# CSCU9V5 REPORT

By 2721301

# Code solution

## Assumptions

- The initial assumption is that this code comprises of a server and two nodes, which are interacting over a network on separate machines.
- At this stage the system is assumed to be designed to handle two sets of traffic lights and no more. Although the final solution may be able to handle more than two lights, it is not a feature that will be purposefully implemented.
- The coordinator is assumed to continuously receive requests from both the traffic light nodes to enter their critical section
- The system is assumed to fail if the coordinator crashes, meaning the nodes can no longer enter the critical section
- Only one node can be in its critical section at one time, and must exit this before the other one may enter

## Description of the solution

With my solution the system is working as intended from the brief. I first modified the Coordinator class by calling the start() method on both C_receiver and C_mutex making them run concurrently. Then in the C_receiver class I created a ServerSocket that will listen to any connections from nodes on a selected port "port". Then I assigned a socket to an accepted connection from this ServerSocket, passing this socket and the buffer to the C_Connection class and calling Start()

In the C_Connection class I initialized the BufferedReader called "in" which gets the InputStream from the socket, passing information from the nodes OutputStream on that socket. I saved the data from the BufferedReader into a string array and then passed this array to be saved in the buffer.

In the C_Mutex class I added some print statements to show that the buffer starts empty, and then an if statement for if it contains items in the vector. Inside the statement I assigned "n_host" and "n_port" to the values that the node previously stored in the buffer, after some type casting. I granted the token to the node by creating a socket to listen to the node, allowing it into its critical section. I then added ss_back.accept() which is a server socket listening for the node to return the token, the details of this is in the node class.

In the Node class I first added some code to delay the thread for a random amount of time to allow some randomness to the system. I then created a socket using the coordinators host IP and the request port for it. I then wrote to the OutputStream the details of the node to be stored in the buffer. To wait for the token, I initialised a new serversocked that will accept requests on the nodes port. Once accepted the node will print out the green light is on, followed by a random delay, then the light off, displaying time in milliseconds to see duration of critical section. To return the token I created a socket using the Coordinators host IP and the return port and printing a suitable message. To finish this, I then closed the sockets.

## Classes & relationships

**Coordinator**
The coordinator starts by creating a buffer object and an integer called port, which is the port that will be receiving connections from the node. This class then instantiates two classes. Receiver to receive requests from the node and mutex which is used let the node enter the critical section. It passes the buffer object, the receiving port to receiver and the token port (port+1) to mutex and runs these classes concurrently.

**C_buffer**
This is a vector classes used to store details on the nodes that are trying to enter the critical section. This class receives data from the connection class and contains methods to manipulate and return the contents within the vector. It then allows the mutex class to create connections with node after it has requested the node information stored in the buffer

**C_receiver**
This class continuingly listens for requests on a socket from node which consist of the node IP and its port to send the token to. On connection to the node it will spawn a thread of the C_Connection_r class passing the socket and buffer.

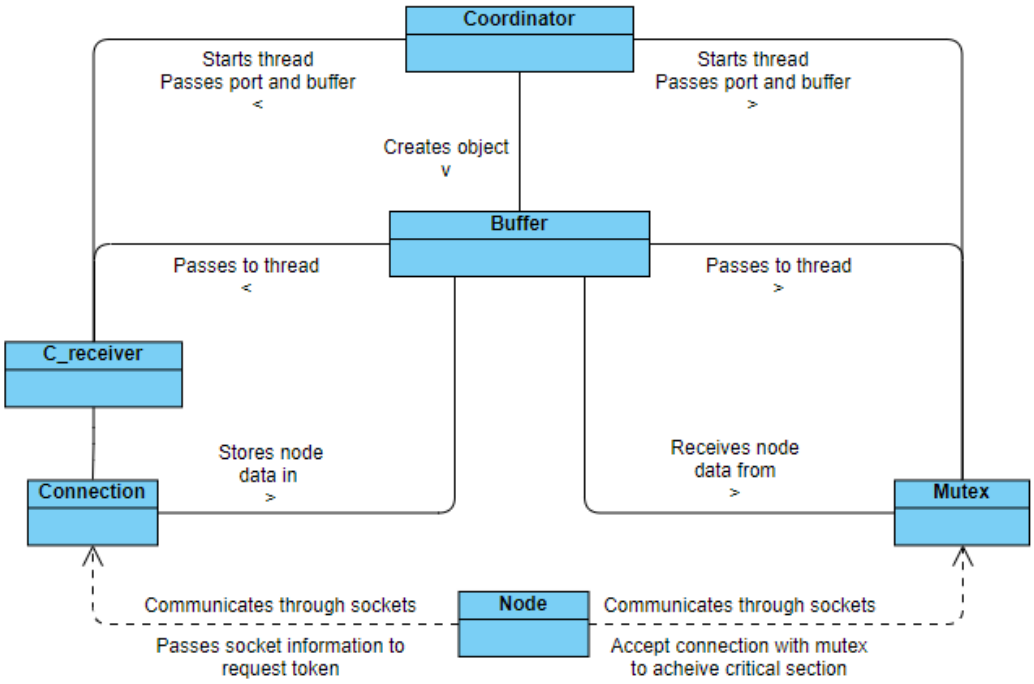
**C_Connection_r**
This class receives the data, the node IP and port, from the InputStream over the socket and inputs it into an array, which is then sent to the buffer to be saved. It displays messages that the connection is successful

**C_mutex**
This class will continuously fetch the node information from the buffer in a first in first out order. After receiving the details it grants a token to the node, allowing it to enter its critical section. It then waits for the token to be returned from the node on the “return_port”.

**Node**
There are two separate instances of this class with two different ports. This class performs a loop of requesting a token by sending its details to the coordinator, waits until token is granted then executes the critical section by turning the light green, only one may turn green at a time. After a delay the light then turns off and the node returns the token back to the coordinator, ending its critical section.

**Class relationships**

The following diagram is how I visualised the relationships of the classes within this system.




As the diagram shows, one location contains the coordinator, buffer, C_receiver, Connection and Mutex class. This location will be classed as the server. At another separate location contains the node class and represents the client. The coordinator class is used to run the C_receiver and mutex class concurrently, passing the buffer to both. The C_receiver class spawns an instance of the Connection class on connection with a node, then receives information from the node class which is then stored in the buffer class. The Mutex class then receives this node information from the buffer and uses it to set a connection with the Node. The relationship between Node and Connection, and Node and mutex is created using a socket on two different ports, one being a request port and one being a return port. The classes pass information using input and output streams over the socket.

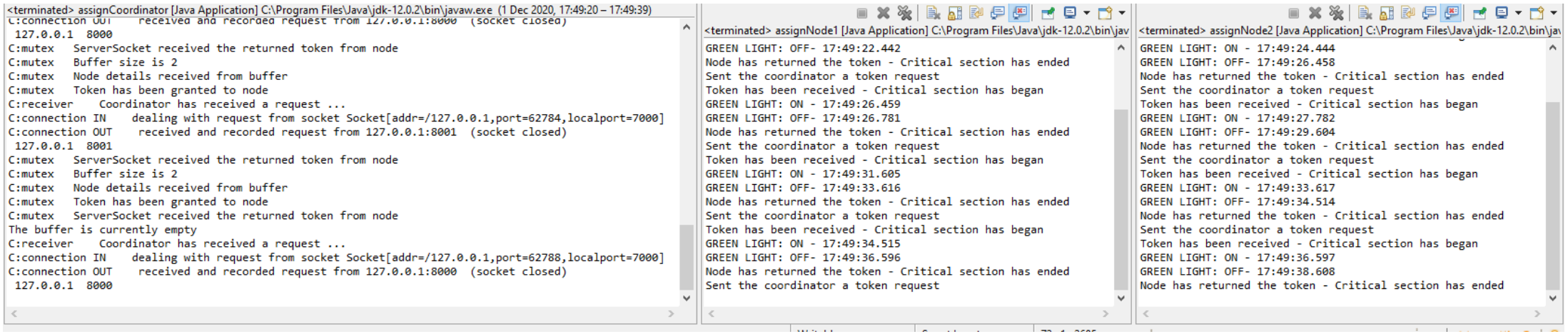
## Important methods

In this system the buffer class provided some important methods to run the system correctly. It contained:

- size() – To return the size of the buffer, which is checked at the start of the C_Mutex class
- saveRequest() – Used to save an array to the buffer containing the node IP address and port, this is a synchronised method meaning both of the nodes can be saving information to this buffer at the same time
- show() – Used to print the contents stored in the buffer, useful for displaying information during run time and for debugging purposes
- get() – used to remove from the buffer in FIFO order to be returned to the mutex class so it can grant the node a token. Calling get() will remove the nodes data from the buffer as it is not required after its critical section, and allows the next node to enter its critical section after.

## Completeness

When comparing my solution to the design brief given almost everything works correctly and the program is running as expected. All changes in the program are clearly printed to help follow the execution and make sure everything is correct as shown in the image of the eclipse consoles below.



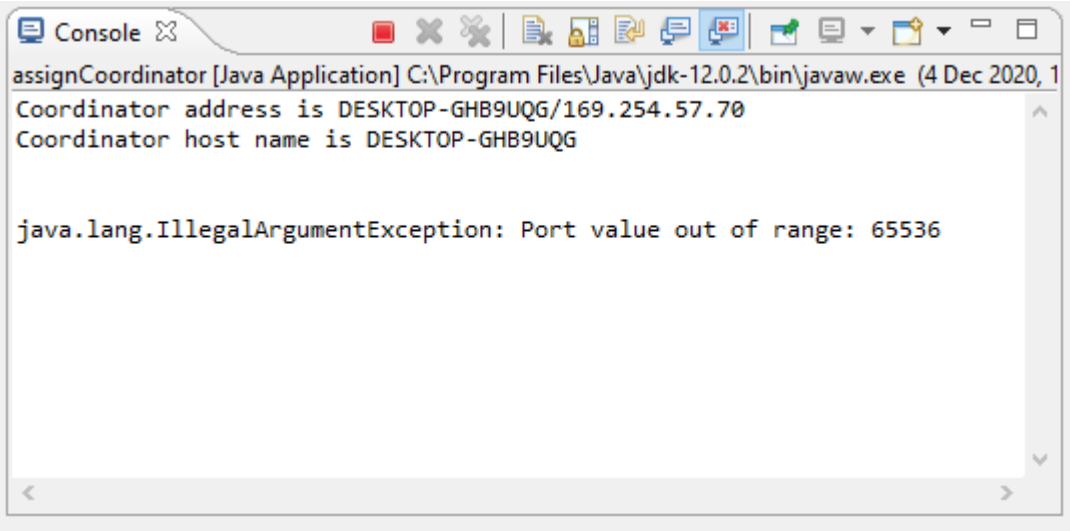

The only part of the design brief that I have not implemented is closing all the sockets and ServerSockets. The brief states that all of these sockets must be closed, however after closing some my program would run into errors and crash. Being unsure why this is I have just closed all the sockets I can without the system crashing and letting the rest of them stay open. I tried troubleshooting this but without any success decided to just leave it and add comments as to why they are not closed.

# Advanced feature 2

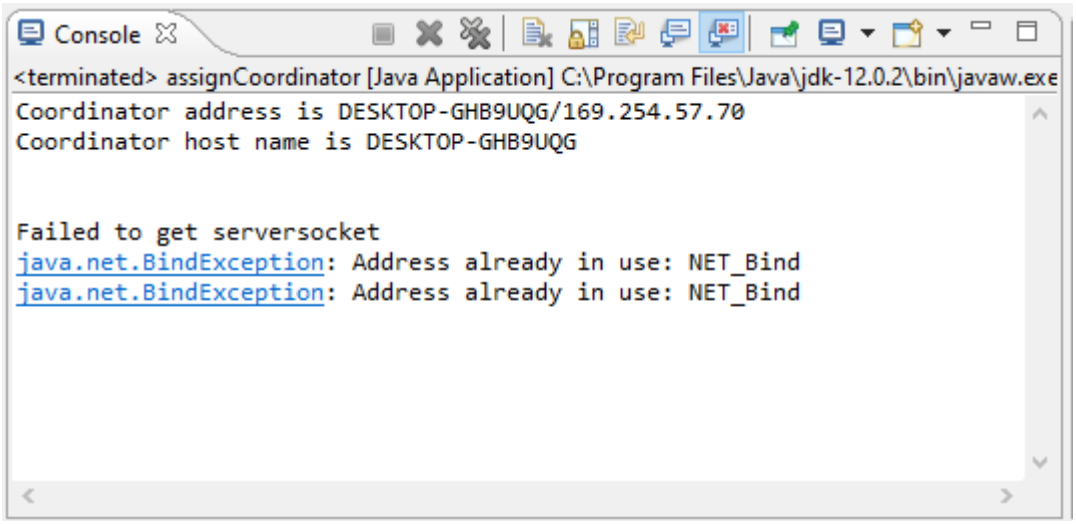
Describe in the report, in an extra and properly marked section titled Advanced feature 2, what would happen if the coordinator crashes. Be as precise as possible and consider all the possible relevant cases, depending on when the coordinator crashes. You may want to report snapshots of the execution of your system to illustrate your findings (you can add extra pages for these snapshots).

Possible scenarios where the coordinator may crash:

- User inputs a port not available on the machine
  - In this scenario the user has entered the arguments a port value that is out of range on a normal system, which has a maximum amount of 65,536 ports. When testing the program with this port value I ran the catch and printed the error “port value out of range”.
  - This error occurs in the C_receiver class and the C_mutex class as they have been passed the port to create the socket, these classes will print out a suitable error message.
  - Due to no sockets being created, the Node will have nothing to talk to and cannot request any tokens, meaning they will never enter their critical section and turn the green light on.
  - A possible solution to this would be to add a condition that if the port requested is outside a certain range, it will assign a different port and prints out that it has done so. It will then pass this port to the running threads as normal.



- The port requested is already being used by another process
  - This time the user has entered a port that is currently already being used by another process on the machine
  - This is similar to the previous error where the socket receives a port that it cannot use, so like previous the nodes are not able to enter their critical section and the system cant function
  - A solution for this may be adding some code to check if the socket is already in use with a try catch statement within a while loop. The try will try to create the socket and the catch will change the port number if unsuccessful, then tries again within the while loop until successful
  - The program may then continue to function as normal




- The coordinator has crashed, and a deadlock has occurred
  - The coordinator has crashed due to an unexpected problem during run time, and the node is currently in its critical section while another is trying to request the token. The first node is unable to return the token to the coordinator to be given to node two meaning the system is currently in deadlock and cannot proceed
  - A solution for this problem may be incorporating an election algorithm called the bully algorithm

  - Using an election algorithm, a unique priority number will be assigned to all active processes within the system, with the coordinator having the highest. The goal of this type of algorithm is to determine where a new version of the coordinator can be started. The algorithm selects this active coordinator as a new coordinator due to its high priority number.
  - Specifically, the Bully algorithm is useful for this system. Can be implemented so that if the coordinator does not answer the node after a certain time interval then it can be assumed to have failed. This node will then try to elect itself as a coordinator and sends a message to every other process with a higher priority number, including the restarted coordinator. If no response then these processes have also failed, however if there are responses then the process with the highest priority number will be elected as the new coordinator. However, this system cannot resume without a coordinator.