Node child : getChildren()) {

child.sendMessage(new Message(type, message, this));

}

}

public void sendMessage(Message message) {

try {

outputStream.writeObject(message);

} catch (IOException e) {

e.printStackTrace();

}

}

public void receiveMessage(Message message) {

switch (message.getType()) {

case "START":

if (id == 0) {

execute();

}

break;

case "REQUEST":

if (parent == null) {

messageQueue.add(message);

} else {

parent.receiveMessage(message);

}

break;

case "REPLY":

message.getSource().receiveReply();

break;

}

}

public void execute() {

// Implement Raymond's algorithm here

}

public void receiveReply() {

// Implement Raymond's algorithm here

}

public Node getParent() {

return parent;

}

public void setParent(Node parent) {

this.parent = parent;

}

public Queue<Message> getMessageQueue() {

return messageQueue;

}

public void run() {

try {

while (true) {

Message message = (Message) inputStream.readObject();

receiveMessage(message);

}

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

public Node[] getChildren() {

Node[] children = new Node[numNodes-1];

int i = 0;

for (int j = 0; j < numNodes; j++) {

if (j != id && j != parentId) {

children[i++] = new Node(j, id);

}

}

return children;

}

}

**Server-side code:**

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.ServerSocket;

import java.net.Socket;

public class Server {

private ServerSocket serverSocket;

private Node rootNode;

private int numNodes;

private int count;

public Server(int port, int numNodes) {

this.numNodes = numNodes;

this.count = 0;

try {

this.serverSocket = new ServerSocket(port);

} catch (IOException e) {

e.printStackTrace();

}

}

public void run() {

try {

System.out.println("Waiting for connections...");

while (true) {

Socket socket = serverSocket.accept();

System.out.println("New client connected: " + socket);

ObjectInputStream objectInputStream = new ObjectInputStream(socket.getInputStream());

ObjectOutputStream objectOutputStream = new ObjectOutputStream(socket.getOutputStream());

Node node = (Node) objectInputStream.readObject();

if (node.getParentId() == -1) {

rootNode = node;

}

node.setSocket(socket);

node.setInputStream(objectInputStream);

node.setOutputStream(objectOutputStream);

count++;

if (count == numNodes) {

System.out.println("All clients connected!");

rootNode.setNumNodes(numNodes);

rootNode.broadcast("START", null);

}

}

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

public static void main(String[] args) throws IOException {

//int port = Integer.parseInt(args[0]);

//int numNodes = Integer.parseInt(args[1]);

if (args.length == 0) {

System.out.println("Usage: java Server <port>");

return;

}

int port = Integer.parseInt(args[0]);

int numNodes = Integer.parseInt(args[1]);

Server server = new Server(port, numNodes);

server.run();

}

}

**Output:**

