**Round Robin Code Report (N0927786)**

Table of Contents

[Statement of Ownership 2](#_Toc102488482)

[Previous Report Overview/ Differences 2](#_Toc102488483)

[Challenges with code 4](#_Toc102488484)

[Project overview 5](#_Toc102488485)

[Control Messages 5](#_Toc102488486)

[LoadBalancer1 6](#_Toc102488487)

[**Main Java Class** 6](#_Toc102488488)

[**Node1** 6](#_Toc102488489)

[**NodeManager1** 7](#_Toc102488490)

[**Server** 7](#_Toc102488491)

[Nodes 8](#_Toc102488492)

[**Main Java Class** 8](#_Toc102488493)

[**LoadBalancer** 8](#_Toc102488494)

[**Node** 9](#_Toc102488495)

[**jobThread** 9](#_Toc102488496)

[Initiator 10](#_Toc102488497)

[**Main Java Class** 10](#_Toc102488498)

[Arguments 11](#_Toc102488499)

[Bibliography 11](#_Toc102488500)

# **Statement of Ownership**

I Renato Dushku N0927786 hereby declare that the work that was done (code and report) and my own work. No parts of the code nor this report was compromised by copying from any other sources. The sources thar were used to aid with this coursework are the reference used in this report, the material that can be found on the module content and my own previous coursework (coursework 1).

# Diagram, engineering drawing Description automatically generated**Previous Report Overview/ Differences**

Overall as can be seen in Figure 1, the design of the algorithm was excessive, this means that there are a few features in which are irrelevant for a Round Robin algorithm and therefore suggesting that some of the aspects would need to be eliminated when creating the final version. This section will go in detail on which of those features are not needed/ were not implemented due to different types of limitations. As it was discussed on the previous report, the Round Robin algorithm will have a few different features. The structure of the Round Robin designed last time was implemented to have a registration message, job message, new job message, a node, a new node and an error section. As it was discussed on the previous report, the way that the Round Robin would work would be by having three different types of messages. The first type of message would be a register, this type of message would serve as a node registration process. The Round Robin algorithm will reserve a place in the node table where it would check for potential space for the node to be added and then a message will be returned to confirm that is added. Once the reservation is acknowledged it will save the instructions to the database.

Figure 1: Round Robin Algorithm Previous Implementation

Once a JOB message is sent its content will be examined by the load balancer this differs from the final product as once a job message is received then it will be ran and completed without the user knowing any other detail that happens within the process, however the user would be notified when a job comes through with relevant data (e.g. job ID, node ID etc.) and once a job is finished via a notification. In the previous report it was suggested that the algorithm would contain a decoding function view encrypted data, however that feature was not implemented in the final version. The job will in this case move to the next available node, however this cannot be implemented fully due to the submission requirements limitation (cannot have more than 3 projects), this means that I was unable to implement a variety of nodes for the algorithm as I would needed to have more than 3 projects in total, however this can be fixed once its ran meaning that the same node project can be ran in multiple machines acting as multiple nodes (same as the initiator acting as multiple clients). Once the job is completed a notification will be displayed.

As can be seen in Figure 1, the scheduler feature is completely irrelevant of this and as it was mentioned previously mentioned it was as a visual affect to show the reader how it would work and therefore it is clear that it is not implemented as a feature, however it is implemented as a method.

The node would act as the host where the messages come and get completed. The node information will be displayed to the user through a print message, this is because it will identify the node in which will undertake the job message. The capacity of the nodes is not important on Round Robin and therefore jobs would be constantly sent to the nodes in a chronological order. The only difference with the final implementation is that there is currently are 2 nodes, this conflicts with the design in Figure 1 as it was intended to have multiple node, this however could be fixed by running the node project in multiple different machines (same as initiator). This can also be fixed by duplicating the code within the same file and changing the variable names, this will create a new node in which the job can go through. The error section still remains as there are numerous error checks that would be mentioned at a later point.

# **Challenges with code**

One of the main problem with the projects were the implementation of code from the labs to the actual programming file. This means that some of the code that was taken from the labs needed to be further implemented and designed and therefore I used a book to help me with some of the features that I added in case the code from the lab did not run. The major problem that occurred with the code was that I could initially only call the IP address automatically (dynamically) only on the “initiator” project whereas the other two projects were crashing when I tried to call the IP using the same code. This was fixed by modifying the code and retrying it. An encoding section was intended to be implemented in the code, however I could not find any recourses within the lab sessions and therefore it was scrapped as it would have meant Google would have need to be used.

# **Project overview**

This project will be used to demonstrate the algorithm, this will be accomplished by coding the system itself using Java. There were two algorithms that could have been coded and the one that was chosen was the Round Robin algorithm. As it was discussed on the previous report “Round-Robin Load Balancing is a simple approach for distributing client requests over a number of servers. A client message is sent to the network servers one at a time and the algorithm prompts the load balancer to return back at the start of the procedure which will then allow the process to be repeated” (R.Dushku, 2022.). This report will go in depth on how the project is coded and what does the code mean (for each of the 3 projects), how it differed from the previous designed discussed on coursework 1 and what were the challenges with the report.

# **Control Messages**

There should be messages that will ensure that the code is actually running or if there is an error that will not be seen by having a syntax error occurring. Those messages include an error message for the packet, error message for the socket, error message for the Input/ Output, error message if the value of something is outside the set numbers, error message if the IP is wrong, error message if the port is out of range, error message for a wrong job ID and name and error message when the projects do not communicate.

There are also other messages used as control messages, these would be when the system functions accordingly. This message will include a registration message, a job message, a job completion message, a node message and other details in which will include more details about the job message and its size and node. This would be used to ensure that everything is working accordingly.

# **LoadBalancer1**

## **Main Java Class**

The main java class on the load balancer is used to run a thread in which would allow to run the server class forever until it is manually interrupted by the user, this is important because the server would need to be running in order to host the nodes in which will take the jobs to complete them. As can be seen in Figure 2, once the server is called by the new variable name “s”, then the thread will start.

Figure 2: Load Balancer

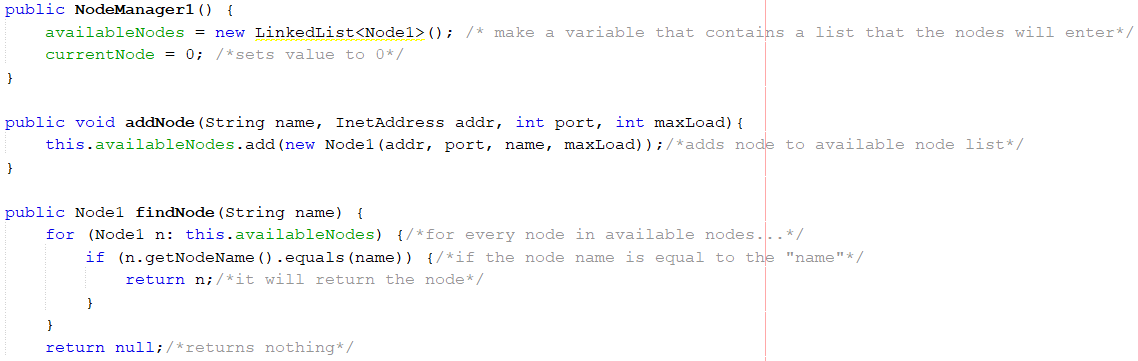
## **Node1**

Figure 3: Load Balancer Node

As can be seen in Figure 3, this java class is used to communicate with the Node project. This class sets the arguments of the node (e.g. node name, port number max jobs and current jobs) which would then be called by the node project. There are also error checks within the code that would be used to notify the user if the node name is not found, this would not crash the code by showing a syntax error however it will print a string stating the issue and then the code would stop running. At the end of this java class there are some error checks which are socket exemption and Input/ Output exemption checks. These are used to check if there is an issue with the socket, for example if the socket has not been found or the code does not call it efficiently. The same is used for the IO exemption.

## **NodeManager1**

Figure 4: Load Balancer Node Manager

The node manager is used to manage the nodes of the system. This would control how the nodes operate and it will manage them, this means that it will find and add the available nodes and set them as available for the jobs to be sent. This will also find the next available node and make sure that the job will recognise it.

## Graphical user interface, text, application Description automatically generated**Server**

Figure 5: Load Balancer Server

The Server connects to the node manager and sets the IP and port number of the load balancer. It will then get the length of the packet data from the datagram packet. Once the job has been received it will be printed on the load balancer output box showing the job name, size and node port. The server is also responsible for the creation of the register message and the job message, however the job message will be later send by the initiator (client). There are also error checks to ensure that the code functions accordingly.

# **Nodes**

## Text Description automatically generated with low confidence**Main Java Class**

This class would be used to run the nodes that will host the jobs and run them until the jobs are completed. This class will create the different nodes and set them so that they are available to be picked up by the algorithm. This will also create a register message in which will be sent to the load balancer and be displayed once the load balancer itself is ran. This will also create and run the job message that was sent via the initiator. There is also an error catching section which would be used to check if the socket is down and not working accordingly. The node information such as the IP address, port number and node name are called through the command arguments which are set for the project.

Figure 6: Node Main Java Class

## **LoadBalancer**

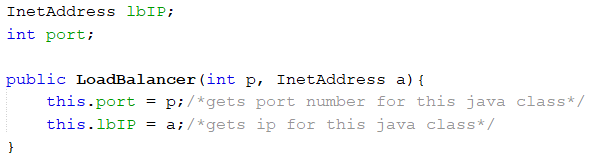
As can be seen by Figure 7, the load balancer IP and port number are set to have some parameters (e.g. the port number is an integer). Once this is done then a class with some elements/ parameters is set. Then the port number and IP address is taken from this java class and sets it to be a value of the java class parameter.

Figure 7: Node Load Balancer

## **Node**

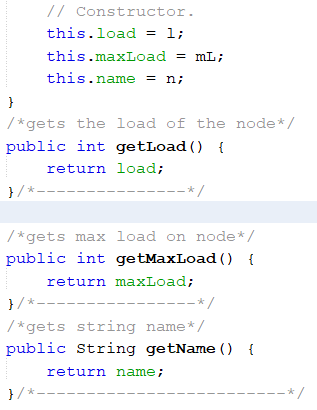
As can be seen in Figure 8, this is the constructor in which would be used to get the load of the node, maximum load/ capacity of the node and the name of the node. this would be helpful when setting the load of the node as it would mean that the user would be able to see how many jobs the node has. The max load would also be used to determine the capacity of a node which is useful as it will allow the user to identify how many jobs a node can have simultaneously. The name of the node would be used as an ID to identify which node undertook the job message.

Figure 8: Node Constructor

## **jobThread**

Graphical user interface, text, application

Description automatically generatedAs can be seen in Figure 9, a thread is created to run for 1000 milliseconds. This thread is used to initiate jobs to the nodes by adding the size of the job (in this case the size is used in seconds, so a size of 5 would be a job that would run for 5 seconds). This thread will then display a message once the job is completed notifying the user that the job is done.

Figure 9: JOB Threading

# **Initiator**

## Graphical user interface, text, application Description automatically generated**Main Java Class**

Figure 10: Initiator

As can be seen in Figure 10, the Initiator is used to sent the jobs to the nodes (acting as a client). There are some characteristics set for the code to ensure that the code works appropriately, for example there is a range of ports that can be scanned from the machine to occupy and operate from and if the input or search is out of range then it would return an error message to notify the user. The static void class serves as a method that will allow me to control the accessibility of members of a class When a class member is prompted by public, it can be obtained by code that is not part of the class where it is announced. Then the IP of the initiator would be extracted from the arguments list and the “0” element would be called. There is a range of 0-15 seconds which indicates that the job can be within those numbers in order for the user not to receive an error message. The last bit of the code is an error checking section by an invalid IP error, catching an error with the sockets and catching an error while sending a packet.

# **Arguments**

As can be seen by Figure 11, these are the parameters that are set for each project. Figure 11 arguments are of the Node Project. The other project would differ (however have a few similar stuff) because they have different parameters that are needed.

Figure 11: Arguments Configuration

# **Bibliography**

Horton, I., n.d. *Beginning Java 2*. 2nd ed. Birmingham, UK: Wrox Press Ltd., pp.482-488.