CSCU9V6 Concurrent and Distributed Systems

Assignment 2: Distributed Systems

Student ID: 2118616

Date: Spring 2024

Generative AI statement

Use of AI

I acknowledge that:

No content generated by AI technology has been knowingly presented as my own work in this submission.

Presentation, structure, and proofreading

I acknowledge that:

I did not use AI technology to assist in structuring or presenting my submission.

Table of Contents

Table of Figures

Overview

This report relates to assignment 2 of CSCU9V6 Operating Systems Concurrency and Distribution. The report contains a discussion of assumptions that were made along with a description of my solution. The code listing is also included at the end of this document.

Distributed Systems

The project involves developing a solution that ensures mutual exclusion of processes in a distributed system. The topic of distributed systems is complex (for me), but a succinct definition is offered by A.S. Tanenbaum and M.v. Steen:

> *A distributed system is a collection of independent computers that appears to its users as a single coherent system.* [1]

With the now far improved local and wide area networks, it is now a reasonable expectation for independent computers – or hosts, or machines, or nodes – to cooperate with other machines and thus the sharing of resources is now an essential consideration. Say, for example, a flight is scheduled to depart Edinburgh and is bound for New York. The air traffic control office in Edinburgh must communicate the flight's status to New York's air traffic control office, whilst at the same time New York must update Edinburgh on their readiness to receive the flight. I am no expert in air traffic control, but it is not difficult to see how essential it is that air traffic control offices have the ability to update other offices in different locations and time zones **in real time**; it is also crucial that the information used in the air traffic management system is **consistent**. A goal of using a distributed system is that it should not be apparent to a user that another user is accessing the same (shared) resource; this is referred to as concurrency transparency [1]. If, for example, the distributed system allows two air traffic control offices to update a shared resource (e.g. a database) at the same time, this can result in inaccurate data, and inaccurate data could result in unwanted consequences. Figure 1 illustrates the basic communication between a "collection of processors" [2].

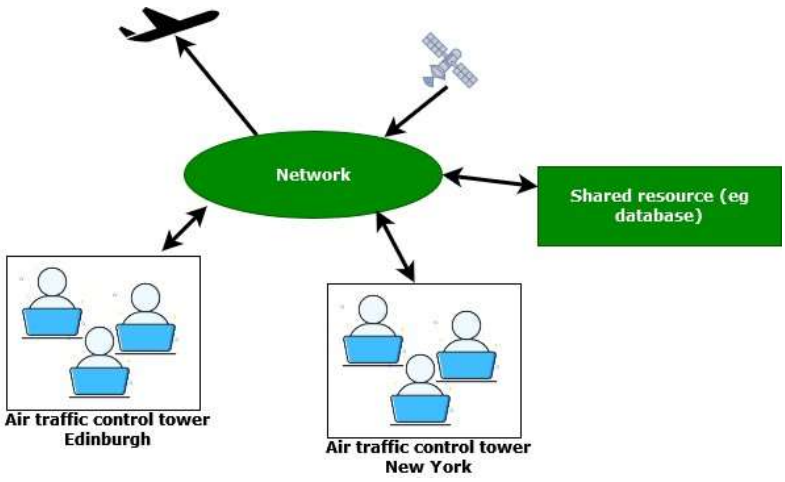

Figure 1 Basic illustration of distributed system for air traffic control

Though hardware forms a component of a distributed system, there are many aspects which impact upon decision-making throughout the software design and implementation process. We must decide what should happen in the **event of failure** of either a single node, or many nodes; the system should preferably recover from failure without a user noticing. Other limitations of a distributed system are:

- the absence of a **global clock** – communications may seem instant to a human user, but messages can only travel so fast, and as such there will always be delays as a result of the time it takes for messages to travel. No two clocks will have the exact same time, thus clocks in a distributed system are asynchronous;
- the lack of **shared memory** – independent computers will each have their own physical memory, so in any distributed system there will be no shared memory and this can present issues relating to scalability. For example, if the number of users using the distributed system grows and there is only one server in the system, an increase in communication traffic can overload the server;
- **latency** – data is expected to travel fast, at the speed of light, in fact. However, in a distributed system, data has to travel from one independent computer to the server, and back, [3] resulting in a delay that varies with each communication. This can potentially cause issues with the accuracy of data as any changes or updates to data will not propagate through the whole system instantaneously; and,
- the **consistency of the system's state** – all of the above contribute to the difficulty in establishing a state that is maintained through the overall system.

There are various logical architectures that can address the faults described above. For this assignment, I will focus on one-to-one communication where sockets – which are commonly used in server-client models – are used as channels of communication, and where there is no shared memory.

Regardless of architecture, mutual exclusion is of critical importance if requirements suggest that various processes must access a shared resource. The idea of achieving mutual exclusion is to allow only a single process at any one time access the shared resource, or, perform their critical section. Consequently, the risk of leaving data in an inaccurate and inconsistent state is far reduced owing to mutual exclusion. There are various approaches and algorithms that can ensure mutual exclusion in a system, but for the purposes of this module, I will focus on and discuss token-based algorithms.

Basic Problem

The central problem of this assignment is to develop a solution that implements Distributed Mutual Exclusion (DME) based on sockets. It adopts a centralised approach using a coordinator node which passes a token to each node that sends a token request. The token

is passed to only one node at a time. When in possession of the token, that node can proceed to execute its critical section. After completion of the node's critical section, the node returns the token back to the coordinator node.

In my solution, I make the following assumptions:

- the system will be run on one machine
- the system may have any number of processes going;
- each process will reside in different Java Virtual Machines;
- each process executes a critical section;
- each process's critical section requires mutual exclusion; and,
- the system will use a single coordinator node.

Static Model

From the starter code provided, it is possible to devise a class diagram with enough useful information to develop the basic solution. Figure 2 provides this class diagram which helps in understanding the current structure and architecture of the system, and also provides a solid starting point for building upon and modifying the basic solution.

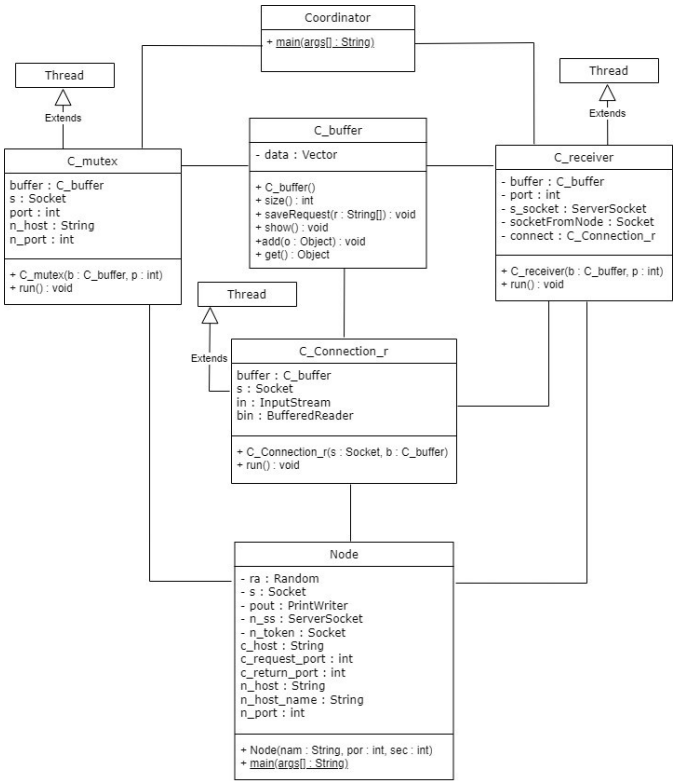

Figure 2 Initial class diagram

From this diagram, we can see that both the C_mutex and C_receiver classes interact with the C_buffer class. It is apparent that the C_mutex class extracts token requests from the buffer, and issues the token to the next node waiting to execute its critical section. To fulfil a node's request, I make use of the get() method in the C_buffer class; to issue the token, we must connect to the node through a Socket s by using the requesting node's host name and the node's port. The final responsibility of the C_mutex class is receiving the token back from the node. To enable communication from the node back to the coordinator, the ServerSocket method accept() is used. This is used because it is important for the coordinator to block whilst it is waiting on the token being returned by the node, and the accept() method achieves this as it

blocks until a connection is established. In other words, by using accept() when getting the token back, the coordinator will not move on to execute any other task:

```
while (true){
// >>> Print some info on the current buffer content for debugging purposes.
// >>> please look at the available methods in C_buffer

    // if the buffer is not empty
    if (buffer.size() > 0) {
        //extract the request
        try{
            n_host = (String) buffer.get();
            n_port = Integer.parseInt((String)buffer.get());
        } catch (Exception e){
            System.err.println("C_mutex: couldn't convert request: " + e);
        }//end try catch

        // >>>  **** Granting the token
        try{
            s = new Socket(n_host, n_port);
            System.out.println("C_mutex: Coordinator granted token to node " + n_host + ":" + n_port);
            s.close();
        } catch (java.io.IOException e) {
            System.out.println(e);
            System.out.println("CRASH Mutex connecting to the node for granting the TOKEN" + e);
        }//end try catch

        //  >>>  **** Getting the token back
        try{
            // THIS IS BLOCKING !
            //ss_back.accept will block until it establishes a connection
            //A connection will only be established when a node has returned a token
            s = ss_back.accept();
            System.out.println("C:mutex: Token returned by node " + n_host + ":" + n_port);
        } catch (java.io.IOException e) {
            System.out.println(e);
            System.out.println("CRASH Mutex waiting for the TOKEN back" + e);
        }//end try catch
    }// endif
}// endwhile
```

Getting request details

Connect to node through Socket s using request details

System won't proceed until token has been returned.

Figure 3 C_mutex.java

To summarise, in the basic solution the C_mutex class:

- reads a node's request details, thus decreases the buffer size by 1;
- sends the token to the node associated with the request, allowing that node to execute its critical section; and,
- receive the token back from the node once it has completed its critical section.

While C_mutex removes data from the buffer, the class C_receiver registers an incoming request. This class is a little like a phone ringing to alert you someone has initiated a call. When a node sends a token request to the coordinator, the C_receiver class is responsible for receiving the node's request through the socket socketFromNode, and then initializes a C_Connection_r object using that socket and of course the buffer. When this connection is established, this is similar to you picking up the phone and saying "Hello?". The C_Connection_r class starts a separate thread to service the request, and ensures consistency when adding the request to the buffer through the use of the keyword synchronized in the method saveRequest() in C_buffer. The usage of synchronized in the C_buffer.saveRequest() method is absolutely crucial because we are not dealing with just

one machine – this is a distributed system; there are numerous nodes that all want a piece of the buffer. The buffer object is manipulated by multiple threads, so we must define a mechanism for preventing these threads from messing each other and any shared resources up. If the synchronized keyword was omitted in C_buffer.saveRequest(), or in C_buffer.get(), then there is absolutely no guarantee that one thread will be able to see the changes another thread made to the shared resource, resulting in the race condition of multiple threads accessing and updating the buffer simultaneously. The use of the synchronized keyword is a sensible strategy to contribute to writing code which is thread-safe, therefore helping to keep data consistent and accurate.

```
public void run () {
    // >>> create the socket the server will listen to
    try {
        s_socket = new ServerSocket(port);
    } catch (IOException e) {
        System.out.println("Incoming connection failed: " + e.getMessage());
    }//end try catch

    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 try catch
    }//end while
}//end run
```

Node wants the token. Phones C_receiver to send the request.

C_receiver notifies C_Connection_r that Node has made a request, and wants to add details to the shared buffer.

Figure 4 C_receiver.java

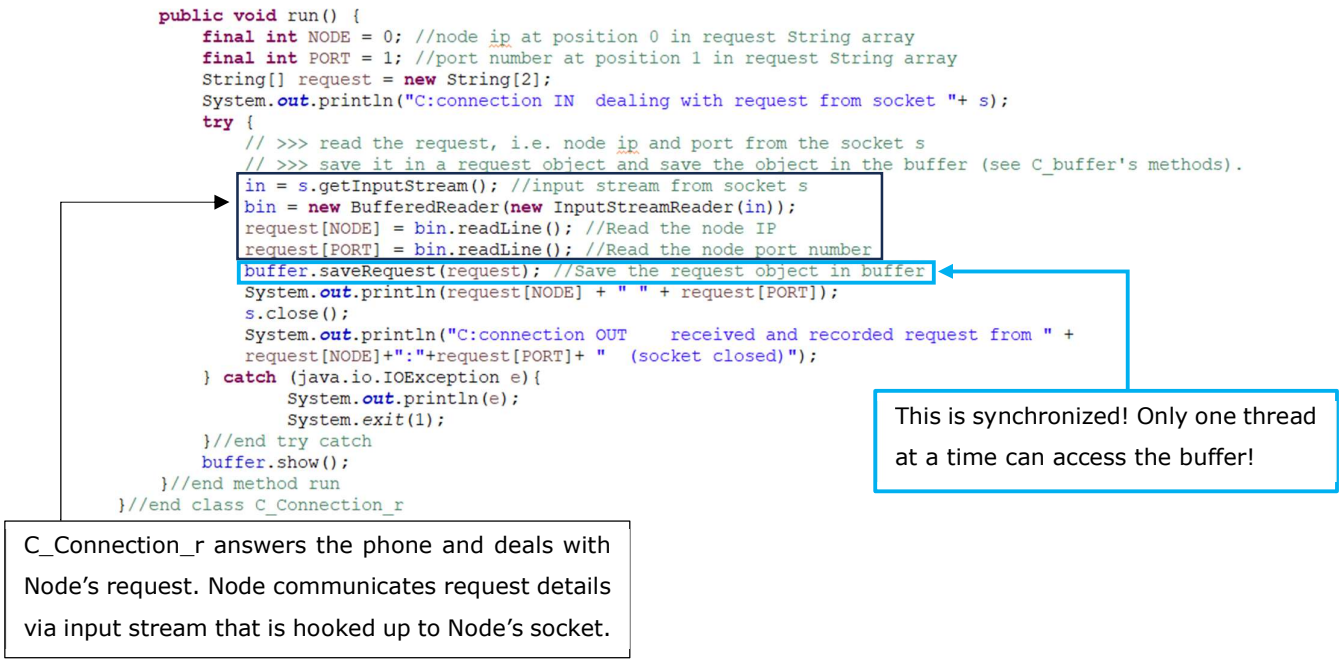

Figure 6 C_Connection_r.java

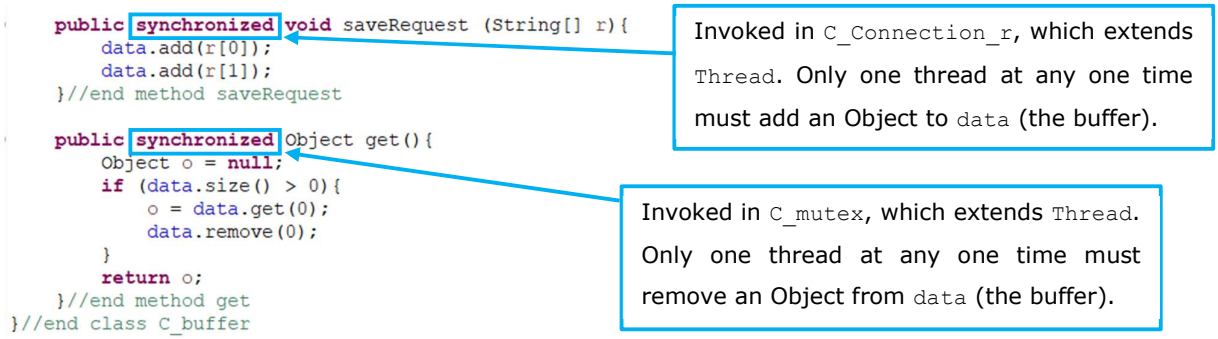

Figure 5 C_buffer.java

Figure 5 shows a request being read through a BufferedReader. Although I was guided to read in data using this (the class variables InputStream in and BufferedReader bin are class variables in C_Connection_r in the starter code), it brings the advantage of efficiency whilst achieving what we need it to. The BufferedReader provides the ability to read in from a character stream [4], and it does this by storing the data from the specified input stream in buffer memory. It will start by reading a chunk of characters, then stores that in the buffer memory which it has created. If and/or when the system continues reading in data through this BufferedReader, the data only has to be retrieved from this buffer memory and not from the input stream itself. Once the buffer memory is empty, it repeats the process and performs a read operation on another chunk of data, storing this new chunk in the buffer memory [5]. This results in a faster and more efficient way to read in

data. Even if these class variables were not included in our starter code, I would still have opted to use a BufferedReader.

With the centralized approach taking shape, some gaps had to be filled in in Node.java. Obviously, a Node can only perform its critical section if it can receive the token, and to receive the token the Node must be able to receive communication from the coordinator. As such, the ServerSocket is initialized using the Node's port number:

```
try { n_ss = new ServerSocket(n_port); }
catch (IOException e) { e.printStackTrace(); }
```

Figure 7 Node.java - n_ss ServerSocket

Now, when a Node is running (while(true)), I specify that the Node's thread should sleep for a random duration. This duration is determined using the Random object ra, and the sec value provided at launch is used to set the upper bound. By incorporating a sleep here, we allow some time for connections to be closed/established. The system doesn't experience the same latency as a distributed system that is geographically far apart, so including Thread sleeps helps to avoid any failures when the system is running.

The Node class has four primary objectives. These are:

- request the token;
- receive the token;
- perform critical section; and,
- return the token.

To send a token request, a Socket is required to provide the Node with a connection to the coordinator. This socket takes the arguments c_host and c_request_port, representing the name and port on which the coordinator is listening:

```
try {
    // **** Send to the coordinator a token request.
    // send your ip address and port number
    s = new Socket(c_host, c_request_port);
    pout = new PrintWriter(s.getOutputStream(), true);
    pout.println(n_host); //ip
    pout.println(n_port); //port
    System.out.println("Node " + n_host_name + ":" + n_port +
            " requested token from coordinator.");
    s.close();
```

Figure 8 Node.java requesting the token

The Node may well have to wait for the token, if the coordinator has issued it to another Node and is yet to receive the token back. This is another example of the ServerSocket method accept() blocking until a connection is established. This method will force the Node to wait until it can connect to the coordinator to receive the token:

```
// **** Then Wait for the token
// Print suitable messages
n_token = n_ss.accept();
System.out.println("Node " + n_host_name + ":" + n_port +
        " received token from coordinator.");
n_token.close();
```

Figure 9 Node.java receiving the token

Once this operation is complete, the Node is in possession of the token. In other words, it has permission to enter its critical section, where it can access and update shared resources such as a database. In this case, the Node simply sleeps for a random duration to simulate a critical section:

```
// **** Sleep for a while
// This is the critical session
System.out.println("Node " + n_host_name + ":" + n_port + "entering critical section");
Thread.sleep(ra.nextInt(sec));
System.out.println("Node " + n_host_name + ":" + n_port + " exiting critical section");
```

Figure 10 Node.java critical section

On leaving its critical section, the Node must return the token through a new Socket using the coordinator's host name along with the coordinator's return port. Assuming there are no communication failures, connecting through this Socket signals the return of the token:

```
// **** Return the token
// Print suitable messages - also considering communication failures
n_token = new Socket(c_host, c_return_port);
System.out.println("Node " + n_host_name + ":" + n_port + " returning token to coordinator.");
n_token.close();
```

Figure 11 Node.java returning the token

This is the end of the routine for the Node and whilst running with no errors or failures, a Node will simply repeatedly request a token, perform critical section, and return the token.

Figure 13 Distributed system set-up is a screenshot of the basic solution before running. There is one instance of Coordinator, and three instances of Node. Figure 12 Distributed system centralised approach running shows the running of the system.

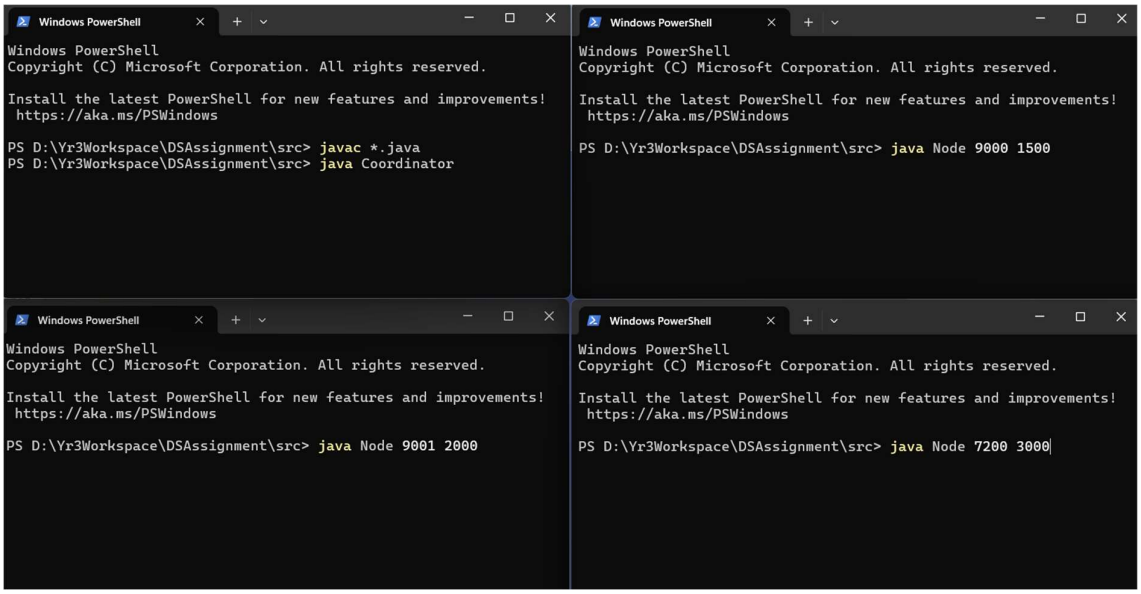

Figure 13 Distributed system set-up

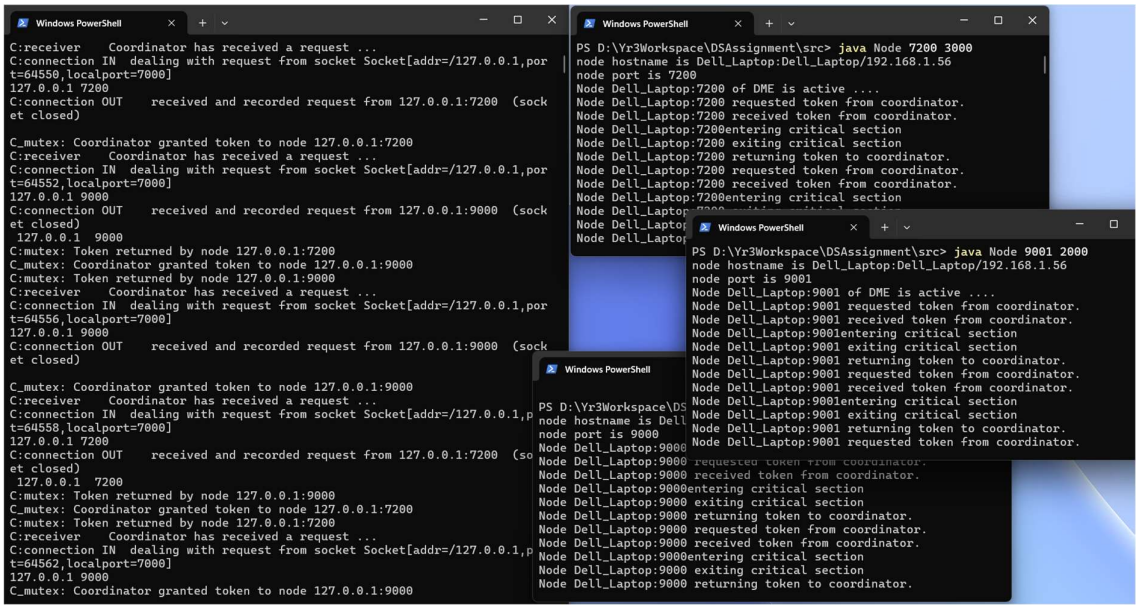

Figure 12 Distributed system centralised approach running

Dynamic Model

While a class diagram forms part of designing the static model of a system, we can also communicate what and/or how system operations are triggered. The class diagram is an effective technique of illustrating how the system should behave independent of time, but it is also important to understand how the system is expected to behave at runtime. Figure 14 shows a sequence diagram of the basic solution. That is, the basic communication links between objects in the system.

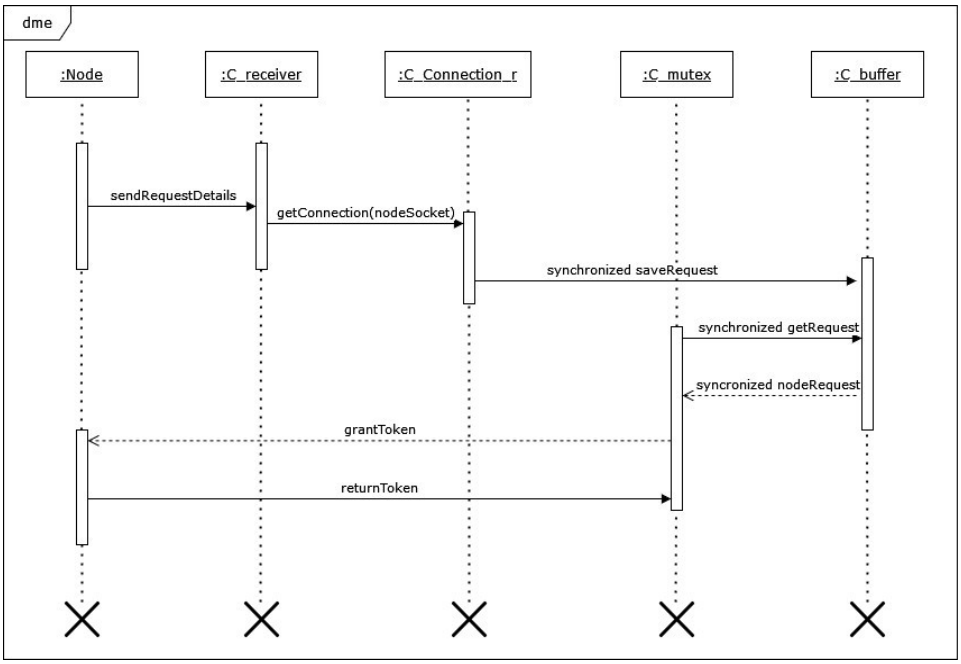

Figure 14 Sequence diagram showing basic solution

The benefit of creating a sequence diagram here is that it provides a method of visualizing asynchronous and synchronous messages, so we can see clearly what operations are synchronized. The above diagram shows the communications of a single Node making a single request:

1. Node sends a request to the Coordinator (C_receiver);
2. C_receiver gets a new connection to receive the node's request, and starts a new thread to service the request (C_Connection_r);
3. the request is added to storage in the buffer. This is a synchronized operation because only one thread at any time should be able to manipulate the data stored in the buffer;
4. the buffer now contains a request. The Coordinator (C_mutex) asks the buffer for the request. The get() method in the buffer returns an Object, which is the request details. This is a synchronized operation because it accesses a shared resource;

5. the buffer returns the request, and the Coordinator (C_mutex) issues the token to the requesting Node;
6. the Node will perform its critical section, which is not depicted above as the critical section is simply the Node taking a short nap; and,
7. once the Node has slept for a while, it returns the token to the Coordinator (C_mutex). The steps are repeated as long as the system is running.

Advanced Features

File Logging

To implement a file logging mechanism, I considered creating a new class to handle file operations. However, this caused undesired behaviour during testing where Nodes would log their start of critical section and token return only once. I made the decision to simply include methods in the relevant classes to handle the file logging mechanism, in a bid to save time that I wanted to use to focus on the remaining advanced features. The following classes include two class level variables and three methods to handle file logging: Node.java, C_Connection_r.java, and C_mutex.java. The class Coordinator.java takes care of clearing the file when the system is launched. I log the following events to file:

- Node.java:
  - Node name and port and the **start of critical section** along with a timestamp and duration; and,
  - Node name and port and **returning the token** with a timestamp.
- C_Connection_r.java:
  - **Requests received** with request details; and
  - **Buffer size** after request is saved to buffer.
- C_buffer.java:
  - A statement indicating a Node's **priority has been boosted**. This was mainly for debugging purposes, but I have retained this log as I think it makes it clear that avoiding starvation mechanism is in place.
- C_mutex.java:
  - The **token being issued** to a Node, along with Node's name, port, and priority; and,
  - **Buffer size** after the token has been issued.

An example of the contents of the text file is as follows:

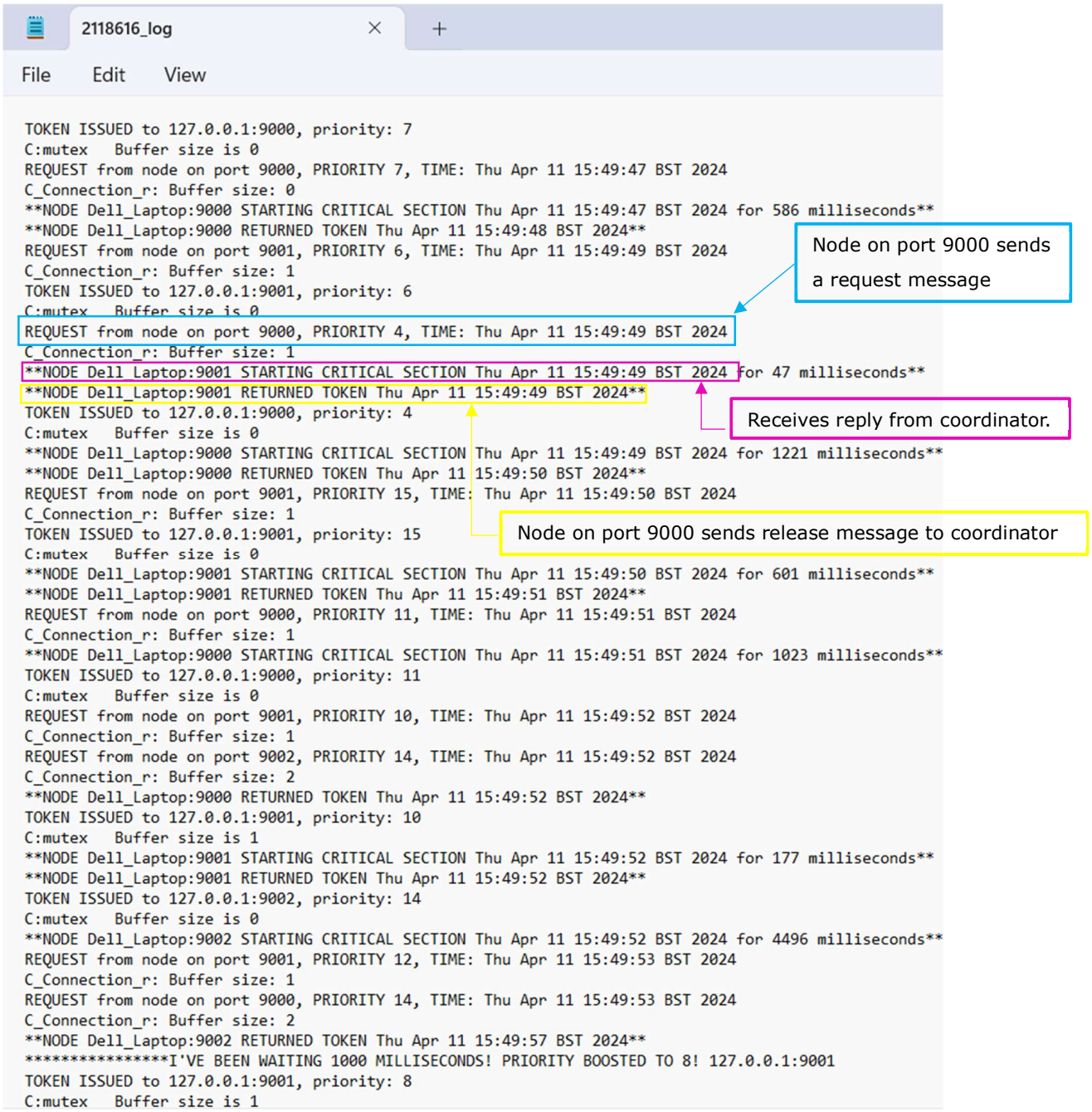

Figure 15 File logging text file

Priority Discipline and Starvation

To implement a priority discipline I made a number of modifications/additions to C_buffer.java. One major advantage of object oriented programming is abstraction. This can give developers the ability to use a module of code without having to know or see its internal implementation. An example of this is using the PrintWriter's method println() to log to file. When considering how to implement a priority discipline in the system, I looked into what the java.util.concurrent package has to offer and noted the class PriorityBlockingQueue<E>. This is described as an "unbounded blocking queue" that "supplies blocking retrieval operations" [6]. To make use of a PriorityBlockingQueue to store node requests, the C_buffer class would have to be altered so that requests are stored in a PriorityBlockingQueue rather than a Vector. The class brings the advantage of being in the Java Collections Framework, making it straightforward to manipulate our data structure with methods like add(E e) and poll(). Furthermore, using PriorityBlockingQueue and implementing Comparator makes it simple to define that if the priority integer is smaller it has a higher integer.

The code snippet at Figure 16 shows that adopting this approach to implementing a priority discipline is very readable and straightforward.

```
@Override
public int compare(String[] request1, String[] request2) {
    //compare priorities
    int p1 = Integer.parseInt(request1[2]);
    int p2 = Integer.parseInt(request2[2]);
    //if p1 is smaller than p2, request1 has higher priority than request2
    //if p1 is larger than p2, request1 has lower priority than request2
    return Integer.compare(p1, p2);
}//end method compare
```

Figure 16 Smaller integers represent higher priorities

However, this is not the only way of creating a priority discipline. The Vector data structure is a dynamic data structure, and as it implements the interface List, there are a variety of methods for us to utilise. In using Vector, we still ensure a thread-safe implementation, unlike the ArrayList. Considering it is simple to implement a sorting algorithm, and that a Vector is a dynamic, thread-safe data structure, the PriorityBlockingQueue does not bring any advantage meaningful and noticeable enough to warrant replacing the Vector. The PriorityBlockingQueue versus Vector decision is not wholly important for the system being built in this context, but it was interesting to learn that this option exists.

In implementing the priority discipline, the issue of starvation comes into question. This means that lower priority processes can effectively sit waiting for the token indefinitely. A solution to this is aging. This technique involves "increasing the priority of processes that wait in the system for a long time" [2]. This is dealt with at the time of sorting the buffer according to Nodes' priority value. I added three new methods:

- an overloaded method to save a request that also contains a priority value and a timestamp:

```
/**
 * Overloaded method to save a request with a priority and timestamp.
 * @param r the request to save
 * @param priority the priority of the request
 * @param time the time the request was made
 */
public synchronized void saveRequest(String[] r, int priority, long time) {
    r[2] = Integer.toString(priority); //insert priority to String array
    r[3] = Long.toString(time); //save time with the request
    data.add(r); //add request r to buffer
}//end method saveRequest
```

Figure 17 C_buffer.java overloaded saveRequest() method

- a new method sortByPriority(), which sorts the requests into ascending order according to priority value. In other words, requests with smaller integers for priority values are brought to the front of the queue:

```
/**
 * Sorts data into ascending order.
 */
private void sortByPriority() {
    for(int i = 0; i < data.size()-1; i++) {
        for(int j = i + 1; j < data.size(); j++) {
            String[] r1 = (String[]) data.get(i);
            String[] r2 = (String[]) data.get(j);
            int p1 = Integer.parseInt(r1[2]);
            int p2 = Integer.parseInt(r2[2]);
            //if priority of process 2 is smaller than priority of process 1
            if(p2 < p1) Collections.swap(data, i, j); //make the swap
        }//end inner for
    }//end outer for
}//end method sortByPriority
```

Figure 18 Sorting the buffer so higher priority Nodes are brought forward

- a new get() method, getRequest(), which sorts the requests by priority and calculates the next request's waiting time, then compares the waiting time to the value of the class variable aging. If waitingTime exceeds aging, the request's priority value is boosted:

```
/**
 * Retrieves and removes a request from the queue.
 * @return
 */
public synchronized Object getRequest() {
    if(!data.isEmpty()) {
        //first sort the data according to priority (ascending because smaller integers = higher priority)
        sortByPriority();
        String[] request = (String[]) data.remove(0); //remove the first request
        if(request != null) {
            long currentTime = System.currentTimeMillis(); //get the current time
            long requestTime = Long.parseLong(request[3]); //get the request timestamp
            long waitingTime = currentTime - requestTime; //calculate how long request has been waiting
            if(waitingTime > aging) { //is waiting time more than default time (aging)?
                int priority = Integer.parseInt(request[2]); //then extract priority of the request
                int updatedPriority = priority - (int) (waitingTime / aging); //boost its priority.
                request[2] = Integer.toString(updatedPriority); //now insert updated priority into String request array
                print_writer.println("****************I'VE BEEN WAITING " + aging + " MILLISECONDS! PRIORITY BOOSTED TO "
                + request[2]+ "!" + " " + request[0]+":"+request[1]);
                System.out.println("****************I'VE BEEN WAITING " + aging + " MILLISECONDS! PRIORITY BOOSTED TO "
                + request[2]+ "!" + " " + request[0]+":"+request[1]);
            }//end if waitingTime > aging
        }//end if request not null
        return request;
    }//end if data not empty
    return null;
}//end method getRequest
```

Figure 19 Applying aging to avoid starvation of lower priority Nodes

This ensures that the requests are sorted just as C_mutex.java extracts a request from the buffer, and as a request is extracted the system repeatedly evaluates if a request's priority value should be boosted.

This does not fully implement the priority discipline, though. A few other additions were made, namely generating a priority value for all Nodes and then modifying the reading in and saving of requests, and adjusting the saving and reading of requests in C_Connection_r and C_mutex too. The Node class now contains an additional method, getPriority() (Figure 21). This is a simple method that uses the Random object to generate an integer between 0 and 15. In reality, the priority value would likely be set according to the importance of the process waiting, but for this assignment I simply need a way of assigning a priority value to each Node. This method is then called in sendRequestDetails():

```
/**
 * Method to generate a priority value for a Node.
 * @return a random integer between 1 and 15 (inclusive).
 */
private int getPriority() {
    return ra.nextInt(15)+1;
}//end method getPriority
```

Figure 21 Generating a priority value for a Node

```
/**
 * Method to send request details.
 */
private void sendRequestDetails() {
    try {
        pout = new PrintWriter(s.getOutputStream(), true);
        pout.println(n_host); //ip
        pout.println(n_port); //port
        pout.println(getPriority());
        pout.println(System.currentTimeMillis());//the timestamp of request
        System.out.println("Node " + n_host_name + ":" + n_port + " requested token from coordinator.");
    } catch (IOException e) {
        System.out.println("Error sending request: " + e);
        sleep(5000);
    }//end try catch
}//end method sendRequestDetails
```

Generating priority value

Figure 20 Sending priority value with request

The code in sendRequestDetails() previously resided directly in the Node's constructor method. To enhance readability, I have made an attempt to modularise the code here. The above code snippet also shows the timestamp associated with this Node's request being sent. I have included System.currentTimeMillis() directly in the pout.println() method in an effort to increase accuracy. The program does not need to locate and read any other method, so this is an effective way of trying to get the most exact timestamp.

These new details that are necessary for implementing a priority discipline that avoids starvation are then read in by C_Connection_r.java:

```
/**
 * Method to add request into the buffer.
 */
private void recordRequest() {
    buffer.saveRequest(request, Integer.parseInt(request[PRIORITY]), Long.parseLong(request[TIME]));
}//end method recordRequest

/**
 * Method to read from input stream from node
 */
private void readRequest() {
    try {
        //Read the node IP
        request[NODE] = bin.readLine();

        //Read the node port number
        request[PORT] = bin.readLine();

        //Read the node priority
        request[PRIORITY] = bin.readLine();

        //Read the node timestamp
        request[TIME] = bin.readLine();
    } catch (IOException ioe) {
        System.out.println("C_Connection_r couldn't read request details: " + ioe);
        System.exit(1);
    }//end try catch
}//end method readRequest
```

For priority discipline

To avoid starvation (aging)

Read from node's input stream

Figure 22 C_Connection_r.java adjustments for priority discipline avoiding starvation

Similarly, C_mutex.java requires some changes also:

```
/**
 * Method to get the highest priority node waiting for a token.
 */
private void getNode() {
    // >>>   Getting the first (highest priority) node that is waiting for a TOKEN form the buffer
    //       Type conversions may be needed.
    try{
            String[] request = (String[])buffer.getRequest();
            n_host = request[0];
            n_port = Integer.parseInt(request[1]);
            priority = Integer.parseInt(request[2]);
            time = Long.parseLong(request[3]);
    } catch (Exception e){
        System.err.println("C_mutex couldn't convert request: " + e);
    }//end try catch
}//end method getNode
```

A sort and waiting time check is performed here. Synchronized operation as only one thread at a time should access the shared resource (the buffer).

Figure 23 C_mutex.java adjustments for priority discipline avoiding starvation

To test that the priority discipline and starvation avoidance works, we can inspect the text file contents at Figure 15:

Node on port 9001 sends a request of priority 12

Another request comes in

```
REQUEST from node on port 9001, PRIORITY 12, TIME: Thu Apr 11 15:49:53 BST 2024
C_Connection_r: Buffer size: 1
REQUEST from node on port 9000, PRIORITY 14, TIME: Thu Apr 11 15:49:53 BST 2024
C_Connection_r: Buffer size: 2
**NODE Dell_Laptop:9002 RETURNED TOKEN Thu Apr 11 15:49:57 BST 2024**
***************I'VE BEEN WAITING 1000 MILLISECONDS! PRIORITY BOOSTED TO 8! 127.0.0.1:9001
TOKEN ISSUED to 127.0.0.1:9001, priority: 8
```

Figure 24 Text file showing priority discipline and aging in action

Node on port 9001 has priority boosted.

From the results above what we can see is that Node on port 9001 sends a request of priority 12 at 15:49:53. Another request is registered from a separate Node on port 9000 at the same time, but with a lower priority of 14. The token has been returned by Node on port 9002 4 seconds after these two requests are recorded. Then we can see that the Node on port 9001 currently waiting in the buffer has its priority boosted because it has been waiting longer than the given aging threshold (set to 1000ms for testing purposes!). The Node on port 9001 is then issued the token because its priority is higher than the other Node that is waiting.

Coordinator Failure

One issue that commonly arises in distributed mutual exclusion systems is the failure of nodes. I have made a few assumptions here:

- If the coordinator closes or crashes, a Node must try to re-establish connection 5 times.
- The Node to reach 5 reconnection attempts first will take over the role of coordinator.
- If a Node has sent a token request, the Node has 10 seconds to receive the token.
- If the token is not received within 10 seconds, the Node's request is abandoned and it is assumed connection to the coordinator is lost.

If the coordinator closes, a Node will throw a ConnectException. Within the while(true) loop of Node.java, a try catch statement accounts for this exception. If this exception is thrown, a class level variable reconnAttempts is incremented by 1. A message is then output to screen and a sleep is included to force the thread to wait before trying to re-establish a connection. Within this catch statement, I also check if the Node's reconnAttempts value exceeds 5. If this check returns true, a message is output to screen and the Node takes over the coordinator role using the c_request_port before breaking from the current iteration of the while(true) loop:

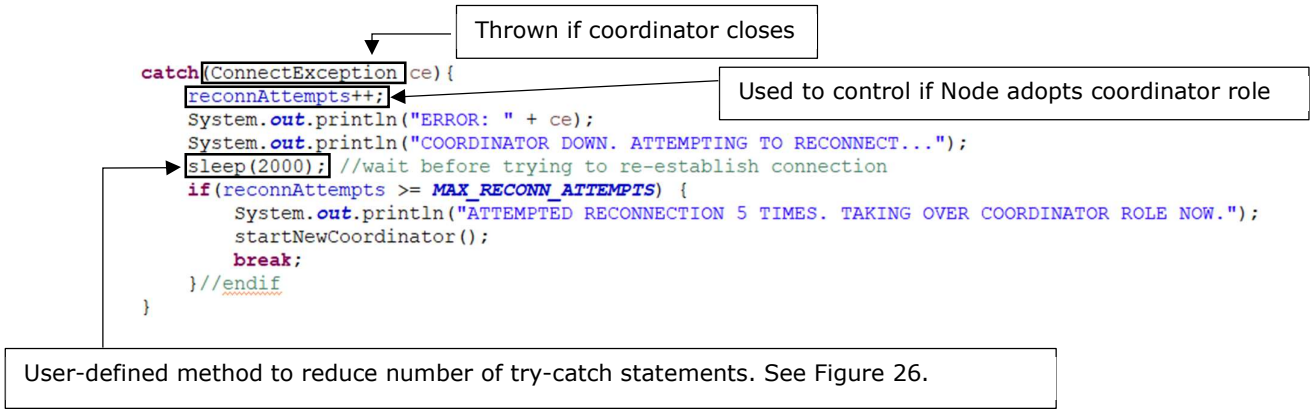

Figure 25 Retry mechanism if coordinator closes

```java
/**
 * Method to make the thread sleep.
 * A useful method to enhance readability.
 * Reduces need for repeating try catch statements.
 * @param duration the duration (integer) of the sleep.
 */
private void sleep(int duration) {
    try {
        Thread.sleep(duration);
    } catch (InterruptedException ie) {
        System.out.println("Thread sleep error: " + n_host_name + ": " + ie);
    }//end try catch
}//end method sleep
```

Figure 26 User-defined method to reduce number of try-catch statements

```java
/**
 * Method to launch a new coordinator.
 * Used in the event of original coordinator failing.
 */
private void startNewCoordinator() {
    Coordinator c = new Coordinator ();
    getAddress();
    clearFile();
    // Create and run a C_receiver and a C_mutex object sharing a C_buffer object
    C_buffer buffer = new C_buffer();// Shared buffer
    C_receiver c_receiver = new C_receiver(buffer, c_request_port); // C_receiver which accesses buffer (the shared resource)
    C_mutex c_mutex = new C_mutex(buffer, c_request_port); // C_mutex which accesses buffer (the shared resource)
    // Run C_receiver and C_mutex
    c_receiver.start();
    c_mutex.start();
}//end method startNewCoordinator
```

Figure 27 Node takes over coordinator role on c_request_port

On testing this implementation, a Node successfully takes over the role as coordinator after five reconnection attempts. However, this implementation is only partially working as it seems from Figure 28 that two Nodes reach the fifth reconnection attempt at the same time, so both Nodes execute the startNewCoordinator() method. After Node on port 9001 takes over the coordinator role, though, the third Node (on port 9002) successfully connects to the new coordinator and resumes activity as normal. This was difficult to debug as system operations execute very quickly, and I am yet to solve the issue of two Nodes taking over the coordinator role at the same time. If time permitted, I would test adding in further checks at different stages, for example checking connection to coordinator before entering the critical section, checking connection on exiting the critical section, checking connection on token return, and perhaps including Boolean flags to indicate if a connection to the coordinator is still detected. This would offer more control over system behaviour if the coordinator fails at any point.

Testing Coordinator Closing

The screenshot below shows the results of closing the Coordinator with three instances of Node running:

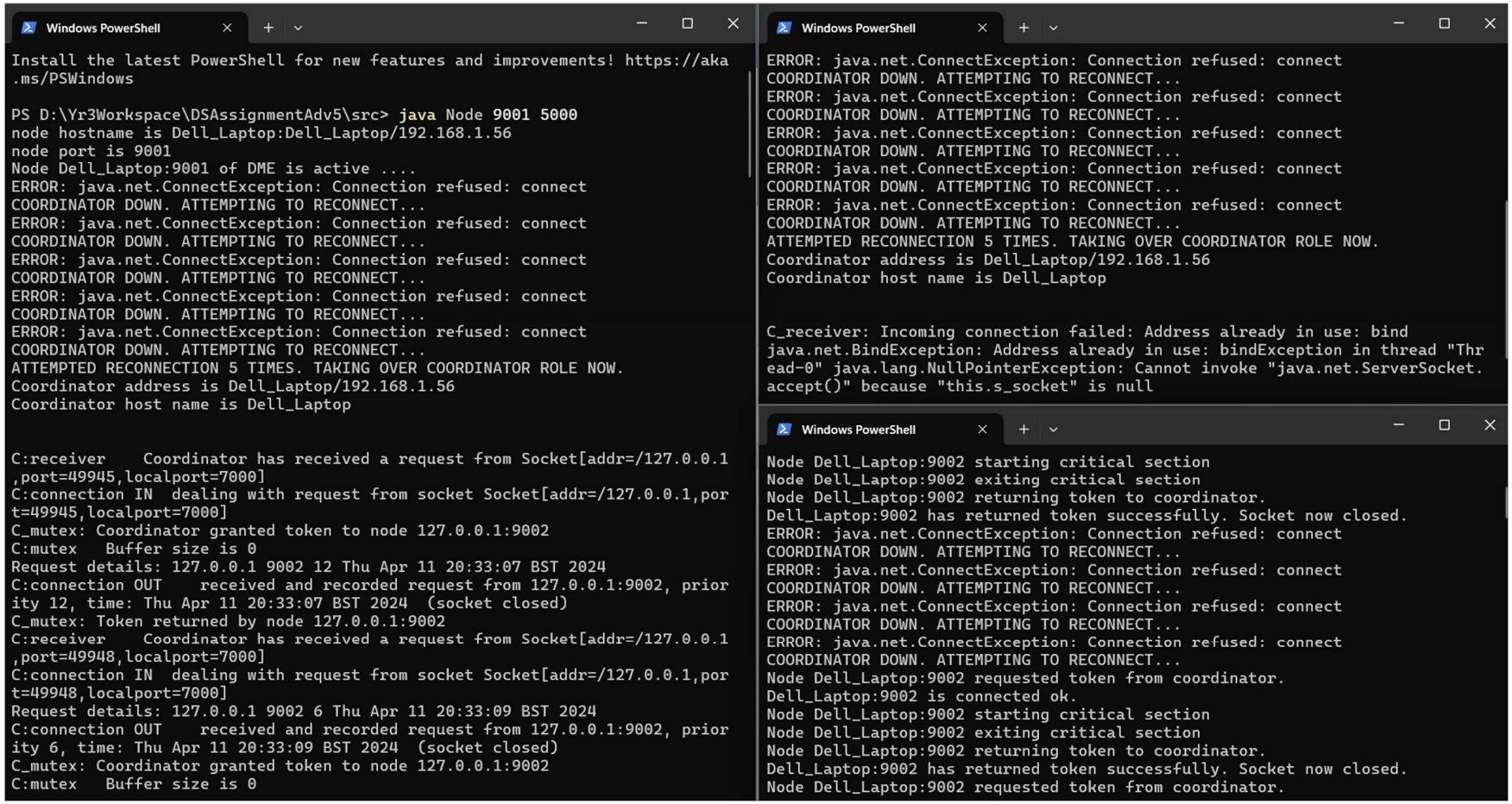

Figure 28 Testing of closing the coordinator

With this feature now working to a fair extent, this is an appropriate point to create an updated sequence diagram.

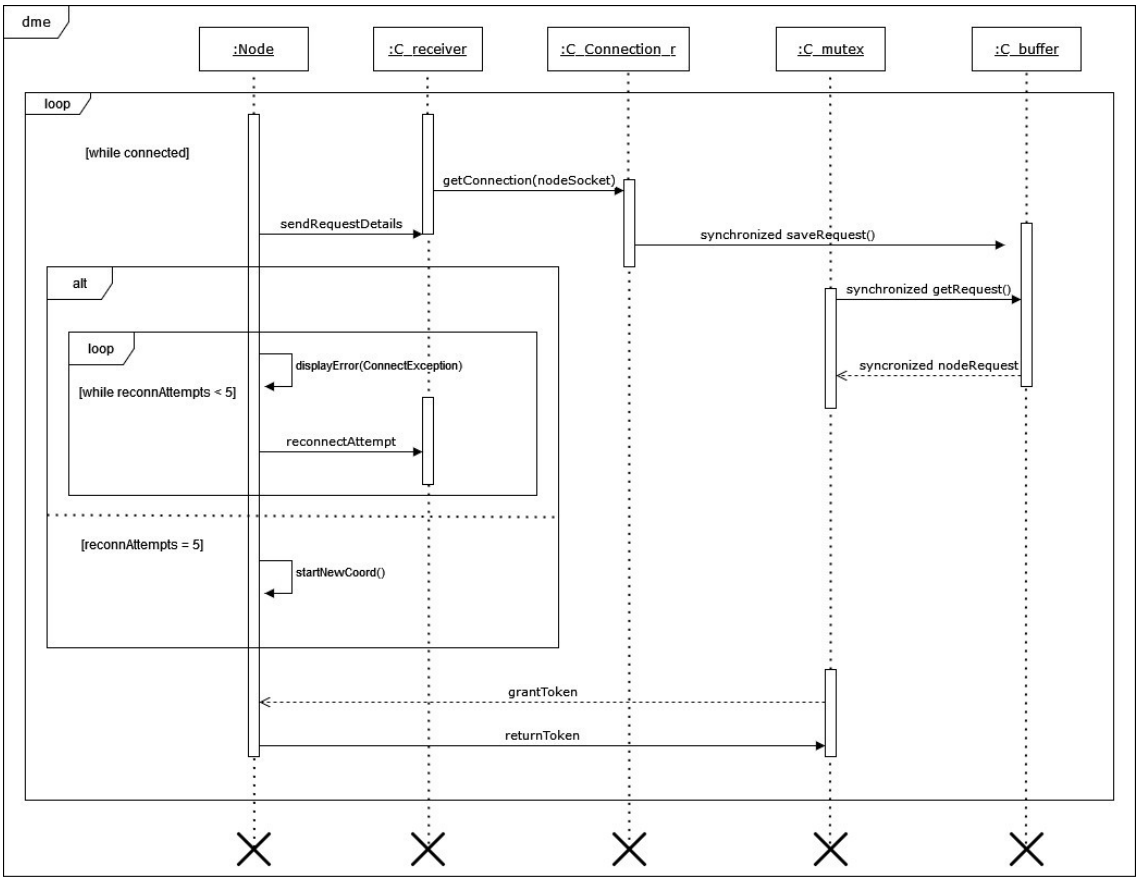

Figure 29 Sequence diagram - coordinator closure

The above diagram shows:

- while a Node is connected:
  - the Node sends a request to coordinator (C_receiver.java);
  - the coordinator establishes a new connection to read the request, and performs a synchronized operation, saving the request in the buffer;
  - now the buffer is not empty, the coordinator (C_mutex.java) can perform a synchronized operation and remove the request;
  - the coordinator can now issue the token to the Node using the request details it has extracted from the buffer;
  - the Node performs its critical section and on completion, returns the token.
- If the cannot connect to the coordinator and reconnAttempts is below 5:
  - it displays an error message and tries to connect again;
  - if reconnAttempts reaches 5, the Node starts a new coordinator.

Token Request Timeout

The solution also works to the extent whereby if the coordinator closes and a Node has just sent a token request, the token request times out after 10 seconds. This is achieved through the ServerSocket method setSoTimeout():

```
/**
 * Method to receive the token from coordinator.
 * Token gives Node permission to perform critical section.
 */
private void getToken() {
    // **** Then Wait for the token
    // Print suitable messages
    try {
        n_ss.setSoTimeout(10000); //wait max 10s for a token
        n_token = n_ss.accept();
    } catch (SocketTimeoutException se) {
        System.out.println("NODE " + n_host_name + ":" + n_port + " REQUEST UNSUCCESSFUL - CONNECTION TIMED OUT." + se);
        sleep(5000); //wait before trying to re-establish connection
    } catch (IOException ioe) {
        System.out.println("ERROR: " + ioe);
        sleep(5000);
    }//end try catch
}//end method getToken
```

Token received through n_ss, so timeout set to 10 seconds.

If no token in 10 seconds, SocketTimeoutException is thrown.

Figure 30 ServerSocket setSoTimeout()

Figure 31 below shows the result of closing the coordinator just as a Node has requested the token.

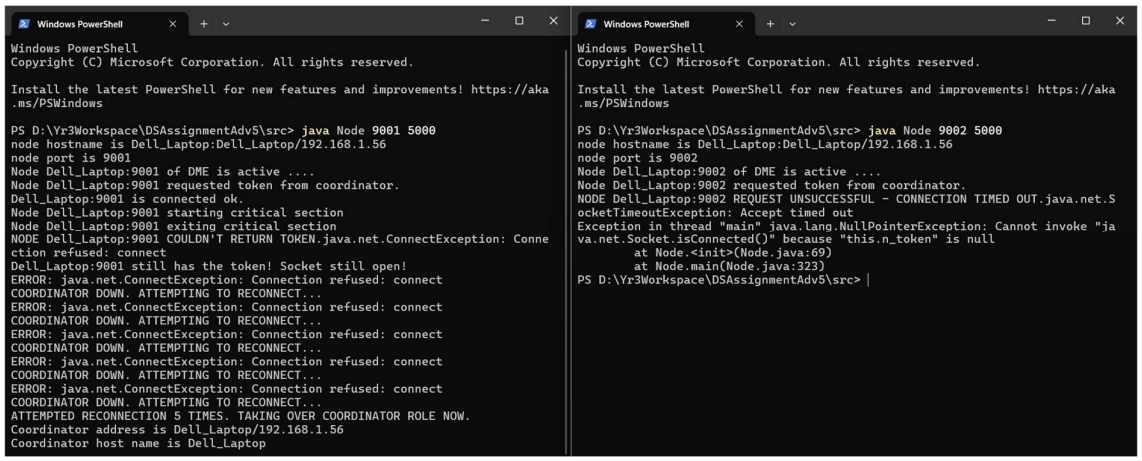

Figure 31 Request time out

What Figure 31 shows is that if the coordinator closes just as a Node has requested a token, the Node's request times out. It also shows that if the coordinator has closed and a Node has not yet returned the token, an exception is thrown and the socket is left open. The Node then iterates over the while(true) loop again and attempts to connect to the coordinator again, and after five attempts, takes over the coordinator role.

When the coordinator is closed and then another coordinator instance is launched, the Nodes successfully re-establish a connection to the coordinator and resume activity:

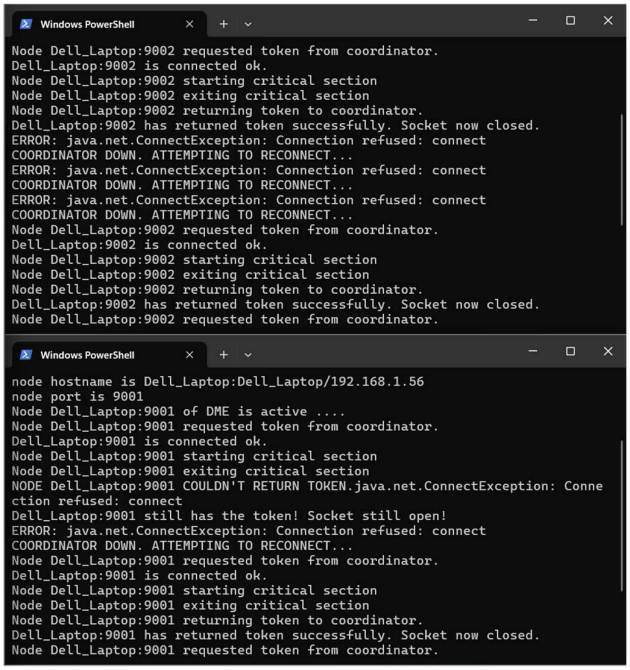

Figure 32 Coordinator closes and another instance launched

When working on this feature, it gave rise to considering what should happen if a non-coordinator Node closes or crashes. Similar to a Node's request having a timeout value, I specify in C_mutex.java that if the coordinator does not receive the token back within 10 seconds, the coordinator can assume the Node has failed or crashed, and can simply move on with the other requests:

```
/**
 * Method to get token back from Node.
 * Timeout value set to 10 seconds.
 * If token not returned in this timeframe, coordinator abandons this Node and proceeds.
 */
private void receiveTokenBack(){
    //  >>>  **** Getting the token back
    try{
        // THIS IS BLOCKING !
        /*
         * ss_back.accept will block until it establishes a connection
         * A connection will only be established when a node has returned a token
         */
        ss_back.setSoTimeout(10000);
        s = ss_back.accept();
        System.out.println("C_mutex: Token returned by node " + n_host + ":" + n_port);
    } catch(SocketTimeoutException ste){
        System.out.println("C_mutex still waiting on: " + n_host + ":" + n_port + " returning token. Moving to next request.")
    } catch (SocketException se) {
        System.out.println("C_mutex lost connection with: " + n_host + ":" + n_port + ". Moving to next request.");
    } catch (java.io.IOException e) {
        System.out.println(e);
        System.out.println("CRASH Mutex waiting for the TOKEN back" + e);
        System.exit(1);
    }//end try catch
}//end method receiveTokenBack
```

Setting timeout value of waiting on token return

If token is not returned in time

Figure 33 Timeout on Node returning token

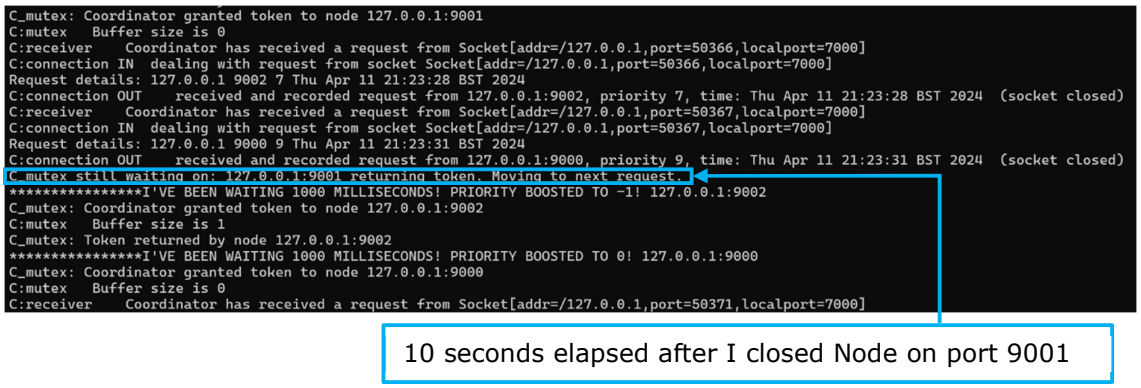

Figure 34 C_mutex.java continues servicing other Nodes if token return times out

Closing Down Requests

Unfortunately I could not get this feature to work successfully. Here I will discuss my approach and the code included in my solution that was intended to satisfy this feature.

Earlier I described the use of one-to-one architecture in a distributed system. In order to implement a full gracefully closing down of the system initiated by a node sending a closing down request, I explored the possibility of implementing an element of the one-to-many architecture. I considered this to be a suitable approach because my objective was to send a signal to all nodes that a closing down request has occurred, and having an additional server that has the single responsibility of continuously listening for closing down requests could then notify all other nodes that are active in the system. I intended to achieve this through creating an additional class, ShutdownServer.java. This is included in my solution submitted alongside this report. The responsibility of this class is to listen for closing down requests, and then broadcast a message to all nodes currently active if a closing down request is received. To do this, I included two class level variables, one to enable listening for connections, and one to store a List of active Nodes:

```
public class ShutdownServer extends Thread {

    private ServerSocket serverSocket; //to listen for incoming connections
    private List<Socket> connectedNodes; //to track active Nodes

    public ShutdownServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
        connectedNodes = new ArrayList<>();
    }//end constructor method
```

Figure 35 ShutdownServer.java

Then, as this is a new thread, I override the run() method and define that while ShutdownServer is running (and it is running continuously), it should accept connections from nodes, add nodes to the List, and check if a closing down request has been received:

```java
/**
 * Overriding default run() method.
 * Continuously listens for connections and checks for closing down requests.
 */
@Override
public void run() {
    try {
        while(true) {
            Socket nodeSocket = serverSocket.accept(); //accept connection
            connectedNodes.add(nodeSocket); //add node to the list
            handleShutdownRequest(nodeSocket);
        }//end while
    } catch (IOException e) {
        e.printStackTrace();
    }//end try catch
}//end method run

/**
 * Method to communicate closing down signal.
 * @param s the socket from the node to connect to.
 * @throws IOException
 */
private void handleShutdownRequest(Socket s) throws IOException {
    InputStream in = s.getInputStream();
    BufferedReader bin = new BufferedReader(new InputStreamReader(in));
    String message = bin.readLine();
    //check if closing down request received
    if("SHUTDOWN".equals(message)) {
        System.out.println("Closing down request initiated by: " + s.getInetAddress());
        s.close(); //close the socket
        broadcastShutdownRequest(); //communicate to all nodes to close down
    }//end if message equals SHUTDOWN
}//end method handleShutdownRequest
```

Enables us to check contents of Node requests.

Figure 37 ShutdownServer.java routine tracks connected Nodes and communicates closing down signal

The method broadcastShutdownRequest(Socket s) is then invoked to take care of communicating the signal. It utilises an enhanced for loop to iterate through each Node that is connected, and closes the Socket that each Node is connected through:

```java
/**
 * Closes all Nodes currently in List of connected Nodes.
 */
private void broadcastShutdownRequest() {
    //for every connected node
    for(Socket node : connectedNodes) {
        //send a signal to close down
        try {
            PrintWriter output = new PrintWriter(node.getOutputStream(), true);
            output.println("SHUTDOWN".getBytes()); //the signal
            node.close(); //now close connection
        } catch (IOException e) {
            e.printStackTrace();
        }//end try catch
    }//end for
}//end method broadcastShutdown
```

Figure 36 ShutdownServer.java communicating closing down signal, then shuts off each Node's socket

Obviously to launch this thread that continuously listens for closing down requests, it must be instantiated. I intended to do this in the Coordinator class, just before the instantiation of the C_buffer, C_receiver, and C_mutex:

```
//ShutdownServer continuously listens for closing down requests
//intended to track List of nodes with pending requests
//closing down signal would be issued to all nodes in the List
/*try {
    ShutdownServer shutdownListener = new ShutdownServer(port);
    shutdownListener.start();
} catch (IOException e) {
    System.out.println("Error in shutdown server: " + e);
}*/
```

Figure 38 Coordinator.java instantiating ShutdownServer

On reflection I feel now that this is not the best approach to listening for closing down requests. As this solution is adopting a centralised approach to ensure mutual exclusion, the coordinator node is always listening anyway. I believe what would have been a more efficient and successful approach is to check what type of request has been received in C_Connection_r.java. A check could have been included, for example, for the first characters in the request; if these characters matched a command such as "shutdown", then this could have triggered notifying all active nodes that they must close immediately. In this manner, an instance of Node with an additional argument – the "shutdown" command – would have to be launched by the user. Other conditions could be used to initiate a closing down request too, such as if X number of reconnection attempts have been made in the event of the coordinator crashing, then a Node should initiate a closing down request. But as I already have something similar in dealing with coordinator closure in the sense that the first Node to reach five reconnection attempts must take over the role of coordinator, my preference was the former approach of launching a "shutdown" Node.

Conclusion

To conclude, the solution submitted alongside this report successfully satisfies the basic solution along with a decent proportion of the advanced features. This has been a challenging assignment; having never studied much networking, it has been fairly difficult to get my head around socket programming. I acknowledge my solution is not perfect and there are issues which need addressing, such as two Nodes taking over the role of coordinator at the same time, checking for coordinator connection at more regular intervals, and full graceful closing down of the system if a closing down request is initiated. I am disappointed to not have fulfilled advanced feature 5, but I would appreciate feedback on the approach I have described and advice on better ways to satisfy this feature would

be appreciated also. Overall, even with these issues, a lot of effort has been applied to this assignment and a great deal has been learned. I have a far deeper understanding and awareness of how important distributed systems are in the world now.

References

[1] A. S. Tanenbaum and M. v. Steen, Distributed systems : principles and paradigms, Upper Saddle River, N.J: Pearson Prentice Hall, 2014.

[2] A. Silberschatz, P. B. Galvin and G. Gagne, Operating system concepts with Java, Hoboken, New Jersey: John Wiley & Sons INC, 2010.

[3] J. Shelton, "Internet at the speed of light," YaleNews, 3 May 2022. [Online]. Available: https://news.yale.edu/2022/05/03/internet-speed-light#:~:text=The%20internet%20is%20such%20a,100%20times%20slower%20than%20that. [Accessed April 2024].

[4] Oracle, "Class BufferedReader," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html. [Accessed April 2024].

[5] baeldung, "Difference Between FileReader and BufferedReader in Java," Baeldung, 7 December 2023. [Online]. Available: https://www.baeldung.com/java-filereader-vs-bufferedreader. [Accessed April 2024].

[6] Oracle, "Package java.util.concurrent," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/package-summary.html. [Accessed April 2024].

Code Listing

Coordinator.java

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

public class Coordinator {

    /**
     * Starting point of the coordinator.
     * @param args command line arguments
     */
    public static void main (String args[]){
        launch(args);
    }//end method main

    /**
     * Method to instantiate coordinator.
     * @param args command line arguments
     */
    private static void launch(String[] args) {
        int port = 7000;
        Coordinator c = new Coordinator ();

        getAddress();
        clearFile();

        // allows defining port at launch time
        if (args.length == 1) port = Integer.parseInt(args[0]);

        //ShutdownServer continuously listens for closing down requests
        //intended to track List of nodes with pending requests
        //closing down signal would be issued to all nodes in the List
        /*try {
            ShutdownServer shutdownListener = new ShutdownServer(port);
            shutdownListener.start();
        } catch (IOException e) {
            System.out.println("Error in shutdown server: " + e);
        }*/

        // Create and run a C_receiver and a C_mutex object sharing a C_buffer object
        // Shared buffer
        C_buffer buffer = new C_buffer();

        // C receiver which accesses buffer (the shared resource)
```

```
44          C_receiver c_receiver = new C_receiver(buffer, port);
45
46          // C_mutex which accesses buffer (the shared resource)
47          C_mutex c_mutex = new C_mutex(buffer, port);
48
49          // Run C_receiver and C_mutex
50          c_receiver.start();
51          c_mutex.start();
52      }//end method launch
53
54      /**
55       * Method to get the InetAddress of coordinator.
56       */
57      private static void getAddress() {
58          try {
59              InetAddress c_addr = InetAddress.getLocalHost();
60              String c_name = c_addr.getHostName();
61              System.out.println ("Coordinator address is "+c_addr);
62              System.out.println ("Coordinator host name is "+c_name+"\n\n");
63          }
64          catch (Exception e) {
65              System.err.println(e);
66              System.err.println("Error in corrdinator");
67          }//end try catch
68      }//end method getAddress
69
70      /**
71       * Method to clear contents of text file if not already clear.
72       */
73      private static void clearFile() {
74          //Clear the text file
75          try {
76              // create fileWriter - false = new file so clear contents
77              FileWriter file_writer_id = new FileWriter("2118616_log.txt", false);
78              file_writer_id.close();
79          } catch (IOException e) {
80              System.err.println("Exception in clearing file: main: " + e);
81          }// end try-catch
82      }//end method clearFile
83
84  }//end class Coordinator
85
```

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;
    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;
    private Date timestmp;
    private String timestamp;
    private int reconnAttempts = 0;
    private static final int MAX_RECONN_ATTEMPTS = 5;
    private FileWriter file_writer;
    private PrintWriter print_writer;
    //private String shutdown_host = "127.0.0.1";
    //private int shutdown_port = 7002;
    //private Socket closingDownRequestSocket; //closing down request would be communicated via this Socket
    //private ServerSocket closeDownSignal_ss;

    /**
     * Constructor method.
     * @param nam the Node's host name.
     * @param por the port number a Node is running on.
     * @param sec upper bound (exclusive) of wait time prior to sending request.
     * @throws InterruptedException
     */
    public Node(String nam, int por, int sec) throws InterruptedException {
        ra = new Random();
        n_host_name = nam;
        n_port = por;
        System.out.println("Node " +n_host_name+ ":" +n_port+ " of DME is active ....");

        //>>> NODE sends n_host and n_port  through a socket s to the coordinator
        //>>> c_host:c_req_port and immediately opens a server socket through which will receive
        //>>> a TOKEN (actually just a synchronization).

      try {
          n_ss = new ServerSocket(n_port);//token received through this ServerSocket
      } catch (IOException e) {
          System.out.println("Node "+n_host_name+ ":" +n_port+" couldn't open socket: " + e);
      }//end try catch

      /*
       * The ServerSocket through which a Node would receive a signal to close down.
       */
      /*try {
          closeDownSignal_ss = new ServerSocket(n_port);
      } catch (IOException e) {
          System.out.println("Node "+n_host_name+ ":" +n_port+" couldn't open socket: " + e);
      }  */

      while(true){

          // >>>  sleep a random number of seconds linked to the initialisation sec value
          sleep(ra.nextInt(sec));

          try {
              openFile();

              s = new Socket(c_host, c_request_port); //send request through socket s
              sendRequestDetails();
              s.close();
              reconnAttempts = 0;//connection established now, so reset reconnection attempts

              getToken();

              if(n_token.isConnected()) {
                  System.out.println(n_host_name + ":" + n_port + " is connected ok.");
                  executeCriticalSection(sec);
              } //endif

              returnToken();

              if(n_token.isClosed()) System.out.println(n_host_name + ":" + n_port + " has returned token successfully. Socket now closed.")
              else System.out.println(n_host_name + ":" + n_port + " still has the token! Socket still open!");

          }
          catch(ConnectException ce){
              reconnAttempts++;
              System.out.println("ERROR: " + ce);
              System.out.println("COORDINATOR DOWN. ATTEMPTING TO RECONNECT...");
              sleep(2000); //wait before trying to re-establish connection
              if(reconnAttempts >= MAX_RECONN_ATTEMPTS) {
```

```
                System.out.println("ATTEMPTED RECONNECTION 5 TIMES. TAKING OVER COORDINATOR ROLE NOW.");
                startNewCoordinator();
                break;
            }//endif
        }
        catch (java.io.IOException e) {
            System.out.println("ERROR: " + e);
            sleep(5000); //wait before trying to re-establish connection
            //System.exit(1);
        }//end try catch
    }//end while
}//end constructor method

/**
 * Method to send token back to coordinator.
 * on completion of critical section.
 */
private void returnToken() {
    // **** Return the token
    // Print suitable messages - also considering communication failures
    try {
        timestmp = new Date();
        timestamp = timestmp.toString();
        writeToFile("**NODE " + n_host_name + ":" + n_port + " RETURNED TOKEN " + timestamp + "**");
        closeFile();
        n_token = new Socket(c_host, c_return_port);
        System.out.println("Node " + n_host_name + ":" + n_port + " returning token to coordinator.");
        n_token.close();
    } catch (IOException e) {
        System.out.println("NODE " + n_host_name + ":" + n_port + " COULDN'T RETURN TOKEN." + e);
        sleep(5000);
    }//end try catch
}//end method returnToken

/**
 * Method to perform critical section code.
 * This is simulated by a sleep here.
 * @param sec used to generate a random sleep (integer) value.
 */
private void executeCriticalSection(int sec) {
    // **** Sleep for a while
    // This is the critical session
    int sleep = ra.nextInt(sec);
    timestmp = new Date();
    timestamp = timestmp.toString();
    writeToFile("**NODE " + n_host_name + ":" + n_port + " STARTING CRITICAL SECTION " + timestamp + " for " + sleep + " milliseconds**");
    System.out.println("Node " + n_host_name + ":" + n_port + " starting critical section");
    sleep(sleep);
    System.out.println("Node " + n_host_name + ":" + n_port + " exiting critical section");
}//end method executeCriticalSection

/**
 * Method to receive the token from coordinator.
 * Token gives Node permission to perform critical section.
 */
private void getToken() {
    // **** Then Wait for the token
    // Print suitable messages
    try {
        n_ss.setSoTimeout(10000); //wait max 10s for a token
        n_token = n_ss.accept();
    } catch (SocketTimeoutException se) {
        System.out.println("NODE " + n_host_name + ":" + n_port + " REQUEST UNSUCCESSFUL - CONNECTION TIMED OUT." + se);
        sleep(5000); //wait before trying to re-establish connection
    } catch (IOException ioe) {
        System.out.println("ERROR: " + ioe);
        sleep(5000);
    }//end try catch
}//end method getToken

/**
 * Method to make the thread sleep.
 * A useful method to enhance readability.
 * Reduces need for repeating try catch statements.
 * @param duration the duration (integer) of the sleep.
 */
private void sleep(int duration) {
    try {
        Thread.sleep(duration);
    } catch (InterruptedException ie) {
        System.out.println("Thread sleep error: " + n_host_name + ": " + ie);
    }//end try catch
}//end method sleep

/**
 * Method to send request details.
 */
private void sendRequestDetails() {
    // **** Send to the coordinator a token request.
    // send your ip address and port number
    try {
        pout = new PrintWriter(s.getOutputStream(), true);
        pout.println(n_host); //ip
        pout.println(n_port); //port
```

```
187             pout.println(getPriority());
188             pout.println(System.currentTimeMillis());//the timestamp of request
189             System.out.println("Node " + n_host_name + ":" + n_port + " requested token from coordinator.");
190             pout.close();
191         } catch (IOException e) {
192             System.out.println("Error sending request: " + e);
193             sleep(5000);
194         }//end try catch
195     }//end method sendRequestDetails
196
197     /**
198      * Method to generate a priority value for a Node.
199      * @return a random integer between 1 and 15 (inclusive).
200      */
201     private int getPriority() {
202         return ra.nextInt(15)+1;
203     }//end method getPriority
204
205     /**
206      * Method to launch a new coordinator.
207      * Used in the event of original coordinator failing.
208      */
209     private void startNewCoordinator() {
210         Coordinator c = new Coordinator ();
211         getAddress();
212         clearFile();
213         // Create and run a C_receiver and a C_mutex object sharing a C_buffer object
214         C_buffer buffer = new C_buffer();// Shared buffer
215         C_receiver c_receiver = new C_receiver(buffer, c_request_port); // C_receiver which accesses buffer (the shared resource)
216         C_mutex c_mutex = new C_mutex(buffer, c_request_port); // C_mutex which accesses buffer (the shared resource)
217         // Run C_receiver and C_mutex
218         c_receiver.start();
219         c_mutex.start();
220     }//end method startNewCoordinator
221
222     /**
223      * Method to get this Node's address.
224      * Used if this Node takes over coordinator role.
225      */
226     private void getAddress() {
227         try {
228             InetAddress c_addr = InetAddress.getLocalHost();
229             String c_name = c_addr.getHostName();
230             System.out.println ("Coordinator address is "+c_addr);
231             System.out.println ("Coordinator host name is "+c_name+"\n\n");
232         }
233         catch (Exception e) {
234             System.err.println(e);
235             System.err.println("Error in corrdinator");
236         }//end try catch
237     }//end method getAddress
238
239     /*
240     private void sendClosingDownRequest() {
241         try {
242             closingDownRequestSocket = new Socket(shutdown_host, shutdown_port);
243             PrintWriter out = new PrintWriter(closingDownRequestSocket.getOutputStream(), true);
244             out.println("SHUTDOWN");
245         } catch (IOException e) {
246             System.out.println("Node " + n_host_name + ":" + n_port + " failed to send closing down request: " + e);
247         }//end try catch
248     }//end method sendClosingDownRequest
249     */
250
251     /*================================ FILE HANDLING METHODS ================================*/
252
253     /**
254      * Method to clear contents of text file if not already clear.
255      */
256     private void clearFile() {
257         //Clear the text file
258         try {
259             // create fileWriter - false = new file so clear contents
260             FileWriter file_writer_id = new FileWriter("2118616_log.txt", false);
261             file_writer_id.close();
262         } catch (IOException e) {
263             System.err.println("Exception in clearing file: main: " + e);
264         }// end try-catch
265     }//end method clearFile
266
267     /**
268      * Method to open text file for file logging if it does not already exist.
269      */
270     private void openFile() {
271         try {
272             file_writer = new FileWriter("2118616_log.txt", true);
273             print_writer = new PrintWriter(file_writer, true);
274         } catch(IOException e) {
275             System.out.println("Error creating file for node " + n_host_name + ":" + n_port + " " + e);
276         }//end try catch
277     }//end method createFile
278
279     /**
280      * Method to log a message to file.
```

```
281      * @param entry the message to log to file.
282      */
283     private void writeToFile(String entry) {
284         print_writer.println(entry);
285     }//end method writeToFile
286 
287     /**
288      * Method to close the file.
289      */
290     private void closeFile() {
291         try {
292             print_writer.close();
293             file_writer.close();
294         } catch (IOException e) {
295             System.out.println("Error closing file for node " + n_host_name + ":" + n_port + " " + e);
296         }//end try catch
297     }//end method closeFile
298 
299     /*============================== END FILE HANDLING METHODS ==============================*/
300 
301     /**
302      * Main method that constructs a Node.
303      * @param args command line argument
304      * @throws InterruptedException
305      */
306     public static void main (String args[]) throws InterruptedException{
307         String n_host_name = "";
308         int n_port;
309 
310         // port and millisec (average waiting time) are specific of a node
311         if ((args.length < 1) || (args.length > 2)){
312             System.out.print("Usage: Node [port number] [millisecs]");
313             System.exit(1);
314         }//endif
315 
316         // get the IP address and the port number of the node
317         try{
318             InetAddress n_inet_address =  InetAddress.getLocalHost() ;
319             n_host_name = n_inet_address.getHostName();
320             System.out.println ("node hostname is " +n_host_name+":"+n_inet_address);
321         }
322         catch (java.net.UnknownHostException e){
323             System.out.println(e);
324             System.exit(1);
325         }//end try catch
326 
327         n_port = Integer.parseInt(args[0]);
328         System.out.println ("node port is "+n_port);
329         Node n = new Node(n_host_name, n_port, Integer.parseInt(args[1]));
330 
331     }//end method main
332 
333 }//end class Node
334 
```

C_receiver.java

```
1  import java.io.IOException;
2  import java.net.*;
3
4  public class C_receiver extends Thread {
5
6      private C_buffer buffer;
7      private int port;
8      private ServerSocket s_socket;
9      private Socket socketFromNode;
10     private C_Connection_r connect;
11
12     /**
13      * Constructor method.
14      * @param b the buffer (the shared resource).
15      * @param p the port the port a request is made on.
16      */
17     public C_receiver (C_buffer b, int p){
18         buffer = b;
19         port = p;
20     }//end constructor method
21
22     /**
23      * Method to override default run() method.
24      * While this thread is running, it listens for incoming connections.
25      * On receiving a connection, it starts a new thread that will save the request.
26      */
27     @Override
28     public void run () {
29         // >>> create the socket the server will listen to
30         createServerSocket();
31         while (true) {
32             // >>> get a new connection
33             getNewConnection();
34             // >>> create a separate thread to service the request, a C_Connection_r thread.
35             serviceRequest();
36         }//end while
37     }//end run
38
39     /**
40      * Method to create the ServerSocket that will listen for incoming connections.
41      */
42     private void createServerSocket() {
43         try {
44             s_socket = new ServerSocket(port);
45         } catch (IOException e) {
46             System.out.println("C_receiver: Incoming connection failed: " + e.getMessage());
47         }//end try catch
48     }//end method createServerSocket
49
50     /**
51      * Method to establish a connection from a Node.
52      * Connection is established through socket that Node sends the request on.
53      */
54     private void getNewConnection() {
55         try {
56             socketFromNode = s_socket.accept();
57             System.out.println ("C:receiver    Coordinator has received a request from " + socketFromNode);
58         } catch(IOException ioe) {
59             System.out.println("C_receiver couldn't connect to " + socketFromNode + ioe);
60             System.exit(1);
61         }//end try catch
62     }//end method getNewConnection
63
64     /**
65      * Method to create a separate thread that will service the request.
66      * Request will be saved to buffer through this connect thread.
67      */
68     private void serviceRequest() {
69         connect = new C_Connection_r(socketFromNode, buffer);
70         connect.start();
71     }//end method serviceRequest
72
73 }//end class C_receiver
74
```

C_Connection_r.java

```
import java.net.*;
import java.text.SimpleDateFormat;
import java.util.Date;
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
    private C_buffer buffer;
    private Socket s;
    private InputStream in;
    private BufferedReader bin;
    private FileWriter file_writer;
    private PrintWriter print_writer;
    private String[] request;
    final int NODE = 0; //node ip at position 0 in request String array
    final int PORT = 1; //port number at position 1 in request String array
    final int PRIORITY = 2; //priority at position 2 in request String array
    final int TIME = 3; //timestamp at position 3 in request String array

    /**
     * Constructor method
     * @param s the socket to access input stream from Node
     * @param b the buffer to add requests to
     */
    public C_Connection_r(Socket s, C_buffer b){
        this.s = s;
        this.buffer = b;
    }//end constructor method

    /**
     * Method to override default run() method.
     * While this thread is running it will read in a Node's request.
     * Then, it will add the request to the buffer and log relevant events to file.
     */
    @Override
    public void run() {

        request = new String[4]; //modified to size 4 to include priority and time

        System.out.println("C:connection IN  dealing with request from socket "+ s);
        try {
            openFile();

            // >>> 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).
          createInputStream();

          readRequest();

          // Creating date format - improves readability of output statements
          SimpleDateFormat simple = new SimpleDateFormat("E MMM dd HH:mm:ss zzz yyyy");
          Date result = new Date(Long.parseLong(request[TIME]));

          recordRequest(); //Save the request object in buffer

          writeToFile("REQUEST from node on port " + request[PORT] + ", PRIORITY " + request[PRIORITY] + ", TIME: " + simple.format(result) +
                  "\nC_Connection_r: Buffer size: " + buffer.size());

          s.close();
          System.out.println("Request details: " + request[NODE] + " " + request[PORT] + " " + request[PRIORITY] + " " + simple.format(result));
          System.out.println("C:connection OUT    received and recorded request from " + request[NODE]+":"+request[PORT]+
                  ", priority " + request[PRIORITY]+ ", time: " + simple.format(result) + "  (socket closed)");

          closeFile();
       } catch (java.io.IOException e){
              System.out.println(e);
              System.exit(1);
       }//end try catch
    }//end method run

    /**
     * Method to add request into the buffer.
     */
    private void recordRequest() {
        buffer.saveRequest(request, Integer.parseInt(request[PRIORITY]), Long.parseLong(request[TIME]));
    }//end method recordRequest

    /**
     * Method to read from input stream from node.
     */
    private void readRequest() {
        try {
            //Read the node IP
            request[NODE] = bin.readLine();

            //Read the node port number
            request[PORT] = bin.readLine();

            //Read the node priority
            request[PRIORITY] = bin.readLine();

            //Read the node timestamp
```

```
96              request[TIME] = bin.readLine();
97
98              bin.close();
99          } catch (IOException ioe) {
100             System.out.println("C_Connection_r couldn't read request details: " + ioe);
101             System.exit(1);
102         }//end try catch
103     }//end method readRequest
104
105     /**
106      * Method to create input stream from node accessed through socket s.
107      * InputStreamReader(in) will convert stream of bytes from socket s to characters
108      * BufferedReader more efficient and provides methods to read text from stream
109      */
110     private void createInputStream() {
111         try {
112             in = s.getInputStream(); //input stream from node accessed through socket s
113             bin = new BufferedReader(new InputStreamReader(in));
114         } catch(IOException e) {
115             System.out.println("C_Connection_r: Error accessing request: " + e);
116         }//end try catch
117     }//end method createInputStream
118
119     /*================================ FILE HANDLING METHODS ================================*/
120
121     /**
122      * Method to open text file if it does not already exist.
123      */
124     private void openFile() {
125         try {
126             file_writer = new FileWriter("2118616_log.txt", true);
127             print_writer = new PrintWriter(file_writer, true);
128         } catch(IOException e) {
129             System.out.println("C_Connection_r couldn't open file: " + e);
130         }//end try catch
131     }//end method createFile
132
133     /**
134      * Method to log a message to file.
135      * @param entry the message to log to file.
136      */
137     private void writeToFile(String entry) {
138         print_writer.println(entry);
139     }//end method writeToFile
140
141     /**
142      * Method to close the file.
143      */
144     private void closeFile() {
145         try {
146             print_writer.close();
147             file_writer.close();
148         } catch (IOException e) {
149             System.out.println("C_Connection_r couldn't close file: " + e);
150         }//end try catch
151     }//end method closeFile
152
153 }//end class C_Connection_r
154
```

C_buffer.java

```java
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.*;
import java.util.concurrent.TimeUnit;

public class C_buffer {

    private Vector<Object> data;
    private FileWriter file_writer;
    private PrintWriter print_writer;
    private long aging = TimeUnit.SECONDS.toMillis(1); //used to avoid starvation; after aging value elapsed, low priority processes have priority increased

    /**
     * Constructor method.
     * Initialises data structure (Vector) that stores Node requests.
     * Opens text file if not already open.
     */
    public C_buffer (){
        data = new Vector<Object>();
        try {
            file_writer = new FileWriter("2118616_log.txt", true);
            print_writer = new PrintWriter(file_writer, true);
        } catch (IOException e) {
            System.out.println("C_buffer: Error opening file: " + e);
        }//end try catch
    }//end constructor method

    /**
     * Method to get number of elements in the buffer.
     * @return the size of the Vector data.
     */
    public int size() {
        return data.size();
    }//end method size

    /**
     * Method to add a request to the buffer.
     * Synchronized so only one thread at a time can manipulate buffer.
     * @param r the Node's request to be added.
     */
    public synchronized void saveRequest (String[] r){
        data.add(r[0]);
        data.add(r[1]);
    }//end method saveRequest

    /**
     * Method to display contents of buffer.
     */
    public void show(){
        for (int i=0; i<data.size();i++)
            System.out.print(" "+data.get(i)+" ");
        System.out.println(" ");
    }//end method show

    /**
     * Method to add an Object to the buffer.
     * Synchronized as it manipulates buffer contents.
     * @param o the Object to be added.
     */
    public synchronized void add(Object o){
        data.add(o);
    }//end method add

    /**
     * Method to retrieve and remove an item from the buffer.
     * @return the item at index 0 in the buffer.
     */
    public synchronized Object get(){
        Object o = null;
        if (data.size() > 0){
            o = data.get(0);
            data.remove(0);
        }
        return o;
    }//end method get

    /**
     * Overloaded method to save a request with a priority and timestamp.
     * Synchronized as only one thread at a time should manipulate buffer contents.
     * @param r the request to save.
     * @param priority the priority of the request.
     * @param time the time the request was made.
     */
    public synchronized void saveRequest(String[] r, int priority, long time) {
        r[2] = Integer.toString(priority); //insert priority to String array
        r[3] = Long.toString(time); //save time with the request
        data.add(r); //add request r to buffer
    }//end method saveRequest

    /**
     * Retrieves and removes a request from the queue.
     * Synchronized as only one thread at a time should manipulate buffer contents.
     * @return the request at index 0 after sorting performed; otherwise null.
     */
```

```
    public synchronized Object getRequest() {
        if(!data.isEmpty()) {
            //first sort the data according to priority (ascending because smaller integers = higher priority)
            sortByPriority();
            String[] request = (String[]) data.remove(0); //remove the first request
            if(request != null) {
                long currentTime = System.currentTimeMillis(); //get the current time
                long requestTime = Long.parseLong(request[3]); //get the request timestamp
                long waitingTime = currentTime - requestTime; //calculate how long request has been waiting
                if(waitingTime > aging) { //is waiting time more than default time (aging)?
                    int priority = Integer.parseInt(request[2]); //then extract priority of the request
                    int updatedPriority = priority - (int) (waitingTime / aging); //boost its priority.
                    request[2] = Integer.toString(updatedPriority); //now insert updated priority into String request array
                    print_writer.println("***************I'VE BEEN WAITING " + aging + " MILLISECONDS! PRIORITY BOOSTED TO "
                    + request[2]+ "!" + " " + request[0]+":"+request[1]);
                    System.out.println("***************I'VE BEEN WAITING " + aging + " MILLISECONDS! PRIORITY BOOSTED TO "
                    + request[2]+ "!" + " " + request[0]+":"+request[1]);
                }//end if waitingTime > aging
            }//end if request not null
            return request;
        }//end if data not empty
        return null;
    }//end method getRequest

    /**
     * Sorts data into ascending order.
     */
    private void sortByPriority() {
        for(int i = 0; i < data.size()-1; i++) {
            for(int j = i + 1; j < data.size(); j++) {
                String[] r1 = (String[]) data.get(i);
                String[] r2 = (String[]) data.get(j);
                int p1 = Integer.parseInt(r1[2]);
                int p2 = Integer.parseInt(r2[2]);
                //if priority of process 2 is smaller than priority of process 1
                if(p2 < p1) Collections.swap(data, i, j); //make the swap
            }//end inner for
        }//end outer for
    }//end method sortByPriority

}//end class C_buffer

```

C_mutex.java

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

public class C_mutex extends Thread {

    private C_buffer buffer;
    private Socket s;
    private int port;

    // ip address and port number of the node requesting the token.
    // They will be fetched from the buffer
    private String n_host;
    private int n_port;
    private int priority;
    private long time;

    private FileWriter file_writer;
    private PrintWriter print_writer;
    private ServerSocket ss_back;

    /**
     * Constructor method.
     * @param b the buffer from where a request is removed.
     * @param p the port the port a request is received on.
     */
    public C_mutex (C_buffer b, int p){
        buffer = b;
        port = p;
    }//end constructor method

    /**
     * Method to override default run() method.
     * While this thread is running it will check buffer size.
     * If buffer is not empty (>0) it will read the request details,
     * then issue the token, and wait for the Node to return the token.
     */
    @Override
    public void run(){
        try{
            //  >>>  Listening from the server socket on port 7001
            // from where the TOKEN will be returned later.
            ss_back = new ServerSocket(7001);

            while (true){
                // if the buffer is not empty
                if (buffer.size() > 0) {
                    openFile(); //for access to shared resource
                    getNode(); //read request details
                    grantToken(); //fulfil request - issue token to node
                    closeFile();
                    receiveTokenBack(); //block until coordinator receives token
                }//endif
            }//endwhile
        }catch (Exception e) {
            System.out.print(e);
        }//end try catch
    }//end method run

    /*================================ SERVICE NODE REQUEST METHODS ================================*/

    /**
     * Method to get token back from Node.
     * Timeout value set to 10 seconds.
     * If token not returned in this timeframe, coordinator abandons this Node and proceeds.
     */
    private void receiveTokenBack(){
        //  >>>  **** Getting the token back
        try{
            // THIS IS BLOCKING !
            /*
             * ss_back.accept will block until it establishes a connection
             * A connection will only be established when a node has returned a token
             */
            ss_back.setSoTimeout(10000);
            s = ss_back.accept();
            System.out.println("C_mutex: Token returned by node " + n_host + ":" + n_port);
        } catch(SocketTimeoutException ste) {
            System.out.println("C_mutex still waiting on: " + n_host + ":" + n_port + " returning token. Moving to next request.");
        } catch (SocketException se) {
            System.out.println("C_mutex lost connection with: " + n_host + ":" + n_port + ". Moving to next request.");
        } catch (java.io.IOException e) {
            System.out.println(e);
            System.out.println("CRASH Mutex waiting for the TOKEN back" + e);
            System.exit(1);
        }//end try catch
    }//end method receiveTokenBack

    /**
     * Method to issue token to Node.
     * Successful connection through Socket s using Node
     * host name and port number indicates token has been granted.
     */
    private void grantToken() {
```

```
96          // >>>  **** Granting the token
97          try{
98              s = new Socket(n_host, n_port);
99              writeToFile("TOKEN ISSUED to " + n_host + ":" + n_port + ", priority: " + priority +
100                     "\nC:mutex   Buffer size is " + buffer.size());
101             System.out.println("C_mutex: Coordinator granted token to node " + n_host + ":" + n_port);
102             System.out.println("C:mutex   Buffer size is "+ buffer.size());
103             s.close();
104         } catch (java.io.IOException e) {
105             System.out.println(e);
106             System.out.println("CRASH Mutex connecting to the node for granting the TOKEN " + e);
107             sleep(2500);
108         }//end try catch
109     }//end method grantToken
110
111     /**
112      * Method to make the thread sleep.
113      * A useful method to enhance readability.
114      * Reduces need for repeating try catch statements.
115      * @param duration the duration (integer) of the sleep.
116      */
117     private void sleep(int duration) {
118         try {
119             Thread.sleep(duration);
120         } catch (InterruptedException ie) {
121             System.out.println("C_mutex sleep failed: " + ie);
122         }//end try catch
123     }//end method sleep
124
125     /**
126      * Method to get the highest priority node waiting for a token.
127      */
128     private void getNode() {
129         // >>>   Getting the first (highest priority) node that is waiting for a TOKEN form the buffer
130         //       Type conversions may be needed.
131         try{
132                 String[] request = (String[])buffer.getRequest();
133                 n_host = request[0];
134                 n_port = Integer.parseInt(request[1]);
135                 priority = Integer.parseInt(request[2]);
136                 time = Long.parseLong(request[3]);
137         } catch (Exception e){
138             System.err.println("C_mutex couldn't convert request: " + e);
139         }//end try catch
140     }//end method getNode
141
142     /*================================ FILE HANDLING METHODS ================================*/
143
144     /**
145      * Method to open text file for file logging if it does not already exist.
146      */
147     private void openFile() {
148         try {
149             file_writer = new FileWriter("2118616_log.txt", true);
150             print_writer = new PrintWriter(file_writer, true);
151         } catch(IOException e) {
152             System.out.println("C_mutex couldn't create file: " + e);
153         }//end try catch
154     }//end method createFile
155
156     /**
157      * Method to log a message to file.
158      * @param entry the message to log to file.
159      */
160     private void writeToFile(String entry) {
161         print_writer.println(entry);
162     }//end method writeToFile
163
164     /**
165      * Method to close the file.
166      */
167     private void closeFile() {
168         try {
169             print_writer.close();
170             file_writer.close();
171         } catch (IOException e) {
172             System.out.println("C_mutex couldn't close file: " + e);
173         }//end try catch
174     }//end method closeFile
175
176 }//end class C_mutex
177
```

ShutdownServer.java (not in use but included in project)

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;

public class ShutdownServer extends Thread {

    private ServerSocket serverSocket; //to listen for incoming connections
    private List<Socket> connectedNodes; //to track active Nodes

    /**
     * Constructor method
     * @param port the port a closing down request is received on
     * @throws IOException
     */
    public ShutdownServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
        connectedNodes = new ArrayList<>();
    }//end constructor method

    /**
     * Overriding default run() method.
     * Continuously listens for connections and checks for closing down requests.
     */
    @Override
    public void run() {
        try {
            while(true) {
                Socket nodeSocket = serverSocket.accept(); //accept connection
                connectedNodes.add(nodeSocket); //add node to the list
                handleShutdownRequest(nodeSocket);
            }//end while
        } catch (IOException e) {
            e.printStackTrace();
        }//end try catch
    }//end method run

    /**
     * Method to communicate closing down signal.
     * @param s the socket from the node to connect to.
     * @throws IOException
     */
    private void handleShutdownRequest(Socket s) throws IOException {
        InputStream in = s.getInputStream();
        BufferedReader bin = new BufferedReader(new InputStreamReader(in));
        String message = bin.readLine();
        //check if closing down request received
        if("SHUTDOWN".equals(message)) {
            System.out.println("Closing down request initiated by: " + s.getInetAddress());
            s.close(); //close the socket
            broadcastShutdownRequest(); //communicate to all nodes to close down
        }//end if message equals SHUTDOWN
    }//end method handleShutdownRequest

    /**
     * Closes all Nodes currently in List of connected Nodes.
     */
    private void broadcastShutdownRequest() {
        //for every connected node
        for(Socket node : connectedNodes) {
            //send a signal to close down
            try {
                PrintWriter output = new PrintWriter(node.getOutputStream(), true);
                output.println("SHUTDOWN".getBytes()); //the signal
                node.close(); //now close connection
            } catch (IOException e) {
                e.printStackTrace();
            }//end try catch
        }//end for
    }//end method broadcastShutdown

    /**
     * Method would be called after all Nodes receive closing down signal.
     * @throws IOException
     */
    public void closeShutdownServer() throws IOException {
        serverSocket.close();
    }//end method closeShutdownServer

}//end class ShutdownServer

```