University Of Stirling

CSCU9V6 – Operating Systems Concurrency and Distribution

Distributed Systems Assignment

CSCU9V6- Operating Systems Concurrency and Distribution

Assignment Name: Distributed Systems Assignment

Student Number:

Use of AI

I acknowledge that :

I used ChatGPT (https://chat.openai.com/) on 5/04/2024 to generate comments and resolve some errors in my code that are included within my submission

Presentation, structure, and proofreading

I acknowledge that:

I used ChatGPT (https://chat.openai.com/) on 5/04/2024 to fix the structure of my code

Introduction

This report outlines the implementation of Distributed Mutual Exclusion (DME) using a token ring algorithm. In distributed systems, multiple processes often need to access shared resources concurrently, which can result in data inconsistency and race conditions if not coordinated properly. DME ensures exclusive access to critical sections of code that modify shared resources, mitigating these issues.

The token ring algorithm employs a virtual token that circulates among participating processes. Only the process holding the token is permitted to enter the critical section. Other processes requesting access must wait until they receive the token. This report provides a comprehensive overview of the design and implementation of a DME system utilizing the token ring approach.

This system consists of 7 classes:

- Node
- Coordinator
- C_mutex
- C_receiver
- C_connection_r
- C_buffer
- Logger

System Design

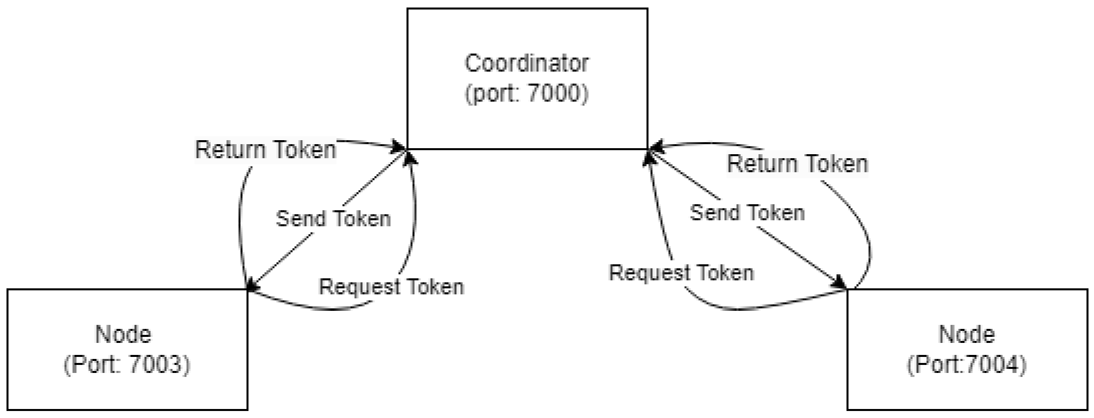
The system consists of two main components:

Coordinator: This process acts as a central authority responsible for managing the token and granting access to the critical section.

Node: This represents a process that wishes to enter the critical section to access a shared resource.

The interaction between these components can be summarized as follows:

1. A node sends a request message containing its hostname and port to the coordinator.
2. The coordinator grants the token to the requesting node, allowing it to enter the critical section.
3. The node executes its critical section operations and finishes accessing the shared resource.
4. The node returns the token back to the coordinator, signalling completion of its critical section.
5. The coordinator makes the token available to the next requesting node.



Assumptions

The implementation assumes:

- Reliable communication channels between nodes and the coordinator, with no message loss or corruption.
- All nodes are aware of the coordinator's hostname and port for sending requests.

Code Description

The system is implemented using several Java classes:

Node.java:

This class embodies a node process, encapsulating the node's hostname, port, duration of critical section execution (in milliseconds), and an optional shutdown command. The Node class fulfills the following functions:

- Requests the token from the coordinator.
- Upon token receipt, enters the critical section, executing it for a random duration.
- Returns the token to the coordinator upon completing the critical section.
- Manages the optional shutdown command to gracefully terminate the program.

Coordinator.java:

This class represents the coordinator process. It initiates the coordinator on a designated port, initializes a shared buffer object to handle token requests and returns, and spawns distinct threads:

- One thread listens for incoming token requests from nodes via C_receiver.java.
- Another thread manages token returns using C_mutex.java.

C_receiver.java & C_Connection_r.java:

These classes collaborate to manage incoming token requests:

- C_receiver listens for requests on a port and instantiates a new thread for each request using C_Connection_r.

- C_Connection_r receives request details (hostname and port) from a node and stores them in the shared buffer.

C_mutex.java:

This class oversees token management and grants access to the critical section:

- It listens for token returns on a separate port.
- Upon token return reception, it retrieves the corresponding request from the shared buffer.
- It allocates the token to the requesting node by establishing a connection.
- It awaits token return by the node after critical section execution.

C_buffer.java:

This class implements a synchronized buffer to oversee token requests:

- It stores node requests (hostname and port) as string arrays.
- It offers methods for addition, removal, and access of elements from the buffer in a thread-safe manner.
- It includes functions to verify buffer size and display its contents for debugging purposes.

Output:

Code Listing

Node.java:

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

public class Node {
    private Random ra;
    private String c_host = "127.0.0.1";
    private int c_request_port = 7000;
    private int c_return_port = 7001;
    private String n_host = "127.0.0.1";
    private String n_host_name;
    private int n_port;

    public Node(String nam, int por, int sec) {
        ra = new Random();
        n_host_name = nam;
        n_port = por;

        System.out.println("Node " + n_host_name + ":" + n_port + " - DME is 
active.");
        Logger.log("Node " + n_host_name + ":" + n_port + " - Requesting token 
from Coordinator.");
        try (Socket coordSocket = new Socket(c_host, c_request_port);
             OutputStreamWriter out = new
OutputStreamWriter(coordSocket.getOutputStream())) {
            Logger.log("Token issued to "+ n_host_name + ":" + n_port);
            out.write(n_host + ":" + n_port);
            out.flush();

            System.out.println("Node " + n_host_name + ":" + n_port + " -
waiting for the token.");

            int sleepTime = ra.nextInt(sec * 1000);
            System.out.println("Critical section Entered");
            Logger.log("Node " + n_host_name + ":" + n_port + " - Entering
critical section.");
            System.out.println("Performing tasks in the critical section");
            Thread.sleep(sleepTime);
            System.out.println("Exiting Critical section");
            Logger.log("Node " + n_host_name + ":" + n_port + " - Exiting
critical section.");
            System.out.println("Node " + n_host_name + ":" + n_port + "
finished critical section after " + (sleepTime / 1000) + " seconds.");
            Logger.log("Node " + n_host_name + ":" + n_port + " - Finished
critical section after " + (sleepTime / 1000) + " seconds.");
            try (Socket returnSocket = new Socket(c_host, c_return_port);
                 PrintWriter returnOut = new
PrintWriter(returnSocket.getOutputStream(), true)) {

                returnOut.println("Token returned to the coordinator");
            }

            System.out.println("Token returned to Coordinator.");
            Logger.log("Token returned to Coordinator.");
        } catch (IOException | InterruptedException e) {
            System.out.println(e);
            System.exit(1);
        }
    }

    public static void main(String args[]) {
        String n_host_name = "";
        int n_port;

        if (args.length != 2) {
            System.out.print("Usage: Node [port number] [millisecs]");
            System.exit(1);
        }

        try {
            InetAddress n_inet_address = InetAddress.getLocalHost();
            n_host_name = n_inet_address.getHostName();
            System.out.println("hostname of node:   " + n_host_name + ":" +
n_inet_address);
            Logger.log("Hostname of node: " + n_host_name + ":" +
n_inet_address);
        } catch (java.net.UnknownHostException e) {
            System.out.println(e);
            System.exit(1);
        }

        n_port = Integer.parseInt(args[0]);
        System.out.println("node port:  " + n_port);
        new Node(n_host_name, n_port, Integer.parseInt(args[1]));
    }
}
```

Coordinator.java:

```
import java.net.*;

public class Coordinator {

    public static void main(String args[]) {
        int port = 7000;

        try {
            System.out.println("Coordinator is starting...");
            Logger.log("Coordinator Started");
            InetAddress c_addr = InetAddress.getLocalHost();
            System.out.println("Coordinator address is " + c_addr);
            System.out.println("Coordinator host name is " +
c_addr.getHostName() + "\n\n");
        } catch (Exception e) {
            System.err.println(e);
            System.err.println("Cooordinator encountered an error");
        }

        if (args.length == 1) {
            port = Integer.parseInt(args[0]);
        }

        C_buffer buffer = new C_buffer();
        C_receiver receiver = new C_receiver(buffer, port);
        receiver.start();

        C_mutex mutex = new C_mutex(buffer, 7001);
        mutex.start();
        System.out.println("Coordinator Mutex has started, ready to manage
tokens.");
    }
}
```

Logger.java:

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Logger {

    private static final String LOG_FILE = "log.txt";
    private static final DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

    public static void log(String message) {
        try (PrintWriter writer = new PrintWriter(new FileWriter(LOG_FILE,
true))) {
            String timestamp = LocalDateTime.now().format(formatter);
            String logMessage = String.format("%s - %s%n", timestamp,
message);
            writer.println(logMessage);
        } catch (IOException e) {
            System.err.println("Error writing to log file: " +
e.getMessage());
        }
    }
}
```

C_Buffer.java:

```
import java.util.*;

public class C_buffer {

    private final Queue<String[]> data;

    public C_buffer() {
        data = new LinkedList<>();
    }

    public synchronized int size() {
        return data.size();
    }

    public synchronized void saveRequest(String[] r) {
        data.add(r);
        notify();
    }

    public synchronized void show() {
        for (String[] request : data)
            System.out.println(Arrays.toString(request));
    }

    public synchronized Object get() throws InterruptedException {
        while (data.isEmpty()) {
            wait();
        }
        return data.poll();
    }
}
```

C_connection_r.java:

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

public class C_Connection_r extends Thread {

    private final C_buffer buffer;
    private final Socket socket;
    private InputStream inputStream;
    private BufferedReader bufferedReader;
    private String pt;

    public C_Connection_r(Socket socket, C_buffer buffer) {
        this.socket = socket;
        this.buffer = buffer;
    }

    public void run() {
        try {
            InetAddress c_addr = InetAddress.getLocalHost();
            inputStream = socket.getInputStream();
            bufferedReader = new BufferedReader(new
InputStreamReader(inputStream));

            String receivedData = bufferedReader.readLine();
            String[] nodeDetails = receivedData.split(":");

            if (nodeDetails.length == 2) {
                buffer.saveRequest(nodeDetails);
                pt=nodeDetails[1];
                System.out.println("C:connection OUT received and recorded
request from " + nodeDetails[0] + ":" + nodeDetails[1] + "  (socket closed)");
                System.out.println("Token given to the node :
"+c_addr.getHostName()+":"+nodeDetails[1]);
            } else {
                System.out.println("Received invalid data from node.");
            }

        } catch (IOException e) {
            System.out.println(e);
            System.exit(1);
        } finally {
            try {
                if (socket != null) {
                    socket.close();
                    System.out.println("Token received back from port:" +pt);
                }
                if (bufferedReader != null) {
                    bufferedReader.close();
                    System.out.println("All Sockets Closed!!!");
                }
            } catch (IOException e) {
                System.out.println("Error closing resources: " + e);
            }
        }

        buffer.show();
    }
}
```

C_mutex.java:

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

public class C_mutex extends Thread {
    private final C_buffer buffer;
    private ServerSocket serverSocketForTokenReturn;

    public C_mutex(C_buffer buffer, int port) {
        this.buffer = buffer;
        try {
            this.serverSocketForTokenReturn = new ServerSocket(port);
            System.out.println("C:mutex server socket for token returns 
created on port: " + port);
        } catch (IOException e) {
            System.out.println("Could not listen on port " + port + ". " + 
e.getMessage());
            System.exit(-1);
        }
    }

    public void run() {
        System.out.println("C:mutex started, listening for token returns on 
port: " + serverSocketForTokenReturn.getLocalPort());
        while (!serverSocketForTokenReturn.isClosed()) {
            try {
                System.out.println("C:mutex Buffer size is: " + 
buffer.size());

                Object requestData = buffer.get();
                if (requestData instanceof String[]) {
                    String[] requestDetails = (String[]) requestData;

                    if (requestDetails.length == 2) {
                        String n_host = requestDetails[0];
                        int n_port = Integer.parseInt(requestDetails[1]);
                        System.out.println("C:mutex processing token request 
for: " + n_host + ":" + n_port);

                        try (Socket socketToNode = new Socket(n_host, n_port)) 
{
                            System.out.println("C:mutex granted token to: " + 
n_host + ":" + n_port);

                            try (Socket nodeReturnSocket = 
serverSocketForTokenReturn.accept()) {
                                System.out.println("C:mutex received token 
back from: " + n_host + ":" + n_port);
                            }
                        // } catch (ConnectException e) {
                        //     System.out.println("C:mutex Error connecting 
to: " + n_host + ":" + n_port + ". Connection refused.");
                        } catch (IOException e) {
                            System.out.println(" ");
                        }
                    }
                }
            } catch (InterruptedException e) {
                System.out.println("C:mutex was interrupted: " + 
e.getMessage());
```

```
                break;
            }
        }

        try {
            if (serverSocketForTokenReturn != null &&
!serverSocketForTokenReturn.isClosed()) {
                serverSocketForTokenReturn.close();
                System.out.println("C:mutex server socket for token returns
closed.");
            }
        } catch (IOException e) {
            System.out.println("C:mutex Error closing server socket for token
returns: " + e.getMessage());
        }
    }
}
```

C_receiver.java:

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

public class C_receiver extends Thread {

    private final C_buffer buffer;
    private final int port;
    private ServerSocket serverSocket;

    public C_receiver(C_buffer b, int p) {
        this.buffer = b;
        this.port = p;
    }

    public void run() {
        try {
            serverSocket = new ServerSocket(port);
            System.out.println("Coordinator Receiver running on port: " +
port);

            while (!serverSocket.isClosed()) {
                try {
                    Socket socketFromNode = serverSocket.accept();
                    System.out.println("C:receiver Coordinator has received a
request ...");

                    C_Connection_r connectionThread = new
C_Connection_r(socketFromNode, buffer);
```

```
                    connectionThread.start();
                } catch (IOException e) {
                    System.out.println("Exception when creating a connection:
" + e);
                }
            }
        } catch (IOException e) {
            System.out.println("Could not listen on port: " + port);
            System.exit(-1);
        } finally {
            try {
                if (serverSocket != null) {
                    serverSocket.close();
                }
            } catch (IOException e) {
                System.out.println("Could not close server socket: " + e);
            }
        }
    }
}
```

Critical Reflection

Strengths:

- Ensures serialized access to critical sections, preventing race conditions.
- Relatively straightforward to understand and implement.

Limitations:

- Relies on reliable communication channels, vulnerable to disruptions.

Future Scope

- **Scalability Enhancements:** Explore alternative DME algorithms for better scalability.
- **Fault Tolerance Mechanisms:** Implement mechanisms to handle node failures or network disruptions.
- **Performance Optimization:** Analyze performance bottlenecks and optimize system efficiency.
- **Security Integration:** Integrate security measures to ensure authorized access.
- **Distributed Deadlock Detection:** Investigate methods for detecting and resolving deadlocks in complex scenarios.