**Operating Systems Concurrency and Distribution**

**CSCU9V5**

**Problem statement:**

Developing a socket based centralised Distributed Mutual Exclusion system based on socket-based token ring. The system should avoid dead locks and address the issues around running the critical section in a multithreaded environment. The critical section should only be executed exclusively by only one Node at a time in this distributed system. After the execution of critical section each node should return the token so that other Nodes can execute their critical sections.

The given coordinator program runs on a JVM on one of the computers in the system and it passes a unique token to each single node that requests it, sequentially. When granted the unique token, a node will be allowed to execute the critical section and then return the token to the coordinator.

**Solution:**

From the given skeleton solution, I have developed a Socket based centralised DME. I have made the following assumptions:

* Multiple Nodes running in the system in different processes.
* Each Node has a critical section that requires mutual exclusion.
* Node should request the Coordinator with its IP address, port number and waits for token.
* Once token is issued, Node should execute its critical section and return the token and close connection.

Node takes the port number, wait time and priority as program arguments, if these arguments are not presented program will exit with a usage message on console. Once the user executes the program with correct arguments the constructor sends the request the coordinator a token, once token is received the critical section of the program (in this case it is just wait) and Node thus returns the token on the returning port.

The coordinator has a class level C\_buffer that is passed to C\_recevier and C\_mutex.

Coordinator also creates C\_receiver and C\_mutex and start them. C\_recevier listens to the node requests and through C\_Connection\_r stores the Node details i.e. hostname, port number and priority in the buffer. C\_­mutex gets the gets the request from the buffer in a FIFO order. For each request the mutex grants the token to the requesting node by a simple synchronisation to the node’s port. Then, it waits for the token to be returned.

As per the given skeleton my solution consists of the following classes:

**Node:**

In the Node class Port, milliseconds and priority arguments must be passed to run the program. In case of invalid input user will be notified of the usage and program will terminate. With a valid input a constructor is invoked with host name, port number, time and priority. Priority is passed to C\_receiver as data not as meta data. Node will send a token request to Coordinator class by sending it’s IP address and Port number also Priority is passed via output stream. Node waits for the token, once token is received from C\_mutex class node starts the execution of its critical section. Once critical section execution is completed token is returned to C\_Mutex.

Note: To run the Node in infinite loop, I created a local variable “cycle” and call the constructor in an infinite while loop, incrementing the port number with cycle.

**Coordinator:**

The Coordinator is the main component creates buffer and starts C-receiver and C-mutex threads. Though these 2 classes coordinator serves all the nodes. Coordinator is listening for Node requests on port number is 7000.

**C\_receiver:**

The C\_receiver receives the request from Node and it will create a new connection i.e. Connection and passes the Node details host, port number

**C\_Connection\_r:**

The connection deals with request from the socket at port 7000. It will read the request, i.e. node IP and port are from socket and save it in a request object, it also reads data from socket i.e. the priority of port and save the request object in the buffer. It uses C\_buffer add method to store the request. After the recorded the request than the socket will be closed.

**C\_mutex:**

In C\_mutex there is an IP address and port number of the node requesting the token and they will be fetched from the buffer. For initial version these requests are fetched sequentially. An integer token will be generated using the random next integer and pass to Node. C\_mutex will be listening on port 7001 from where the TOKEN will be returned later from Node. C\_mutex verifies if the same token is been returned from Node and prints the information to the user. The thread continues to serve next Node returned from C\_buffer. To get the next request C\_mutex uses C\_buffer get method.

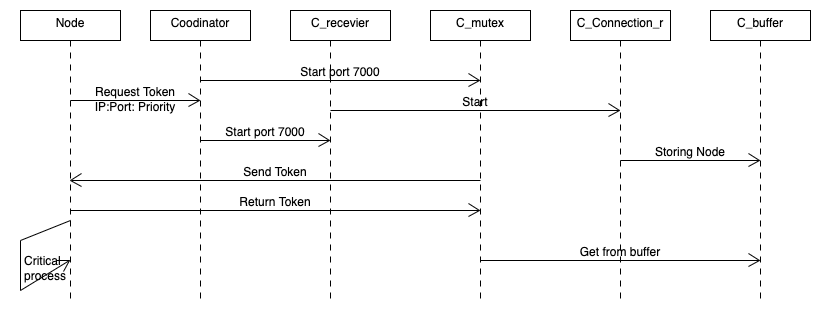
**C\_buffer:**

C\_buffer class is used to save the Node requests, the add method of this class is used by C\_connection\_r to save the Node requests, the requests are stored in a vector. Each request object consists of IP address, port number and priority. C\_buffer get method determines the highest priority Node and sends that node to C\_mutex to send the token for processing the critical section. C\_buffer class has got some utility methods show and size, there print data on the console and returns the size of the buffer respectively.

**Class Diagram:** Below is the class diagram of the DME system which shows the class level variables and methods.



**Sequence Diagram:** Below is the sequence diagram of my DME system. All major interactions are shown below.



**Code :**

**Node.java :**

import java.io.IOException;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.net.InetAddress;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.net.UnknownHostException;  
import java.util.Date;  
  
public class Node {  
  
  
 private ServerSocket server;  
 private Socket client;  
  
  
 String c\_host = "127.0.0.1";  
 final int c\_request\_port = 7000;  
 final int c\_return\_port = 7001;  
  
 String n\_host = "127.0.0.1";  
 String n\_host\_name;  
 int n\_port;  
  
 ObjectOutputStream outputStream;  
 ObjectInputStream inputStream;  
 int token = 0;  
  
 public Node(String name, int port, int sec, int priority) throws InterruptedException, UnknownHostException, IOException, ClassNotFoundException {  
  
 n\_host\_name = name;  
 n\_port = port;  
 System.out.println("Node: Node " + n\_host\_name + ":" + n\_port + " of DME is active ....");  
  
 while (true) {  
  
 //Sleep a random number of seconds linked to the initialisation sec value  
 Thread.sleep(sec \* 1000);  
  
 try {  
 //Send to the coordinator a token request.  
 //sending your ip address and port number  
 client = new Socket(n\_host\_name, c\_request\_port, InetAddress.getLocalHost(), port);  
 System.out.println("Node: Opened socket to requesting Coordinator for token on port 7000");  
  
 outputStream = new ObjectOutputStream(client.getOutputStream());  
 //Priority of the node is passed via outputStream  
 outputStream.writeObject(priority);  
 outputStream.close();  
  
 //Wait for the token  
 //this is just a synchronization  
 // Print suitable messages  
 synchronized (this) {  
 System.out.println(new Date() + ":Node: Listening for token from C\_Mutex on port: " + port);  
 server = new ServerSocket(port);  
  
 client = server.accept();  
 System.out.println("Node: Client accepted: " + client);  
  
 inputStream = new ObjectInputStream(client.getInputStream());  
 token = (Integer) inputStream.readObject();  
 System.out.println("Node: Token received: " + token);  
 client.close();  
 }  
  
  
 // Sleep half a second, say  
 // This is the critical session  
 System.out.println("Node: Started executing critical section");  
 Thread.sleep(3 \* 1000);//this is where we do critical processing  
 System.out.println("Node: Finished critical section");  
  
  
 //Return the token  
 // this is just establishing a sync connection to the coordinator's ip and return port.  
 // Print suitable messages - also considering communication failures  
 client = new Socket(n\_host\_name, c\_return\_port); //"127.0.0.1:7001"  
 System.out.println("Node: opened socket on port: " + c\_return\_port + " to return token");  
  
 outputStream = new ObjectOutputStream(client.getOutputStream());  
 outputStream.writeObject(token);  
 System.out.println("Node: token returned");  
  
 inputStream.close();  
 outputStream.close();  
 server.close();  
 client.close();  
  
 break;  
 } catch (java.io.IOException e) {  
 System.out.println(e);  
 System.exit(1);  
 }  
 }  
 }  
  
 public static void main(String args[]) throws NumberFormatException, InterruptedException, UnknownHostException, IOException, ClassNotFoundException {  
  
 String n\_host\_name = "";  
 int n\_port;  
 int cycle = 0;  
  
 //Port and milli seconds program arguments must be passed to run the Node class  
 //Priority of the Node is also passed as argument here  
 if ((args.length < 2) || (args.length > 3)) {  
 System.out.print("Node: Usage: Node [port number], [millisecs], [priority]");  
 System.exit(1);  
 }  
  
 //Get the IP address and the port number of the node  
 try {  
 InetAddress n\_inet\_address = InetAddress.getLocalHost();  
 n\_host\_name = n\_inet\_address.getHostName();  
 System.out.println("Node: node hostname is " + n\_host\_name + ":" + n\_inet\_address);  
 } catch (java.net.UnknownHostException e) {  
 System.out.println(e);  
 System.exit(1);  
 }  
  
 n\_port = Integer.parseInt(args[0]);  
 System.out.println("Node: node port is " + n\_port);  
 // Creating the node using the constructor  
 //priority also passed to the constructor  
 while (true) {  
 cycle++;  
 Node n = new Node(n\_host\_name, n\_port+cycle, Integer.parseInt(args[1]), Integer.parseInt(args[2]));  
 }  
 }  
}

**Coodinator.java :**

import java.net.\*;  
  
public class Coordinator {  
  
 public static void main(String args[]) {  
  
 int port = 7000;  
 //Created a class level C\_buffer object  
 C\_buffer buffer = new C\_buffer();  
  
 Coordinator c = new Coordinator();  
  
 try {  
 InetAddress c\_addr = InetAddress.getLocalHost();  
 String c\_name = c\_addr.getHostName();  
 System.out.println("Coordinator: Coordinator address is " + c\_addr);  
 System.out.println("Coordinator: Coordinator host name is " + c\_name + "\n\n");  
 } catch (Exception e) {  
 System.err.println(e);  
 System.err.println("Coordinator: Error in coordinator");  
 }  
  
 // override default port with user input  
 if (args.length == 1) {  
 port = Integer.parseInt(args[0]);  
 }  
  
 // Create and run a C\_receiver and a C\_mutex object sharing a C\_buffer object  
 Thread receiver = new C\_receiver(buffer, port);  
 //Starting the receiver thread  
 receiver.start();  
 System.out.println("Coordinator: receiver has started");  
  
 Thread mutex = new C\_mutex(buffer, port);  
 //Starting the mutex thread  
 mutex.start();  
 System.out.println("Coordinator: mutex has started");  
 }  
}

**C\_recevier.java :**

import java.io.IOException;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
  
public class C\_receiver extends Thread {  
  
 private C\_buffer buffer;  
 private int port;  
  
 private ServerSocket server;  
 private Socket client;  
  
 private C\_Connection\_r connection;  
  
  
 public C\_receiver(C\_buffer buffer, int port) {  
 this.buffer = buffer;  
 this.port = port;  
 System.out.println("C\_receiver: started with port: " + port);  
 }  
  
  
 public void run() {  
  
 //Create the socket the server will listen to  
 try {  
 server = new ServerSocket(7000);  
 } catch (IOException e1) {  
 System.err.println("C\_receiver: Error in listening on port 7000");  
 e1.printStackTrace();  
 System.exit(1);  
 }  
  
 while (true) {  
 try {  
 //Get a new connection  
 client = server.accept();  
 System.out.println("C\_receiver: Coordinator has received a request from node: " + client);  
  
 // Create a separate thread to service the request, a C\_Connection\_r thread.  
 connection = new C\_Connection\_r(client, buffer);  
 connection.start();  
 } catch (java.io.IOException e) {  
 System.out.println("C\_receiver: Exception when creating a connection " + e);  
 }  
 }  
 }//end run  
}

**C\_mutex.java :**

import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.net.ServerSocket;  
import java.net.Socket;  
import java.util.Random;  
  
/\*\*  
 \* Class that fetches request from buffer in FIFO.  
 \* This sends the token to client and wait for it to return.  
 \*/  
public class C\_mutex extends Thread {  
  
  
 private Random ra = new Random();  
  
 C\_buffer buffer;  
 Socket client;  
 int port;  
  
 //ip address and port number of the node requesting the token.  
 //They will be fetched from the buffer  
 String n\_host;  
 int n\_port;  
  
 ObjectOutputStream outputStream;  
 ObjectInputStream inputStream;  
  
 //Integer token that is passed to Node  
 int token;  
  
  
 public C\_mutex(C\_buffer buffer, int port) {  
 this.buffer = buffer;  
 this.port = port;  
 }  
  
 public void go() {  
  
 try {  
 // Listening from the server socket on port 7001  
 // from where the TOKEN will be later on returned.  
 // This place the server creation outside the while loop.  
 ServerSocket server = new ServerSocket(7001);  
  
 while (true) {  
  
 // if the buffer is not empty  
 if (buffer.size() > 0) {  
  
 String[] request;  
 // Getting the first (FIFO) node that is waiting for a TOKEN form the buffer  
 // Type conversions may be needed.  
 Thread.sleep(3 \* 1000);  
 request = (String[]) buffer.get();  
  
 n\_host = request[0];  
 n\_port = Integer.parseInt(request[1]);  
 System.out.println("C\_mutex: Fetched details from buffer " + n\_host + ":" + n\_port);  
  
 //Granting the token  
 try {  
 System.out.println("C\_mutex: opening socket on port: " + n\_port + " to send token");  
 client = new Socket(n\_host, n\_port);  
  
 outputStream = new ObjectOutputStream(client.getOutputStream());  
 //Using random number we are generating the integer token  
 token = ra.nextInt();  
 outputStream.writeObject(token);  
 System.out.println("C\_mutex: token sent: " + token);  
 } catch (java.io.IOException e) {  
 System.err.println(e);  
 System.err.println("C\_mutex: CRASH Mutex connecting to the node for granting the TOKEN" + e);  
 }  
  
 //Getting the token back  
 try {  
 client = server.accept();  
 System.out.println("C\_mutex: Client accepted: " + client);  
  
 inputStream = new ObjectInputStream(client.getInputStream());  
 int tokenReceived = (Integer) inputStream.readObject();  
 System.out.println("C\_mutex: Have you received same token: " + (tokenReceived == token));  
 System.out.println("C\_mutex: Token sent is returned: " + tokenReceived);  
 } catch (java.io.IOException e) {  
 System.err.println(e);  
 System.err.println("C\_mutex: CRASH Mutex waiting for the TOKEN back" + e);  
 }  
  
 }// endif  
 }// endwhile  
 } catch (Exception e) {  
 System.err.print(e);  
 }  
 }  
  
 public void run() {  
 go();  
 }  
}

**C\_Connection\_r.java :**

import java.io.\*;  
import java.net.Socket;  
  
  
/\*\*  
 \* Class that acts on node request.  
 \* It receives and store the node request in the buffer  
 \*/  
public class C\_Connection\_r extends Thread {  
  
 // class variables  
 C\_buffer buffer;  
  
 Socket client;  
  
 public C\_Connection\_r(Socket client, C\_buffer buffer) {  
 this.client = client;  
 this.buffer = buffer;  
 }  
  
 @Override  
 public void run() {  
  
 final int NODE = 0;  
 final int PORT = 1;  
 final int PRIORITY = 2;  
  
 String[] request = new String[3];  
  
 System.out.println("C\_connection: IN connection in dealing with request from socket " + client);  
  
 try {  
  
 // read the request, i.e. node ip and port from the socket s  
 // save it in a request object and save the object in the buffer (see C\_buffer's methods)  
 // assign node ip address and port details to request string array  
 request[NODE] = client.getInetAddress().getHostAddress();  
 request[PORT] = Integer.toString(client.getPort());  
  
 System.out.println("C\_connection: storing request: " + request[NODE] + ":" + request[PORT] + ":" + request[PRIORITY]);  
 ObjectInputStream inputStream = new ObjectInputStream(client.getInputStream());  
 //reading the priority of the Node  
 request[PRIORITY] = String.valueOf(inputStream.readObject());  
 // store request in the buffer  
 buffer.add(request);  
  
 client.close();  
 System.out.println("C\_connection: OUT received and recorded request from " + request[NODE] + ":" + request[PORT] + " (socket closed)");  
 } catch (IOException | ClassNotFoundException e) {  
 System.out.println(e);  
 System.exit(1);  
 }  
  
 buffer.show();  
 } // end of run() method  
} // end of class

**C\_buffer.java :**

import java.util.\*;  
  
public class C\_buffer {  
  
 private Vector<Object> data;  
  
  
 public C\_buffer() {  
 data = new Vector<Object>();  
 }  
  
  
 /\*\*  
 \* Thread safe method to store the request  
 \*  
 \* @param request  
 \*/  
 public synchronized void saveRequest(String[] request) {  
 data.add(request[0]); //ip  
 data.add(request[1]); //port  
 data.add(request[2]); //priority  
 }  
  
  
 /\*\*  
 \* Thread safe method to return first object in data structure  
 \*  
 \* @return  
 \*/  
 synchronized public Object get() {  
  
 Object obj = null;  
  
 int key = 0;  
 int index = 0;  
 String[] str = new String[3];  
 //Looping though the buffer data and getting the the high priority Node index  
 for (int i = 0; i < data.size(); i++) {  
 str = (String[]) data.get(0);  
 int curPriority = Integer.parseInt(str[2]);  
 if (curPriority > key) {  
 key = curPriority;  
 index = i;  
 System.out.println(String.format("C\_buffer: Highest priority of the Node %s, index of Node: %s", key, index));  
 }  
 }  
  
 if (data.size() > 0) {  
 obj = data.get(index);  
 data.remove(index);  
 }  
  
 System.out.println("C\_buffer: Object fetched from buffer: " + str[0] + ":" + str[1] + ":" + str[2]);  
 return obj;  
 }  
  
 /\*\*  
 \* To store obj in buffer  
 \*  
 \* @param obj  
 \*/  
 public void add(Object obj) {  
 data.add(obj);  
 }  
  
  
 /\*\*  
 \* To show objects in buffer  
 \*/  
 public void show() {  
  
 if (data == null  
 || data.size() == 0) {  
 System.out.println("C\_buffer: There are no object in buffer");  
 return;  
 }  
  
 for (int i = 0; i < data.size(); i++) {  
 String[] str = (String[]) data.get(i);  
 System.out.print("C\_buffer: " + i + "# " + str[0] + ":" + str[1]);  
 }  
  
 System.out.println(" ");  
 }  
  
 /\*\*  
 \* Method to return the buffer size  
 \*  
 \* @return  
 \*/  
 public int size() {  
 return data.size();  
 }  
}