**I acknowledge that :**

Assignment 2: Distributed Systems

CSCU9V6 Concurrent and distributed systems

2831597

2024

No content generated by AI technology has been knowingly presented as my own work

in this submission;

Presentation, structure, and proofreading

**I acknowledge that:**

My final submission has been proofread by a dedicated proofreading software or AI tool (chatGPT 3.5)

Contents

[Assumptions 2](#_Toc163175047)

[The basic problem solution 3](#_Toc163175048)

[Coordinator 3](#_Toc163175049)

[C\_receiver 3](#_Toc163175050)

[C\_Connection\_r 3](#_Toc163175051)

[C\_mutex 3](#_Toc163175052)

[Node 4](#_Toc163175053)

[Advanced features 5](#_Toc163175054)

[Implementation of file logging mechanism 5](#_Toc163175055)

[Graceful shutdown initiated by a Node 5](#_Toc163175056)

[Non-implemented advanced features 7](#_Toc163175057)

[Priority discipline request queue 7](#_Toc163175058)

[Modified nodes that can deal with coordinator shutdown (or crash) 7](#_Toc163175059)

[Class diagram 7](#_Toc163175060)

[Code listings 8](#_Toc163175061)

[Coordinator.java 8](#_Toc163175062)

[C\_mutex.java 9](#_Toc163175063)

[C\_buffer.java 10](#_Toc163175064)

[C\_receiver.java 11](#_Toc163175065)

[C\_Connection\_r.java 12](#_Toc163175066)

[Node.java 13](#_Toc163175067)

[Logger.java 16](#_Toc163175068)

# Assumptions

In this section I will briefly describe the assumptions.

1. Code base consists of a server which receives requests and a node which sends requests.
2. The server processes incoming requests and send the nodes a token back to allow them to initiate their process.
3. There doesn’t seem to be a limit on the number of nodes that can be active at once suggesting that the application is scalable, allowing for an increase in the number of nodes running without any significant modification to the code.
4. The coordinator will receive request for a token continuously from any active node.
5. If the coordinator crashes the nodes will no longer be able to function as they can not receive a token (meaning they may not enter their critical section)
6. Only one node can be in the critical section at any given time and the other nodes must wait to enter their critical section.

# The basic problem solution

With my solution the application is currently working fully under the basic solution.

## Coordinator

I first started by modifying the Coordinator class by creating a single instance of the C\_buffer(), C\_receiver() and C\_mutex classes. The C\_receiver and C\_mutex classes are then started by calling the start method on both instances which makes them run concurrently.

## C\_receiver

Then within the C\_receiver class I created a ServerSocket that will listen for any incoming connections from any active nodes, passing in the port from the server.

A screen shot of a computer program

Description automatically generated

Once the socket receives a connection request from a node it will accept the request passing both the socket and the buffer to a new instance of the C\_Connection\_r class and then calls the start function on the new instance.

## C\_Connection\_r

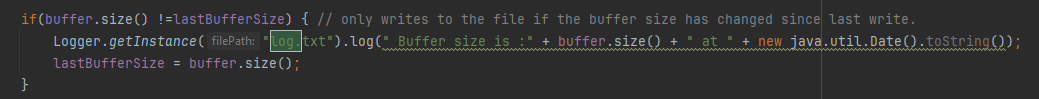
Within the C\_Connection\_r class I initiated an inputStream and BufferedReader, i then pass the node and port from the node socket into the BufferedReader with print lines along the way to relay to the user what is currently happening. Once this is done the socket is closed.

## C\_mutex

Within the C\_mutex class I put a print line to display the size of the buffer at the start of the while for debugging purposes.

I then initialise the n\_host and n\_port variables to be the values stored within the buffer after they are typecasted to the correct type.

I then have a line in to check if the buffer size has changed from the last time the buffer was read and if it has a message is printed to the log file.



This class then grants a token to the node by creating a socket that listens to the node which allows it to enter its critical section, once the nodes critical section is complete it then returns the token with ss\_back.accept();.

## Node

The node class is by far the class changed the most in. I start by creating a new node server socket “n\_ss” using the port number passed in through the runtime argument.

Once this is done i create a sleepTime variable with a max of 5000 milliseconds and a min of the sec input from the runtime args.

A screen shot of a computer

Description automatically generated

The program then sleeps the thread for a random time based on these variables.

The program then creates a new socket with the details of node and server address from the Coordinator class.

I then write to the output stream “pout” the nodes port and host name (n\_port and n\_host respectively)

Once this is completed the node will accept the token, sleep for a random time to simulate the critical section and then returns the token when it is complete printing suitable messages at each step to display to the user which part the application is currently undergoing (e.g start of critical section, end of critical section). Once completed all non-required sockets are closed.

As can be seen from this screenshot of the log file (explained later) no two nodes enter critical section at the same time.

A screenshot of a computer

Description automatically generated

This therefor completes the basic problem task.

# Advanced features

This section will cover advanced features and how I implement them / partially implemented them or my thinking on how I could implement them if they are not implemented.

## Implementation of file logging mechanism

This task I fully implement with great success, I first started by creating a class called “Logger” which deals with all requests to log information to the file, this class works by first resetting the log file as soon as the Coordinator is first ran, this ensures that the log file is empty and does not contain any information from previous runs of the application.

There is also a method used to get an instance of the logger, this will only create an instance if one does not already exist, if one exists it will return that instance instead.

A screen shot of a computer program

Description automatically generated

And finally the Logger class contains a function used to log messages to the file, it works by seeking the end of the file with log.seek(log.length()); and then uses log.writeBytes to write the message that is passed in the the function call. ( I originally used log.writeChars but that left a blank space between each letter in the message.)

Within other classes where appropriate an instance of the logger is gotten using Logger.getInstance along with the .log method to write a specific message to the log file.



Messages are written the the log file in three locations, start of the critical section of a node, end of the critical section of a node and in C\_mutex when the buffer size is changed.

The getInstance() and log() methods are synchronized to ensure that they are thread safe. If they were not synchronized it could be possible for multiple threads to write to the log file simultaneously creating inconsistences in the data or even corruption.

## Graceful shutdown initiated by a Node

For this advanced feature I only managed partial implementation, I’ve achieved full graceful shutdown of the nodes but was unable to make the Coordinator also shut down once all nodes had stopped.

The way I achieved this was by adding an extra argument when starting the node.

A computer screen with colorful text

Description automatically generated

This takes in char y or n (although the way its implemented it only looks for a y)



This will pass true or false to the node depending on if the last arg is y or not.

A screen shot of a computer

Description automatically generatedIt then sets the value of this to a variable called n\_shutdown which is used to tell the node if it should initiated a shutdown or not. When the node with the shutdown call finishes its critical section it will close all the sockets and create a new file called “SHUTDOWN\_FLAG”

This node will then close its self using System.exit(0);

When another node goes to request a token from the coordinator it will first check of the SHUTDOWN\_FLAG file exists and if it does will print a message saying shutdown request received and printing its nodeID saying it has shutdown before running System.exit(0);

A screen shot of a computer

Description automatically generated

This stops the node from re-entering the critical section which would cause a crash as the sockets have been closed.

I attempted to implement a tracker that would count how many active node instances there are and when there are none it would close the Coordinator down too thus completing the full graceful shutdown but my attempts were unsuccessful, the program would still run normally but the coordinator shutdown would never occur thus making the code redundant so I removed it.

## Non-implemented advanced features

I did not implement the other 3 advanced features although I have ideas for how they could be implemented.

### Priority discipline request queue

Firstly, modify the Coordinator’s request handling to support priority levels. This would assign a higher priority to the critical process then use a PriorityQueue to manage token requests to ensure that a critical process request is handled before all others. Then include a priority in a node when it requests a token so when the coordinator receives this information it knows what order to issue tokens in. this would allow nodes with higher priority to jump ahead of the queue.

### Modified nodes that can deal with coordinator shutdown (or crash)

This feature I assume would have a check at each point in the nodes process (get token, start critical process, return token) that if a connection cannot be made to the coordinator it will wait a certain amount of time before then retrying, this would allow for the coordinator to be restarted and have the nodes then reconnect and continue without having to restart every node.

# Class diagram

As the diagram shows the Node Class has a constructor and a main method as it is used to start an instance of the Node, the Coordinator is much the same as it starts a Coordinator instance. C\_mutex ,C\_receiver and C\_buffer are all started by the coordinator. C\_Connection\_r is used to receive communication from a Node instance that is the passed to the C\_receiver class. While the Logger class is not directly connected to any other class it does have methods within it that are called in both the Node class and C\_mutex class.

# Code listings

## Coordinator.java

import java.net.\*;  
import java.util.Set;  
import java.util.concurrent.ConcurrentHashMap;  
  
  
public class Coordinator {  
  
  
  
 public static void main (String args[]){  
 int port = 7000;  
  
 Coordinator c = new Coordinator ();  
 Logger.*resetLogFile*();  
  
 try {  
 InetAddress c\_addr = InetAddress.*getLocalHost*();  
 String c\_name = c\_addr.getHostName();  
 System.*out*.println ("Coordinator address is "+c\_addr);  
 System.*out*.println ("Coordinator host name is "+c\_name+"\n\n");  
  
 }  
 catch (Exception e) {  
 System.*err*.println(e);  
 System.*err*.println("Error in coordinator");  
 }  
  
 // allows defining port at launch time  
 if (args.length == 1) port = Integer.*parseInt*(args[0]);  
  
  
 // Create and run a C\_receiver and a C\_mutex object sharing a C\_buffer object  
 C\_buffer cb = new C\_buffer();  
 C\_receiver cr = new C\_receiver(cb,port);  
 C\_mutex cm = new C\_mutex(cb,port);  
 cm.start();  
 cr.start();  
  
 }  
  
}

## C\_mutex.java

import java.io.RandomAccessFile;  
import java.net.\*;  
import java.util.Set;  
import java.util.concurrent.ConcurrentHashMap;  
  
public class C\_mutex extends Thread{  
 C\_buffer buffer;  
 Socket s;  
 int port;  
  
 // ip address and port number of the node requesting the token.  
 // They will be fetched from the buffer  
 String n\_host;  
 int n\_port;  
 int lastBufferSize = -1;  
  
 public C\_mutex (C\_buffer b, int p){  
 buffer = b;  
 port = p;  
 }  
  
 public void run(){  
 try{  
 // >>> Listening from the server socket on port 7001  
 // from where the TOKEN will be returned later.  
 ServerSocket ss\_back = new ServerSocket(7001);  
  
  
 while (true){  
  
 //Didn't put a print for debugging here as it would spam the console, moved it to after the buffer size check.  
  
 // if the buffer is not empty  
 if (buffer.size() !=0) {  
  
 System.*out*.println("C:mutex Buffer size is " + buffer.size());  
 buffer.show();  
 // >>> Getting the first (FIFO) node that is waiting for a TOKEN form the buffer  
 // Type conversions may be needed.  
  
 n\_host =(String) buffer.get();  
 n\_port = Integer.*parseInt*((String) buffer.get());  
  
 if(buffer.size() !=lastBufferSize) { // only writes to the file if the buffer size has changed since last write.  
 Logger.*getInstance*("log.txt").log(" Buffer size is :" + buffer.size() + " at " + new java.util.Date().toString());  
 lastBufferSize = buffer.size();  
 }  
  
  
 // >>> \*\*\*\* Granting the token  
 //  
 try{  
 s = new Socket(n\_host,n\_port);  
 System.*out*.println("token granted to the node on port : "+n\_port);  
 }  
 catch (java.io.IOException e) {  
 System.*out*.println(e);  
 System.*out*.println("CRASH Mutex connecting to the node for granting the TOKEN" + e);  
 }  
  
  
 // >>> \*\*\*\* Getting the token back  
 try{  
 ss\_back.accept();  
 System.*out*.println("C:Mutex received the token back from the node");  
 s.close();  
 }  
 catch (java.io.IOException e) {  
 System.*out*.println(e);  
 System.*out*.println("CRASH Mutex waiting for the TOKEN back" + e);  
 }  
  
  
 }// endif  
 }// endwhile  
 }catch (Exception e) {System.*out*.print(e);}  
  
 }  
}

## C\_buffer.java

import java.util.\*;  
  
public class C\_buffer {  
  
 private Vector<Object> data;  
  
 public C\_buffer (){  
 data = new Vector<Object>();  
 }  
  
 public int size(){  
 return data.size();  
 }  
  
 public synchronized void saveRequest (String[] r){  
 data.add(r[0]);  
 data.add(r[1]);  
 }  
  
 public void show(){  
 for (int i=0; i<data.size();i++) {  
 System.*out*.print(" " + data.get(i) + " ");  
 }  
 System.*out*.println(" ");  
 }  
  
 public void add(Object o){  
 data.add(o);  
 }  
  
 synchronized public Object get(){  
 Object o = null;  
  
 if (data.size() > 0){  
 o = data.get(0);  
 data.remove(0);  
 }  
 return o;  
 }  
}

## C\_receiver.java

import java.io.IOException;  
import java.net.\*;  
  
public class C\_receiver extends Thread{  
  
 private C\_buffer buffer;  
 private int port;  
 private ServerSocket s\_socket;  
 private Socket socketFromNode;  
 private C\_Connection\_r connect;  
  
 public C\_receiver (C\_buffer b, int p){  
 buffer = b;  
 port = p;  
 }  
  
 public void run () {  
  
 // >>> create the socket the server will listen to  
 try {  
 s\_socket = new ServerSocket(port);  
 System.*out*.println("Waiting for connection...");  
 } catch (IOException e) {  
 System.*err*.println("Can't create server socket: " + e);  
 System.*exit*(1);  
 }  
  
 while (true) {  
 try{  
 // >>> get a new connection  
 socketFromNode = s\_socket.accept();  
 System.*out*.println ("C:receiver Coordinator has received a request ...") ;  
  
 // >>> create a separate thread to service the request, a C\_Connection\_r thread.  
 connect = new C\_Connection\_r(socketFromNode, buffer);  
 connect.start();  
 }  
 catch (java.io.IOException e) {  
 System.*out*.println("Exception when creating a connection "+e);  
 }  
  
 }  
 }//end run  
}

## C\_Connection\_r.java

import java.net.\*;  
import java.io.\*;  
// Reacts to a node request.  
// Receives and records the node request in the buffer.  
//  
public class C\_Connection\_r extends Thread{  
  
 // class variables  
 C\_buffer buffer;  
 Socket s;  
 InputStream in;  
 BufferedReader bin;  
  
 public C\_Connection\_r(Socket s, C\_buffer b){  
 this.s = s;  
 this.buffer = b;  
 }  
  
 public void run() {  
 final int NODE = 0;  
 final int PORT = 1;  
  
 String[] request = new String[2];  
  
 System.*out*.println("C:connection IN dealing with request from socket "+ s);  
 try {  
  
 // >>> read the request, i.e. node ip and port from the socket s  
 // >>> save it in a request object and save the object in the buffer (see C\_buffer's methods).  
  
 in = s.getInputStream();  
 bin = new BufferedReader(new InputStreamReader(in));  
  
 request[NODE] = bin.readLine();  
 request[PORT] = bin.readLine();  
 this.buffer.saveRequest(request);  
  
 s.close();  
 System.*out*.println("C:connection OUT received and recorded request from "+ request[NODE]+":"+request[PORT]+ " (socket closed)");  
  
 }  
 catch (java.io.IOException e){  
 System.*out*.println(e);  
 System.*exit*(1);  
 }  
 //buffer.show();  
  
 }  
}

## Node.java

import java.net.\*;  
import java.io.\*;  
import java.util.\*;  
  
public class Node {  
  
 private Random ra;  
 private Socket s;  
 private PrintWriter pout = null;  
 private ServerSocket n\_ss;  
 private Socket n\_token;  
 String c\_host = "127.0.0.1";  
 int c\_request\_port = 7000;  
 int c\_return\_port = 7001;  
 String n\_host = "127.0.0.1";  
 String n\_host\_name;  
 int n\_port;  
 boolean n\_shutdown;  
  
 private static final String *SHUTDOWN\_FLAG* = "shutdown.flag";  
  
  
 public Node(String nam, int por, int sec, boolean shutdown) throws InterruptedException {  
 ra = new Random();  
 n\_host\_name = nam;  
 n\_port = por;  
 n\_shutdown = shutdown;  
  
  
 System.*out*.println("Node " + n\_host\_name + ":" + n\_port + " of DME is active ....");  
  
 String nodeId = n\_host\_name + ":" + n\_port; // Unique ID for the node  
  
  
 if(new File(*SHUTDOWN\_FLAG*).exists()){  
 new File(*SHUTDOWN\_FLAG*).delete();  
 }  
  
 try {  
 n\_ss = new ServerSocket(n\_port);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
  
 while (true) {  
  
 int min = sec;  
 int max = 5000;  
 int sleepTime = ra.nextInt((max - min) + 1) + min;  
 Thread.*sleep*(sleepTime);  
  
 //formats the sleepTime into seconds and milliseconds purely to make the output more readable.  
 int seconds = sleepTime / 1000;  
 int milliseconds = sleepTime % 1000;  
 String sleepTimeOutput = String.*format*("%d.%03d", seconds, milliseconds);  
 System.*out*.println("Sleeping for: " + sleepTimeOutput + " Seconds.");  
  
  
 try {  
  
 //stops the node from starting a new request if the shutdown flag exists  
 if(new File(*SHUTDOWN\_FLAG*).exists()){  
 System.*out*.println("Global node shut down request recieved...");  
 System.*out*.println("Node " + nodeId + " has shut down gracefully.");  
 System.*exit*(0);  
 }  
 // \*\*\*\* Send to the coordinator a token request.  
 // send your ip address and port number  
 n\_token = new Socket(c\_host, c\_request\_port);  
 pout = new PrintWriter(n\_token.getOutputStream(), true);  
 pout.println(n\_host);  
 pout.println(n\_port);  
 System.*out*.println("\*\*- Request Token -\*\*");  
 System.*out*.println(n\_host + " is requesting a token on port " + c\_request\_port);  
 n\_token.close();  
  
 // \*\*\*\* Then Wait for the token  
 // Accept the token and print suitable messages  
 n\_ss.accept();  
 System.*out*.println("\*\*-Critical Section start for port : " + n\_port + " : at [" + new Date().toString() + "]-\*\*"); //prints time the critical section started  
 System.*out*.println("Token received by " + n\_host + " on port: " + c\_request\_port);  
 Logger.*getInstance*("log.txt").log("\*\*-Critical Section start for port : " +n\_port+" : at [" + new Date().toString() + "]-\*\*");  
  
 // \*\*\*\* Sleep for a while  
 // This simulates the critical session  
 Thread.*sleep*(sleepTime);  
  
 // \*\*\*\* Return the token  
 // Create a new socket for returning the token and print suitable messages, also considering communication failures  
 s = new Socket(c\_host, c\_return\_port);  
 System.*out*.println("\*\*-Critical Section end for port : " + n\_port + " : at [" + new Date().toString() + "]-\*\*"); //prints time the critical section ended  
 System.*out*.println(n\_host + " is returning a token on port " + c\_return\_port);  
 Logger.*getInstance*("log.txt").log("\*\*-Critical Section end for port : " + n\_port + " : at [" + new Date().toString() + "]-\*\*");  
  
 // runs if the shutdown flag is received  
 if(n\_shutdown){  
 try {  
 //prints that a shutdown request has been received and closes all sockets  
 System.*out*.println("Request to shutdown received....");  
 n\_ss.close();  
 s.close();  
 pout.close();  
 //creates the shutdown flag file  
 File shutdownFlag = new File(*SHUTDOWN\_FLAG*);  
 try{  
 if(shutdownFlag.createNewFile()){  
 System.*out*.println("Shutdown flag created");  
 }else  
 System.*out*.println("Shutdown flag already created");  
 }catch (IOException e){  
 System.*out*.println(e);  
 }  
 System.*exit*(0);  
 }catch (IOException e){  
 System.*out*.println(e);  
 }  
 }  
  
 } catch (IOException e) {  
 System.*out*.println(e);  
 System.*exit*(1);  
 }  
 try {  
 s.close();  
  
 } catch (IOException e) {  
 System.*out*.println("Something went wrong closing the sockets " + e);  
 }  
  
 }  
 }  
  
  
 public static void main(String args[]) throws InterruptedException {  
 String n\_host\_name = "";  
 int n\_port;  
  
  
 // port and millisec (average waiting time) are specific of a node  
 if ((args.length < 2) || (args.length > 3)) {  
 System.*out*.print("Usage: Node [port number] [millisecs] [y / n]");  
 System.*exit*(1);  
 }  
  
 // get the IP address and the port number of the node  
 try {  
 InetAddress n\_inet\_address = InetAddress.*getLocalHost*();  
 n\_host\_name = n\_inet\_address.getHostName();  
 System.*out*.println("node hostname is " + n\_host\_name + ":" + n\_inet\_address);  
 } catch (UnknownHostException e) {  
 System.*out*.println(e);  
 System.*exit*(1);  
 }  
  
 n\_port = Integer.*parseInt*(args[0]);  
 System.*out*.println("node port is " + n\_port);  
 Node n = new Node(n\_host\_name, n\_port, Integer.*parseInt*(args[1]),"y".equalsIgnoreCase(args[2]));  
  
 }  
  
  
}

## Logger.java

import java.io.IOException;  
import java.io.RandomAccessFile;  
import java.util.Date;  
  
// Defines a Logger class for logging time stamps of the start and finish of critical section and buffer size to a file.  
public class Logger {  
 private final String filePath;  
 private static Logger *instance*;  
  
 private Logger(String filePath) {  
 this.filePath = filePath;  
 }  
  
 public static synchronized Logger getInstance(String filePath) {  
 // If the instance is null, create a new Logger instance with the provided file path.  
 if (*instance* == null) {  
 *instance* = new Logger(filePath);  
 }  
 // Return the existing/newly created Logger instance.  
 return *instance*;  
 }  
  
 // Synchronized method to log a message to the file. Synchronization ensures thread safety.  
 public synchronized void log(String message) {  
 try (RandomAccessFile log = new RandomAccessFile(filePath, "rw")) {  
 // Move to the end of the file to append the new message.  
 log.seek(log.length());  
 // Write the message along with the current timestamp to the file.  
 log.writeBytes(System.*lineSeparator*() + message + " at " + new Date().toString());  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 //Method used to clear the log file content when the Coordinator is started  
 public static void resetLogFile() {  
 try (RandomAccessFile log = new RandomAccessFile("log.txt", "rw")) {  
 // Set the length of the file to 0, clearing it.  
 log.setLength(0);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}