Report DS Discovery

Alexander Michielsen, Group: **#b10197**

For this part of the lab, we split the task among each other.

I was responsible for implementing the Discovery and Bootstrap functionality.

Vital was responsible for Failure

Asif for shutdown

Louis for the Running state (pinging)

First I will quickly go over what we as a team achieved. Then, I will go deeper into my work specifically

Global achievements

Before going into the workings of our system, we determined which ports we were going to use. The following is an overview:

```
User web interface: 80
REST: 8080
MC: 5000
TCP file transfer: 5044
```

We managed to implement a system in which nodes can join and configure themselves with the help of the other nodes already in the network (Bootstrap+Discovery). (Will be talked about extensively later on).

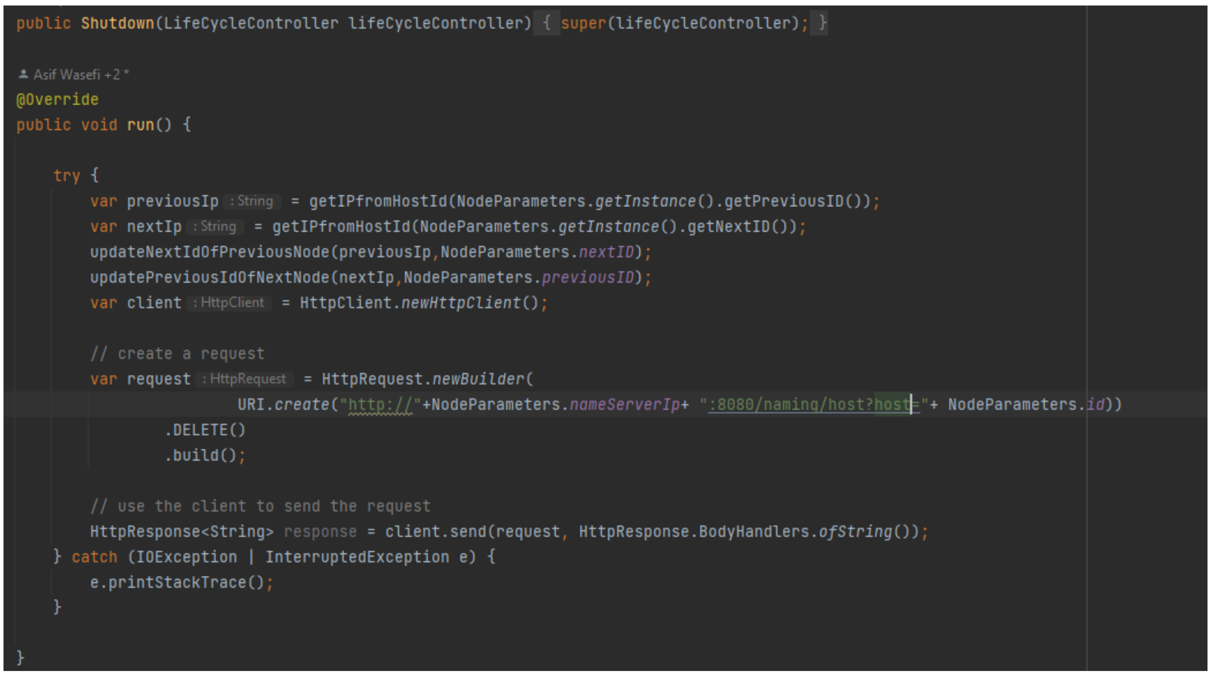
After this setup the node will do two things: Listen for multicasts of new nodes, ping its neighbours.

Multicast: This is just a simple MulticastSocket. This was implemented by me, so I'll talk about it more extensively later on

Pinging: Whenever a node is running and set up, it pings it neighbours in 1 second intervals. This is done by sending a REST request to the node, requesting its status. Whenever a ping fails 5 times, the neighbour is deemed to have failed and the failure sequence is executed.

Whenever a node fails, the other nodes detect this and execute the tasks necessary for the network to recover. I also worked quite extensively on this, so later on more.

Then there is also a shutdown state. In this case, the node sends to its neighbours that they have to update their previous and next node because this node is going to shut down. And it also lets the server know that this node should be deleted from the list. It wasn't finished in time so after helping with the failure I finished this as well. It is only part of the failure so was quite easy.



```
public Shutdown(LifeCycleController lifeCycleController) { super(lifeCycleController); }

Asif Wasefi +2 *
@Override
public void run() {

    try {
        var previousIp :String = getIPfromHostId(NodeParameters.getInstance().getPreviousID());
        var nextIp :String = getIPfromHostId(NodeParameters.getInstance().getNextID());
        updateNextIdOfPreviousNode(previousIp,NodeParameters.nextID);
        updatePreviousIdOfNextNode(nextIp,NodeParameters.previousID);
        var client :HttpClient = HttpClient.newHttpClient();

        // create a request
        var request :HttpRequest = HttpRequest.newBuilder(
                        URI.create("http://"+NodeParameters.nameServerIp+ ":8080/naming/host?host="+ NodeParameters.id))
                .DELETE()
                .build();

        // use the client to send the request
        HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }

}
```

My work/achievements

First I'm going to talk about my main goal: Discover/bootstrap and then the other tasks I've helped with.

Attempt 1

On my first attempt I decided to implement a broadcast. The new Node would send out a broadcast to all nodes on the network so that it would certainly reach the Nameserver. However it quickly came to my attention that working with Multicast is better.

This is because a multicast is only sent to hosts that have joined the multicast group to which a multicast is being sent this reduces the overhead on the network.

Attempt 2

When implementing the multicast, I tried to do it using REST to streamline our whole process. However, this turned out to be way more difficult and complex than initially anticipated.

After spending way too much time reading Spring Integration documentation and experimenting with Message Channels, Gateways and a whole lot more I decided to send an email.

In the reply I was simply told to use the regular Java classes. This was a bit confusing to me because I was told otherwise in during the lab session.

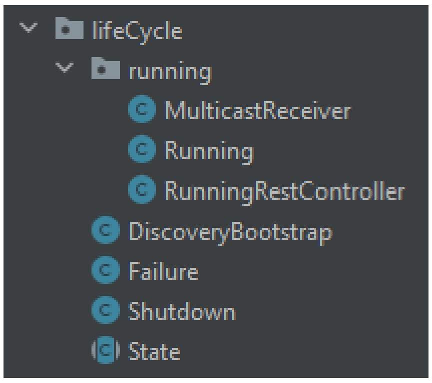
Attempt 3

Third time's a charm!

After some setbacks I decided to keep it nice and simple and use the regular Multicast sockets in Java.

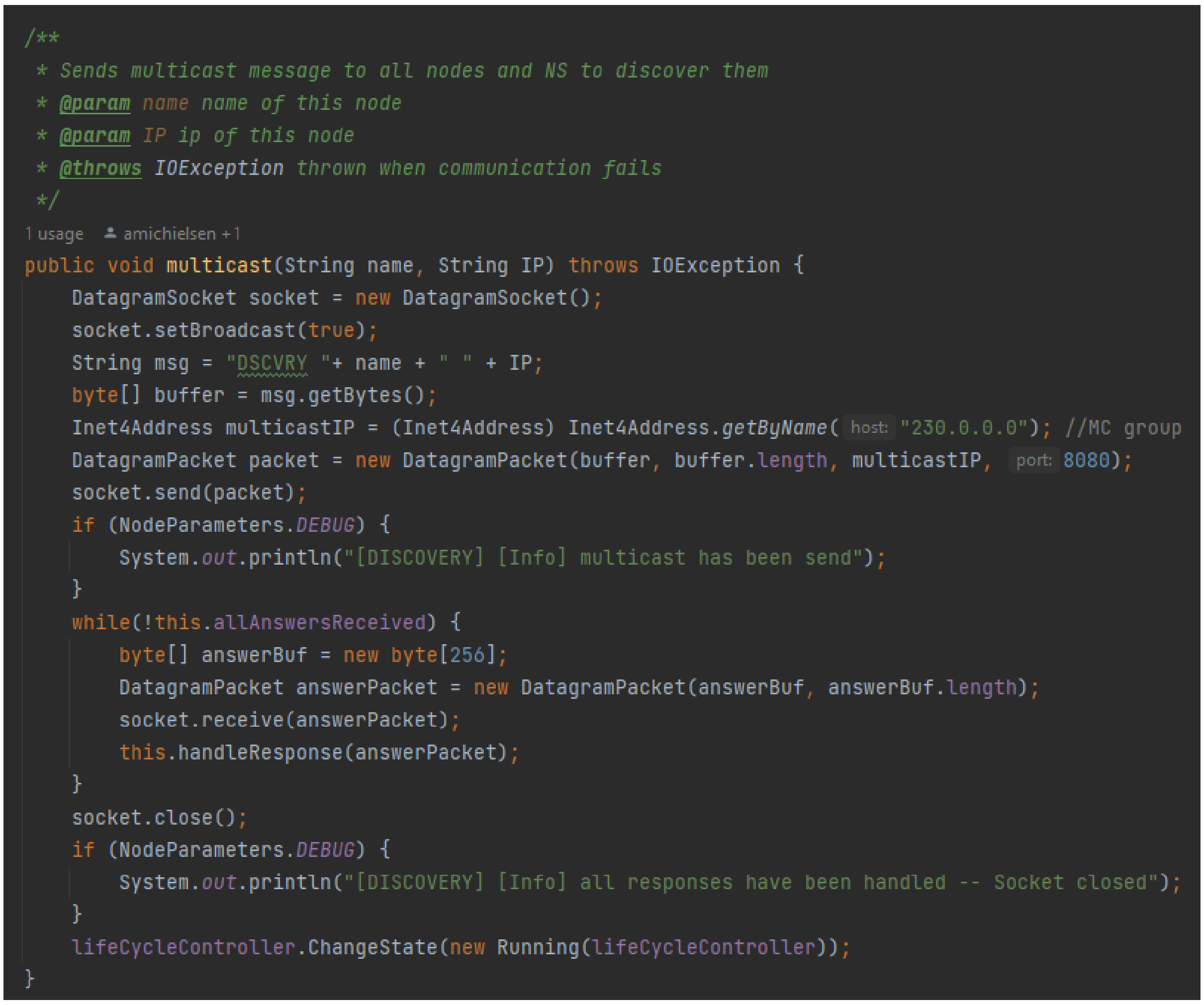
First off, sending a multicast from the newly added Node:

My teammate Louis already started working on the lifecycle of the node in which he implemented different states. My work was done in the DiscoveryBootstrap state



```
lifeCycle
    running
        MulticastReceiver
        Running
        RunningRestController
    DiscoveryBootstrap
    Failure
    Shutdown
    State
```

In this class I first created a simple method that sends multicasts

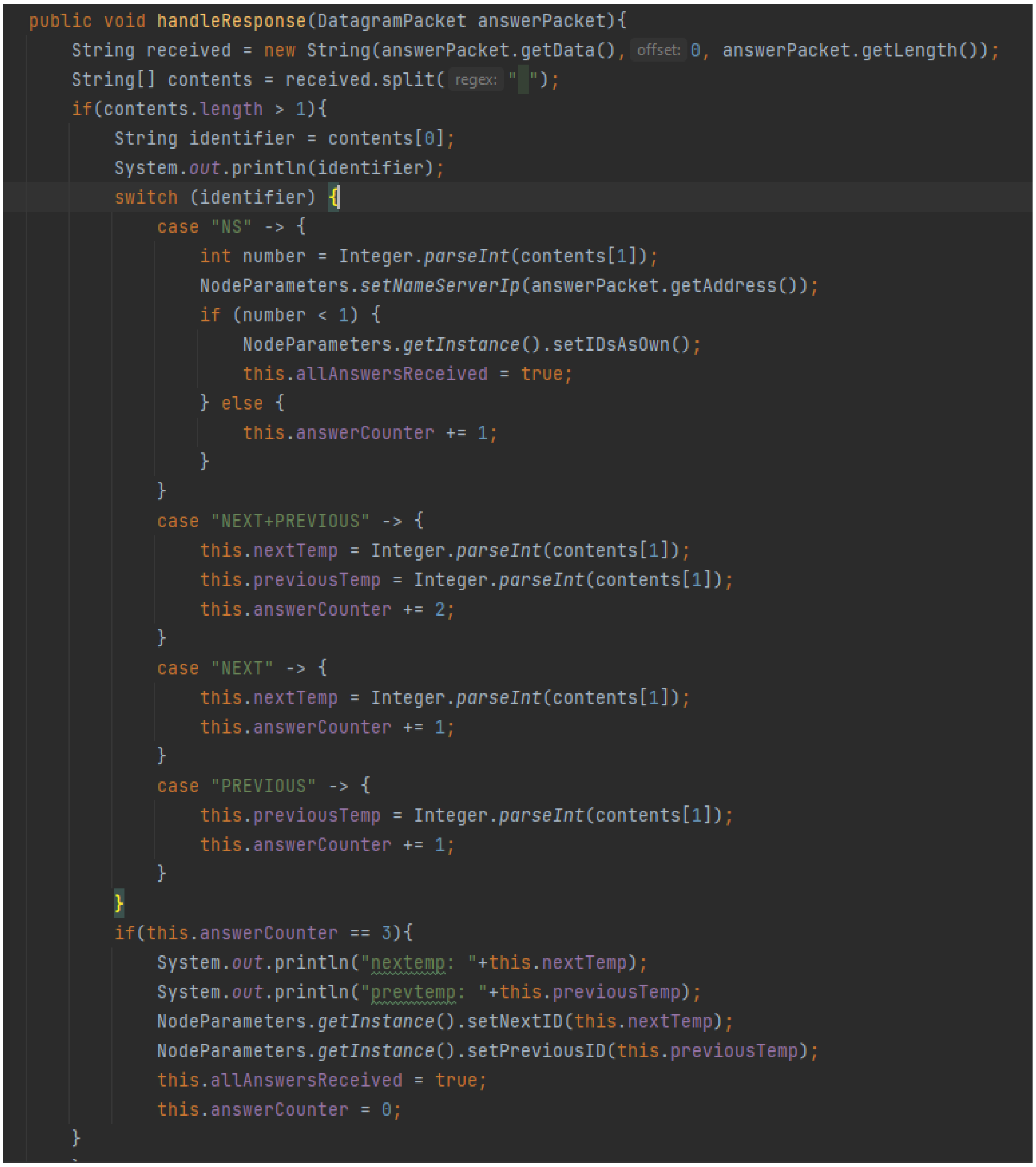


```java
/**
 * Sends multicast message to all nodes and NS to discover them
 * @param name name of this node
 * @param IP ip of this node
 * @throws IOException thrown when communication fails
 */
1 usage  amichielsen +1
public void multicast(String name, String IP) throws IOException {
    DatagramSocket socket = new DatagramSocket();
    socket.setBroadcast(true);
    String msg = "DSCVRY "+ name + " " + IP;
    byte[] buffer = msg.getBytes();
    Inet4Address multicastIP = (Inet4Address) Inet4Address.getByName( host: "230.0.0.0"); //MC group
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length, multicastIP,  port: 8080);
    socket.send(packet);
    if (NodeParameters.DEBUG) {
        System.out.println("[DISCOVERY] [Info] multicast has been send");
    }
    while(!this.allAnswersReceived) {
        byte[] answerBuf = new byte[256];
        DatagramPacket answerPacket = new DatagramPacket(answerBuf, answerBuf.length);
        socket.receive(answerPacket);
        this.handleResponse(answerPacket);
    }
    socket.close();
    if (NodeParameters.DEBUG) {
        System.out.println("[DISCOVERY] [Info] all responses have been handled -- Socket closed");
    }
    lifeCycleController.ChangeState(new Running(lifeCycleController));
}
```

For this I use a regular DatagramSocket that sends it message to the IP of the MC group. The contents are its name and IP address

To maintain clear communication, we decided to add an identifier to the beginning of all of our messages. In this case it was “DSCVRY” to indicate this was a Discovery message.

After sending the multicast the Node would wait on the responses.



```java
public void handleResponse(DatagramPacket answerPacket){
    String received = new String(answerPacket.getData(), offset: 0, answerPacket.getLength());
    String[] contents = received.split( regex: " ");
    if(contents.length > 1){
        String identifier = contents[0];
        System.out.println(identifier);
        switch (identifier) {
            case "NS" -> {
                int number = Integer.parseInt(contents[1]);
                NodeParameters.setNameServerIp(answerPacket.getAddress());
                if (number < 1) {
                    NodeParameters.getInstance().setIDsAsOwn();
                    this.allAnswersReceived = true;
                } else {
                    this.answerCounter += 1;
                }
            }
            case "NEXT+PREVIOUS" -> {
                this.nextTemp = Integer.parseInt(contents[1]);
                this.previousTemp = Integer.parseInt(contents[1]);
                this.answerCounter += 2;
            }
            case "NEXT" -> {
                this.nextTemp = Integer.parseInt(contents[1]);
                this.answerCounter += 1;
            }
            case "PREVIOUS" -> {
                this.previousTemp = Integer.parseInt(contents[1]);
                this.answerCounter += 1;
            }
        }
        if(this.answerCounter == 3){
            System.out.println("nextemp: "+this.nextTemp);
            System.out.println("prevtemp: "+this.previousTemp);
            NodeParameters.getInstance().setNextID(this.nextTemp);
            NodeParameters.getInstance().setPreviousID(this.previousTemp);
            this.allAnswersReceived = true;
            this.answerCounter = 0;
        }
    }
```

Every message received back would be checked to be a valid response.

First the size was checked, if this is lower or equal to one it can't possible be a valid response according to the way we put together messages.

Then, because we use identifiers in all of our messages, we can check who had sent the packet

“NS” indicating Naming Server, the others in the list regular nodes.

If the Naming Server responded that the number of nodes was equal to 0, we didn't have to wait for other nodes to respond (because they are non-existent). So, we set the “allAnswersReceived” flag to true indicating that the Node doesn’t have to wait for more answers to arrive. Additionally, the next and previous node are set as its own ID

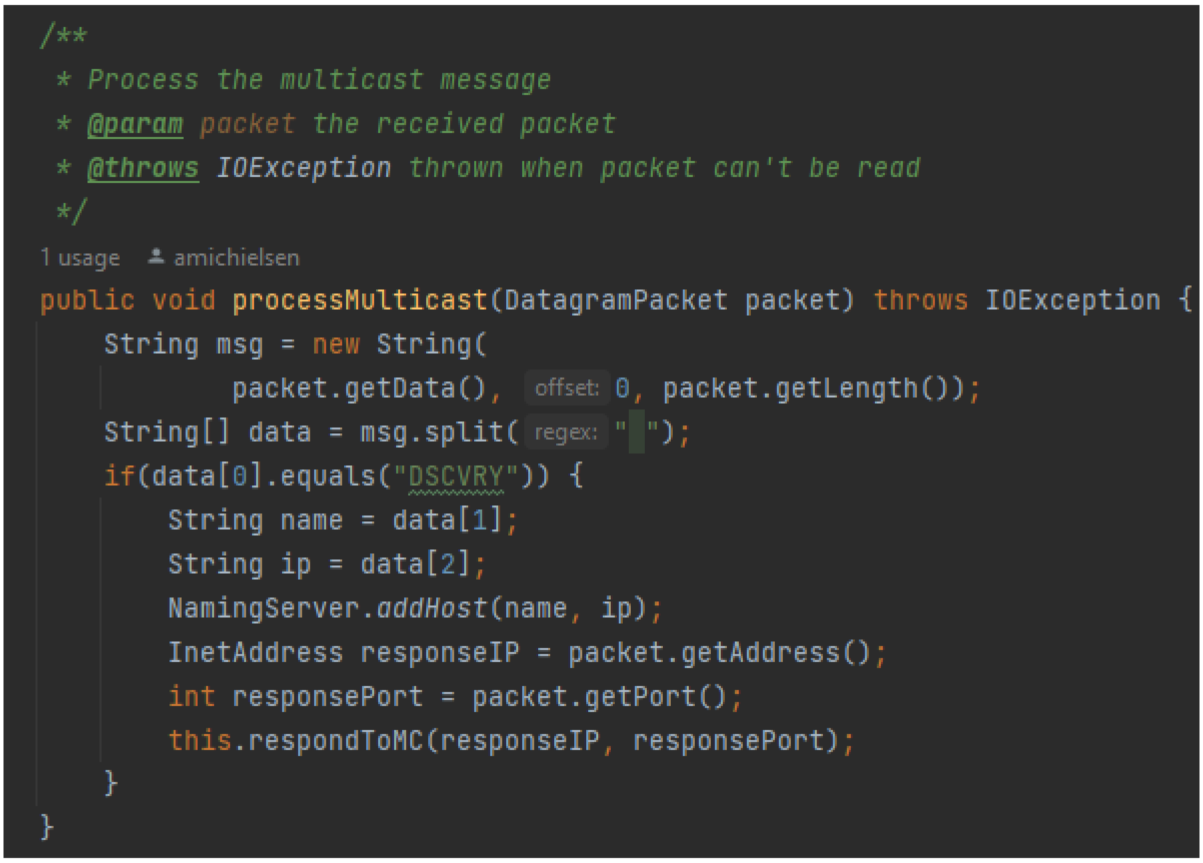
If the Naming Server responds that there is at least one node in the system, we wait for other nodes to respond. Depending on the identifier we wait for one (NEXT+PREVIOUS, if there is just one other node in the system) or two (NEXT and PREVIOUS) responses.

After all answers are received, we save the ID of the next and previous Node.

Now on the Naming Server side:

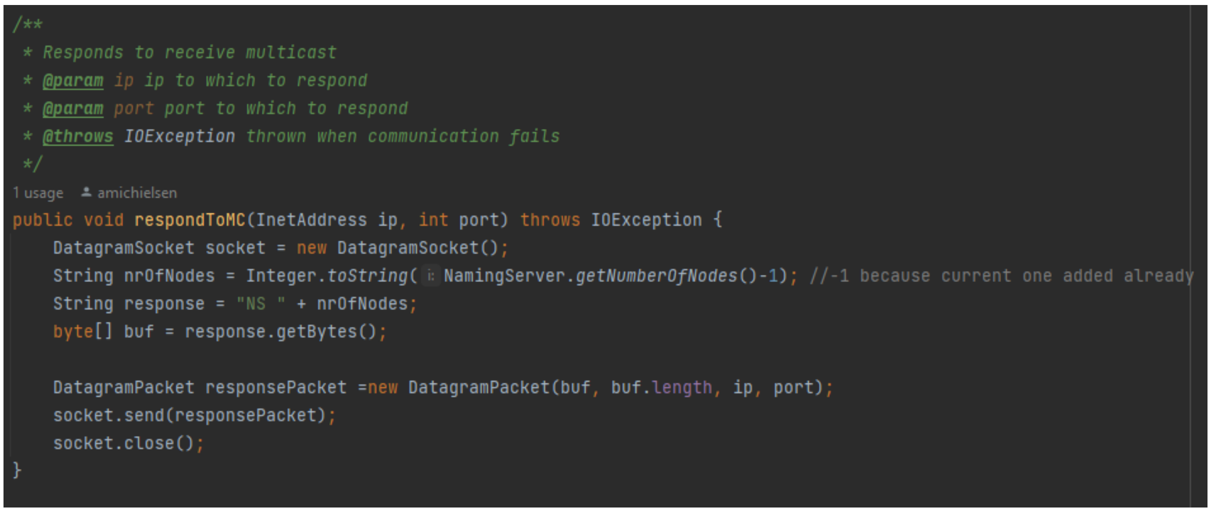
There is a MulticastListener constantly listening for packages. This listener has joined the predefined Multicast group as well.

Whenever a multicast destined for the group arrives, the server will firstly check its identifier. If it is equal to “DSCVRY” it will continue processing the contents, otherwise the package is dropped.



```java
/**
 * Process the multicast message
 * @param packet the received packet
 * @throws IOException thrown when packet can't be read
 */
1 usage   amichielsen
public void processMulticast(DatagramPacket packet) throws IOException {
    String msg = new String(
            packet.getData(), offset: 0, packet.getLength());
    String[] data = msg.split( regex: " ");
    if(data[0].equals("DSCVRY")) {
        String name = data[1];
        String ip = data[2];
        NamingServer.addHost(name, ip);
        InetAddress responseIP = packet.getAddress();
        int responsePort = packet.getPort();
        this.respondToMC(responseIP, responsePort);
    }
}
```

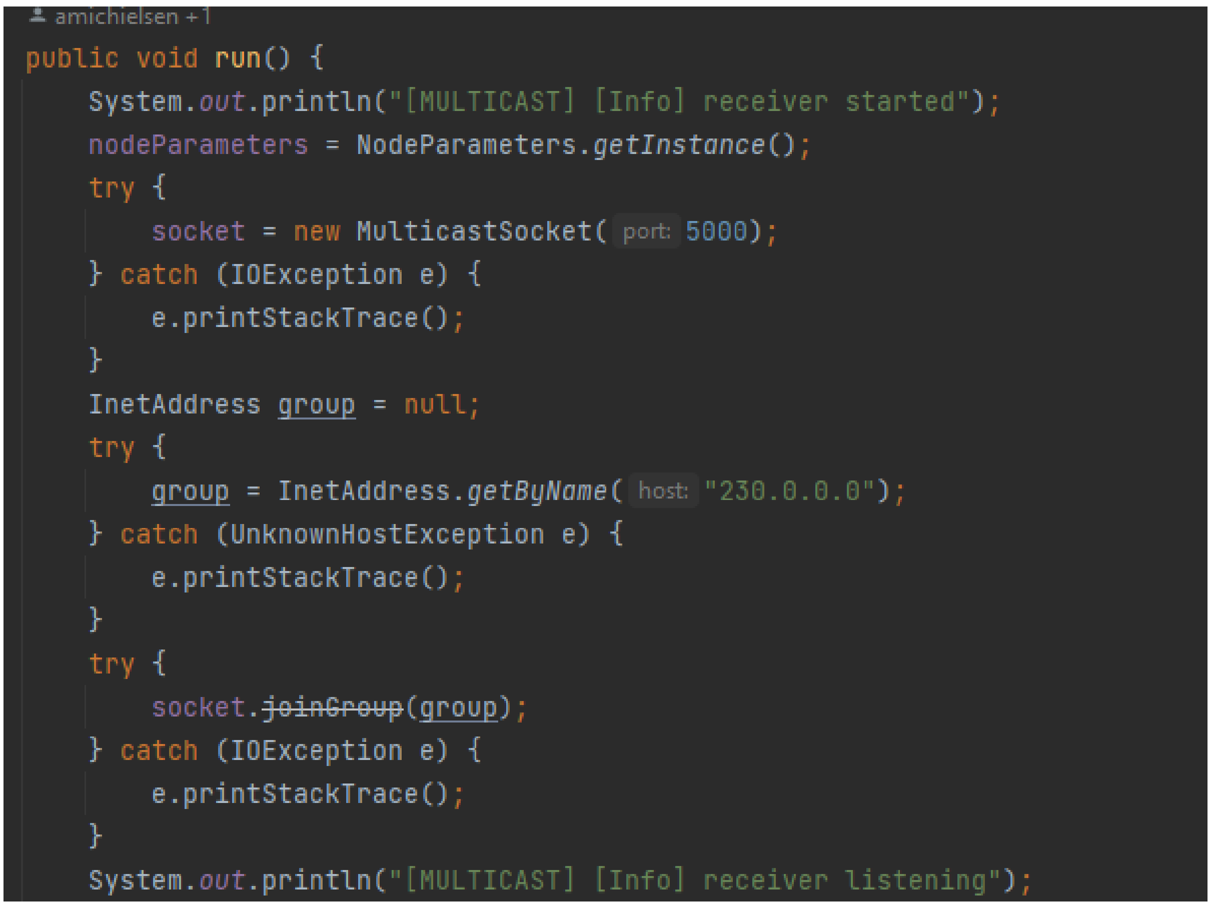
The server adds the name and IP to the list, using the hash that we already implemented in previous sessions and sends a response.



```java
/**
 * Responds to receive multicast
 * @param ip ip to which to respond
 * @param port port to which to respond
 * @throws IOException thrown when communication fails
 */
1 usage  amichielsen
public void respondToMC(InetAddress ip, int port) throws IOException {
    DatagramSocket socket = new DatagramSocket();
    String nrOfNodes = Integer.toString( i: NamingServer.getNumberOfNodes()-1); //-1 because current one added already
    String response = "NS " + nrOfNodes;
    byte[] buf = response.getBytes();

    DatagramPacket responsePacket =new DatagramPacket(buf, buf.length, ip, port);
    socket.send(responsePacket);
    socket.close();
}
```

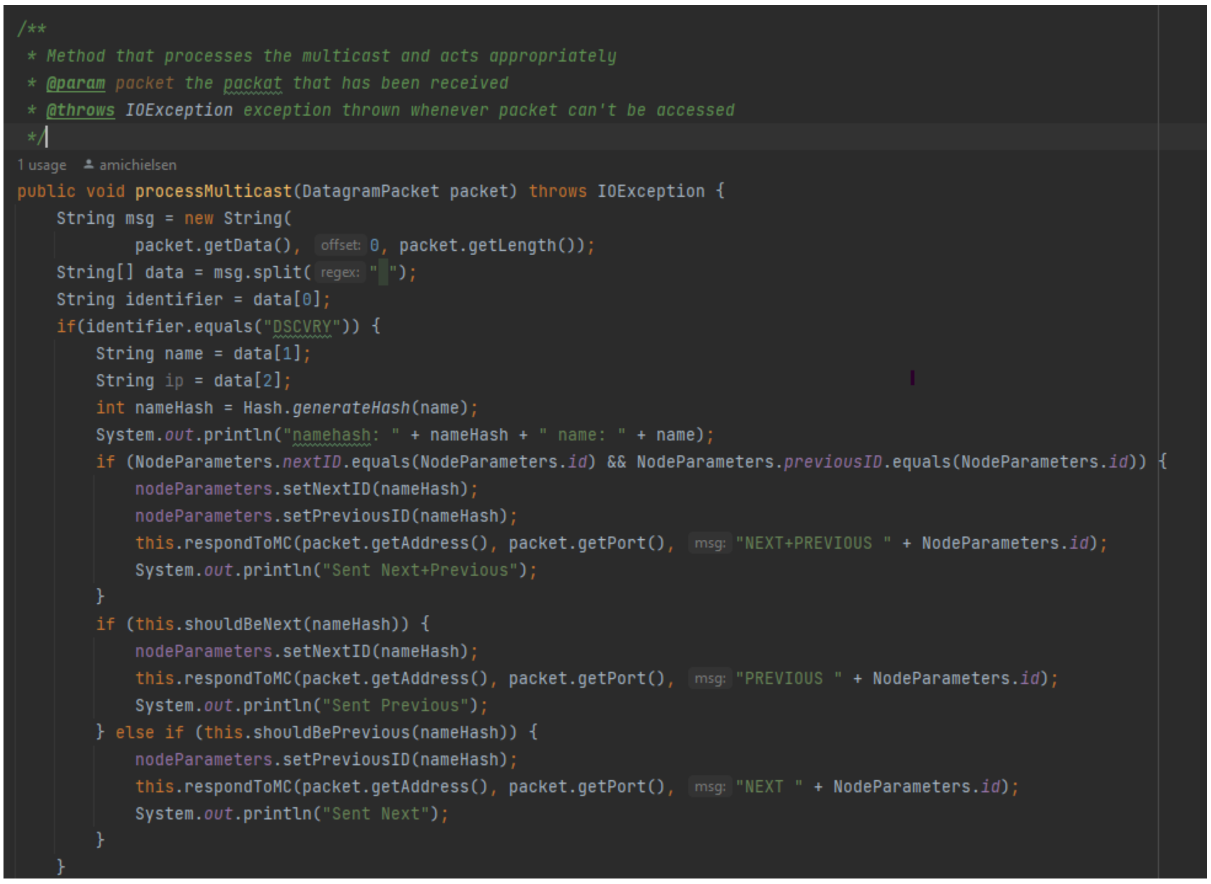
Using the “NS” identifier as the start of the message it sends back the number of nodes already in the network (before this one joined).

Once a node has finished setting Bootstrap and it knows its next and previous node, it starts in the running state in which it joins the same multicast group and also listens for Multicast messages.



```java
 amichielsen +1
public void run() {
    System.out.println("[MULTICAST] [Info] receiver started");
    nodeParameters = NodeParameters.getInstance();
    try {
        socket = new MulticastSocket( port: 5000);
    } catch (IOException e) {
        e.printStackTrace();
    }
    InetAddress group = null;
    try {
        group = InetAddress.getByName( host: "230.0.0.0");
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
    try {
        socket.~~joinGroup~~(group);
    } catch (IOException e) {
        e.printStackTrace();
    }
    System.out.println("[MULTICAST] [Info] receiver listening");
```

It’s a separate thread that waits for a multicast package.



```java
/**
 * Method that processes the multicast and acts appropriately
 * @param packet the packat that has been received
 * @throws IOException exception thrown whenever packet can't be accessed
 */
1 usage  amichielsen
public void processMulticast(DatagramPacket packet) throws IOException {
    String msg = new String(
            packet.getData(), offset: 0, packet.getLength());
    String[] data = msg.split( regex: " ");
    String identifier = data[0];
    if(identifier.equals("DSCVRY")) {
        String name = data[1];
        String ip = data[2];
        int nameHash = Hash.generateHash(name);
        System.out.println("namehash: " + nameHash + " name: " + name);
        if (NodeParameters.nextID.equals(NodeParameters.id) && NodeParameters.previousID.equals(NodeParameters.id)) {
            nodeParameters.setNextID(nameHash);
            nodeParameters.setPreviousID(nameHash);
            this.respondToMC(packet.getAddress(), packet.getPort(), msg: "NEXT+PREVIOUS " + NodeParameters.id);
            System.out.println("Sent Next+Previous");
        }
        if (this.shouldBeNext(nameHash)) {
            nodeParameters.setNextID(nameHash);
            this.respondToMC(packet.getAddress(), packet.getPort(), msg: "PREVIOUS " + NodeParameters.id);
            System.out.println("Sent Previous");
        } else if (this.shouldBePrevious(nameHash)) {
            nodeParameters.setPreviousID(nameHash);
            this.respondToMC(packet.getAddress(), packet.getPort(), msg: "NEXT " + NodeParameters.id);
            System.out.println("Sent Next");
        }
    }
```

When a multicast arrives, its identifier is checked first.

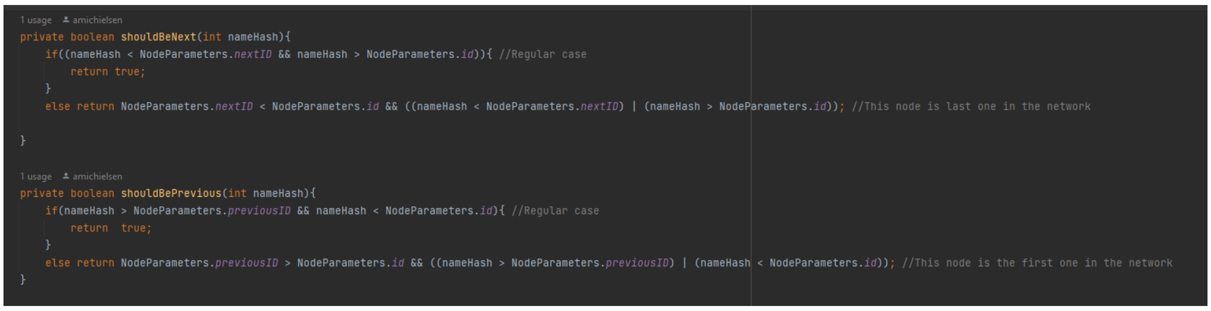
Afterwards, the node will calculate whether or not this new node should be one of its neighbours.

To do this, it hashes the name using our previously built hashing method.

Then, it does a couple of checks. If current next and previous nodes on the network are its own ID, the node will respond by saying he is the next and previous one of the new node.

Otherwise it will test whether the new node would take the place of one of its existing neighbours (by checking the hash value of the node's hostname), updates its neighbours and sends back a response saying "I'm your next/previous".

These checks also take into account that the current node might be the last or first one in the network.


```
1 usage  amichielsen
private boolean shouldBeNext(int nameHash){
    if((nameHash < NodeParameters.nextID && nameHash > NodeParameters.id)){ //Regular case
        return true;
    }
    else return NodeParameters.nextID < NodeParameters.id && ((nameHash < NodeParameters.nextID) | (nameHash > NodeParameters.id)); //This node is last one in the network

}

1 usage  amichielsen
private boolean shouldBePrevious(int nameHash){
    if(nameHash > NodeParameters.previousID && nameHash < NodeParameters.id){ //Regular case
        return  true;
    }
    else return NodeParameters.previousID > NodeParameters.id && ((nameHash > NodeParameters.previousID) | (nameHash < NodeParameters.id)); //This node is the first one in the network
}
```

As can be seen, in the case that this node is the only one in the network (prev=own=next), it will send back that this node will be the next and previous node of the one that joined.

After finishing up the discovery and bootstrap, I tested my code and helped out my teammates with some things.

First the tests:

We use Java 17 for our whole project. Sadly enough the nodes we use can’t install the newest Maven version from apt-get install that supports Java 17.

Therefore I had to install it manually using this script I found on StackOverflow

```
TMP_MAVEN_VERSION=${1:-"3.8.4"}

# Download Maven
wget https://apache.org/dist/maven/maven-3/$TMP_MAVEN_VERSION/binaries/apache-maven-$T
MP_MAVEN_VERSION-bin.tar.gz -P /tmp

# Unzip
sudo tar xf /tmp/apache-maven-*.tar.gz -C /opt
sudo rm /tmp/apache-maven-*-bin.tar.gz
sudo ln -s /opt/apache-maven-$TMP_MAVEN_VERSION /opt/maven

# Setup environment variables
sudo touch /etc/profile.d/maven.sh
sudo chown ubuntu /etc/profile.d/maven.sh
sudo chmod +x /etc/profile.d/maven.sh
>> /etc/profile.d/maven.sh echo "export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-arm64"
>> /etc/profile.d/maven.sh echo "export M2_HOME=/opt/maven"
>> /etc/profile.d/maven.sh echo "export MAVEN_HOME=/opt/maven"
>> /etc/profile.d/maven.sh echo "export PATH=\$M2_HOME/bin:\$PATH"

sudo chmod +x /etc/profile.d/maven.sh
source /etc/profile.d/maven.sh
```

Afterwards I could run our project

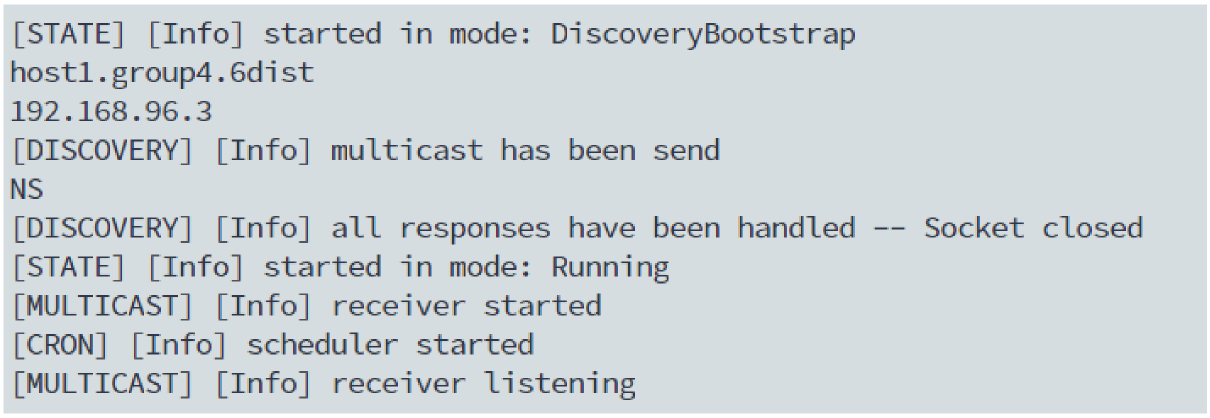
First, we launch our NamingServer; To do this, I wrote a little script that would set the PATH variable for Java, pull the latest release from Github, build it and run our project:

```
export JAVA_HOME=/usr/lib/jvm/java-1.17.0-openjdk-amd64
cd ./DS_Project/
git pull https://amichielsen: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@github.com/amich
iels>
cd ./NameServer
mvn clean install
cd ./target
java -jar NamingServer-0.0.1-SNAPSHOT.jar
```

With our NamingServer running, I started a Node using a similar script

```
export JAVA_HOME=/usr/lib/jvm/java-1.17.0-openjdk-amd64
cd ./DS_Project/
git pull https://amichielsen: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@github.com/amich
iels>
cd ./NameServer
mvn clean install
cd ./target
java -jar NamingServer-0.0.1-SNAPSHOT.jar
```

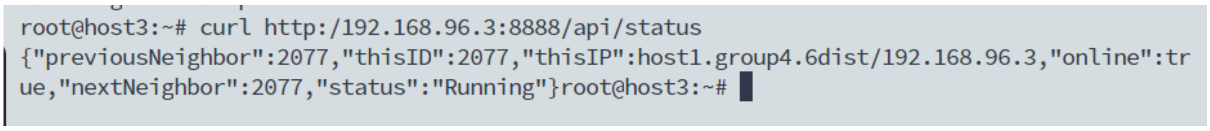
We see that Node starts up as expected and goes through Discovery perfectly



```
[STATE] [Info] started in mode: DiscoveryBootstrap
host1.group4.6dist
192.168.96.3
[DISCOVERY] [Info] multicast has been send
NS
[DISCOVERY] [Info] all responses have been handled -- Socket closed
[STATE] [Info] started in mode: Running
[MULTICAST] [Info] receiver started
[CRON] [Info] scheduler started
[MULTICAST] [Info] receiver listening
```

NS identifying that it received an answer from the Naming server. In this case, it mentioned that there were no other nodes in the system, thus all responses are handled

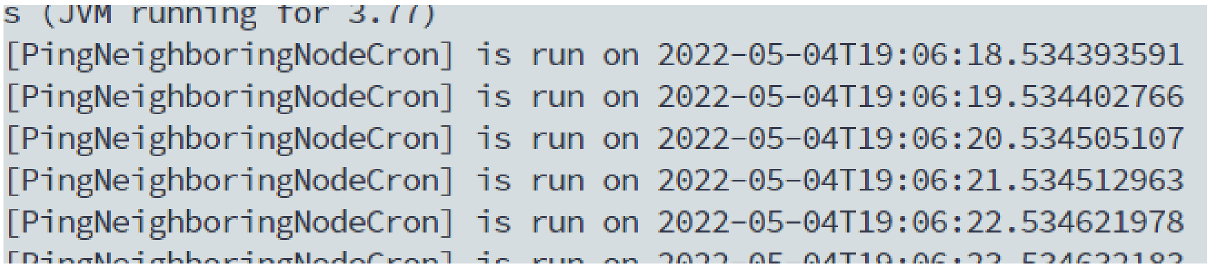
We check the data on this node by sending a REST request from another session to this Node



```
root@host3:~# curl http:/192.168.96.3:8888/api/status
{"previousNeighbor":2077,"thisID":2077,"thisIP":host1.group4.6dist/192.168.96.3,"online":tr
ue,"nextNeighbor":2077,"status":"Running"}root@host3:~#
```

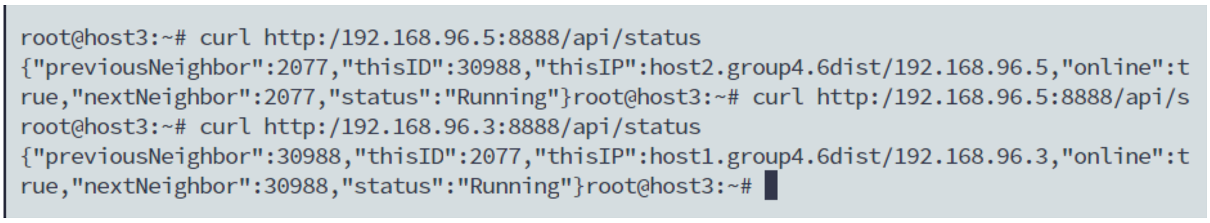
We see that the neighbours are configured correctly (its own id)

We also see that the node is pinging its neighbours (itself) correctly.



```
s (JVM running for 3.77)
[PingNeighboringNodeCron] is run on 2022-05-04T19:06:18.534393591
[PingNeighboringNodeCron] is run on 2022-05-04T19:06:19.534402766
[PingNeighboringNodeCron] is run on 2022-05-04T19:06:20.534505107
[PingNeighboringNodeCron] is run on 2022-05-04T19:06:21.534512963
[PingNeighboringNodeCron] is run on 2022-05-04T19:06:22.534621978
```

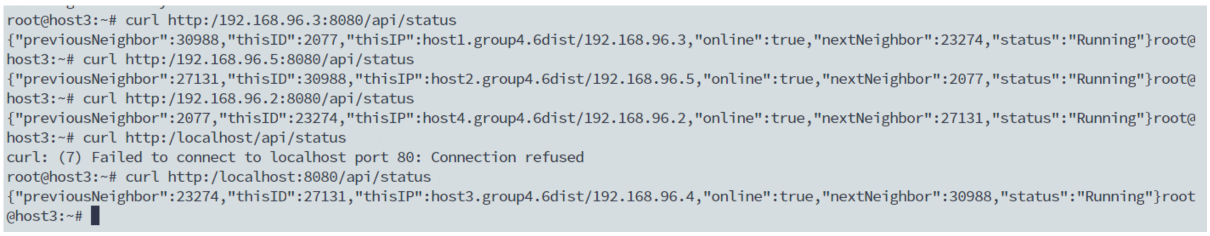
Then we start another node:



```
root@host3:~# curl http:/192.168.96.5:8888/api/status
{"previousNeighbor":2077,"thisID":30988,"thisIP":host2.group4.6dist/192.168.96.5,"online":t
rue,"nextNeighbor":2077,"status":"Running"}root@host3:~# curl http:/192.168.96.5:8888/api/s
root@host3:~# curl http:/192.168.96.3:8888/api/status
{"previousNeighbor":30988,"thisID":2077,"thisIP":host1.group4.6dist/192.168.96.3,"online":t
rue,"nextNeighbor":30988,"status":"Running"}root@host3:~#
```

On both nodes the neighbours are configured correctly!

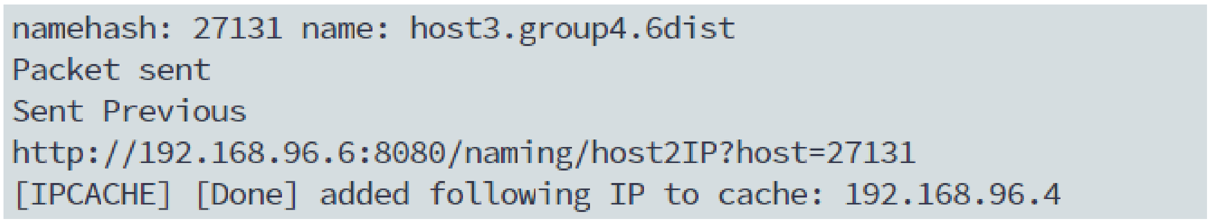
Then we added two more nodes and checked their configuration



```
root@host3:~# curl http:/192.168.96.3:8080/api/status
{"previousNeighbor":30988,"thisID":2077,"thisIP":host1.group4.6dist/192.168.96.3,"online":true,"nextNeighbor":23274,"status":"Running"}root@
host3:~# curl http:/192.168.96.5:8080/api/status
{"previousNeighbor":27131,"thisID":30988,"thisIP":host2.group4.6dist/192.168.96.5,"online":true,"nextNeighbor":2077,"status":"Running"}root@
host3:~# curl http:/192.168.96.2:8080/api/status
{"previousNeighbor":2077,"thisID":23274,"thisIP":host4.group4.6dist/192.168.96.2,"online":true,"nextNeighbor":27131,"status":"Running"}root@
host3:~# curl http:/localhost/api/status
curl: (7) Failed to connect to localhost port 80: Connection refused
root@host3:~# curl http:/localhost:8080/api/status
{"previousNeighbor":23274,"thisID":27131,"thisIP":host3.group4.6dist/192.168.96.4,"online":true,"nextNeighbor":30988,"status":"Running"}root
@host3:~#
```

All four nodes are configured correctly!

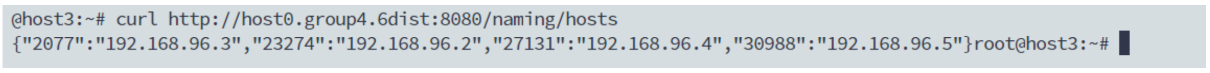
On the responding nodes we can see what has happened



```
namehash: 27131 name: host3.group4.6dist
Packet sent
Sent Previous
http://192.168.96.6:8080/naming/host2IP?host=27131
[IPCACHE] [Done] added following IP to cache: 192.168.96.4
```

This node for example, determined that it should be the previous one of then node that has sent the multicast. In addition it saves it IP address in a cache. If a ping fails to this IP, it will request the IP from the Naming Server

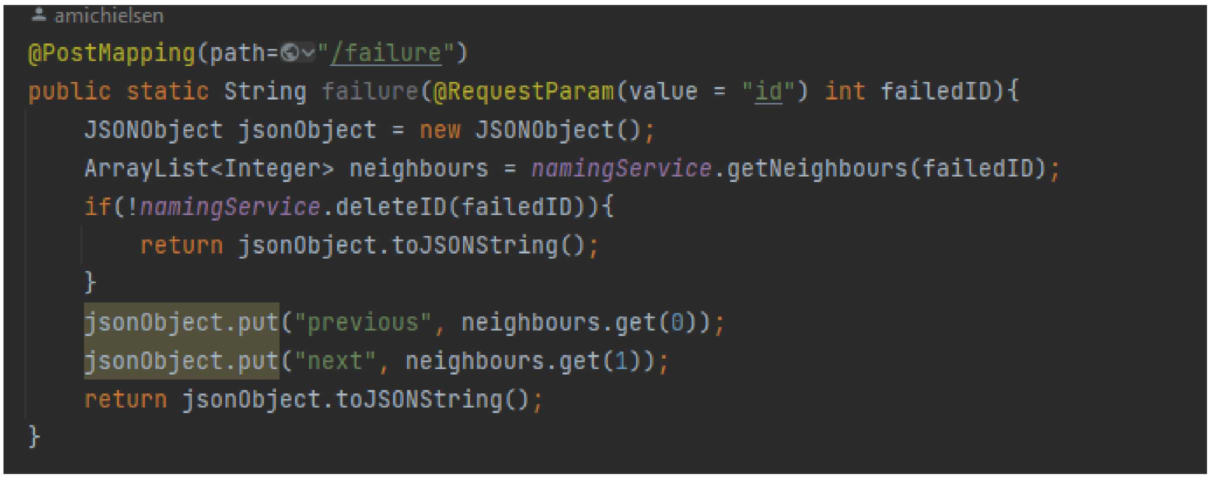
We can also check the current list of nodes on the nameserver:



```
@host3:~# curl http://host0.group4.6dist:8080/naming/hosts
{"2077":"192.168.96.3","23274":"192.168.96.2","27131":"192.168.96.4","30988":"192.168.96.5"}root@host3:~#
```

Then the extra work I did:

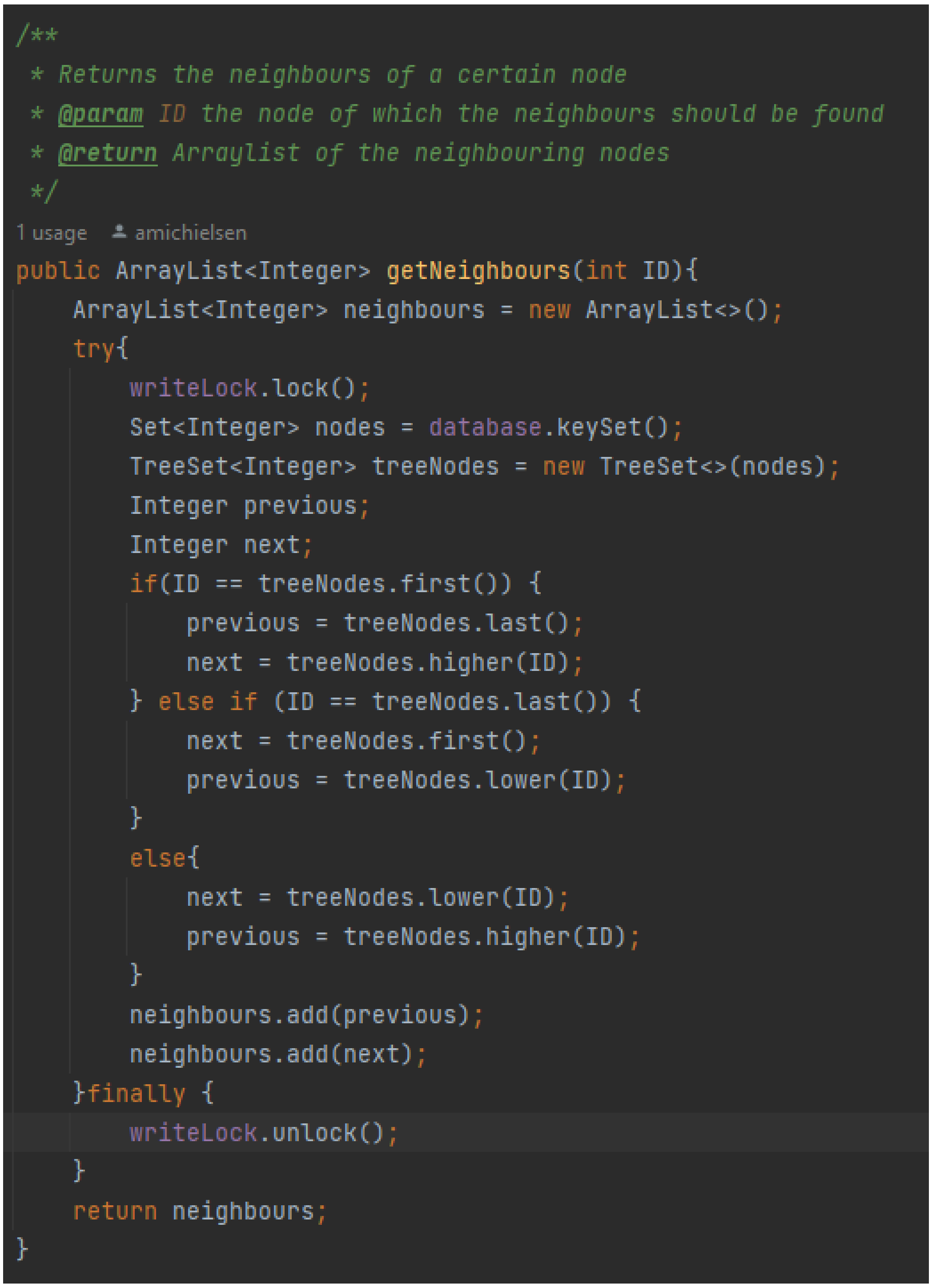
For the Failure, I Implemented a new REST request on the NameServer that returns the neighbours of a given node and deletes the failed node. With this information, the node that noticed the failure can inform the neighbours of the failed one with updated information. The nodes have built-in REST requests for updating its neighbours so these ones can be used.



```
amichielsen
@PostMapping(path="/failure")
public static String failure(@RequestParam(value = "id") int failedID){
    JSONObject jsonObject = new JSONObject();
    ArrayList<Integer> neighbours = namingService.getNeighbours(failedID);
    if(!namingService.deleteID(failedID)){
        return jsonObject.toJSONString();
    }
    jsonObject.put("previous", neighbours.get(0));
    jsonObject.put("next", neighbours.get(1));
    return jsonObject.toJSONString();
}
```

REST on naming server

The if-statement is there to make sure that when two nodes simultaneously notice another one has failed. The deletion of the node and sending back the neighbours only happens ones to one node. The other node will just receive an empty answer, indicating that another node has already detected the failure.

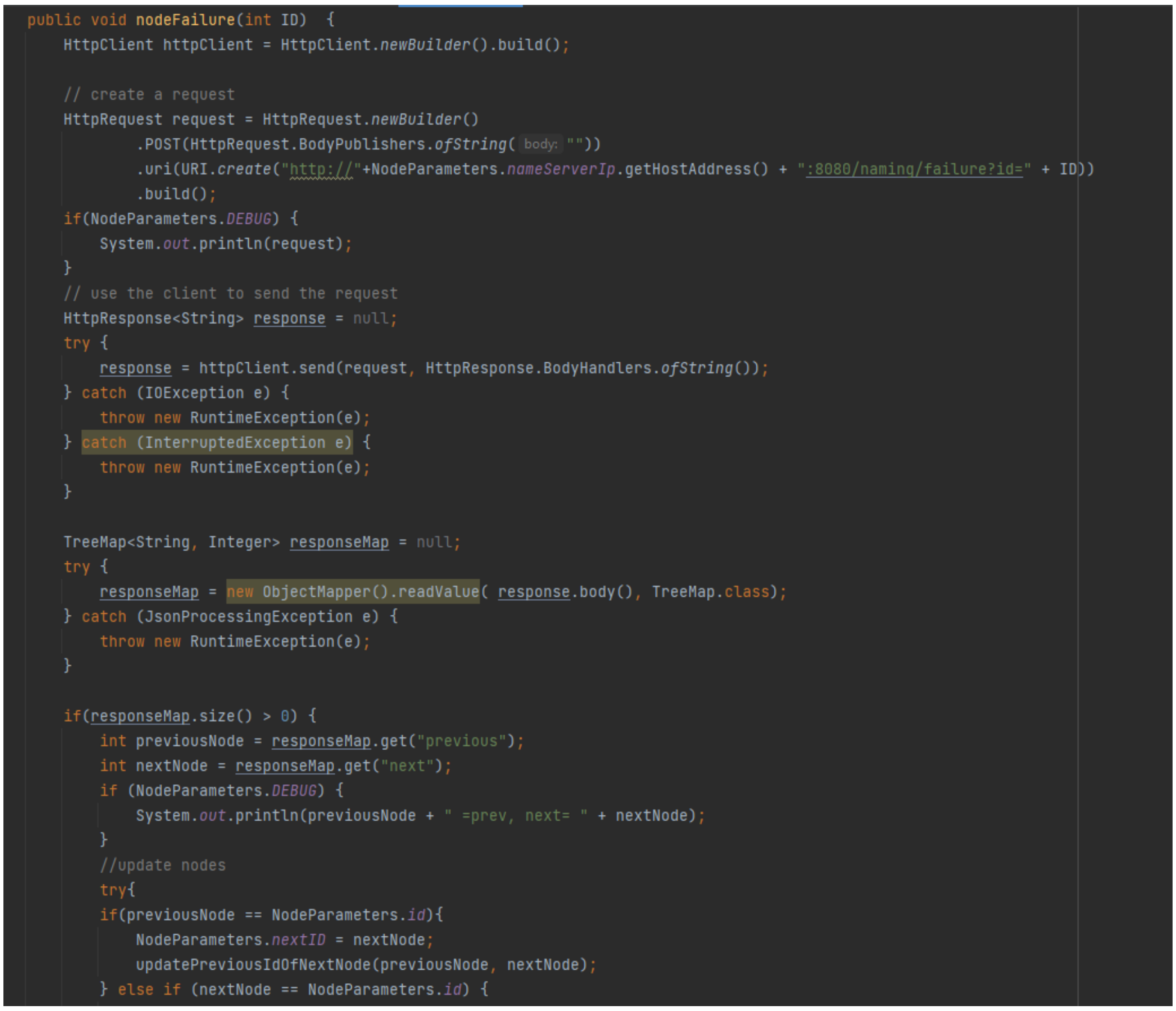

```java
/**
 * Returns the neighbours of a certain node
 * @param ID the node of which the neighbours should be found
 * @return Arraylist of the neighbouring nodes
 */
1 usage   amichielsen
public ArrayList<Integer> getNeighbours(int ID){
    ArrayList<Integer> neighbours = new ArrayList<>();
    try{
        writeLock.lock();
        Set<Integer> nodes = database.keySet();
        TreeSet<Integer> treeNodes = new TreeSet<>(nodes);
        Integer previous;
        Integer next;
        if(ID == treeNodes.first()) {
            previous = treeNodes.last();
            next = treeNodes.higher(ID);
        } else if (ID == treeNodes.last()) {
            next = treeNodes.first();
            previous = treeNodes.lower(ID);
        }
        else{
            next = treeNodes.lower(ID);
            previous = treeNodes.higher(ID);
        }
        neighbours.add(previous);
        neighbours.add(next);
    }finally {
        writeLock.unlock();
    }
    return neighbours;
}
```

Implementation on naming server

Implementation of method on nameserver

I also takes into account the edge cases that a node is the first or last one (lowest and highest). In which case it returns the value from the other side of the list as can be seen above. I first thought about taking into account the case that there is just one node. However, in that case there wouldn't be another node to detect the failed node, so it wouldn't make sense.

Then, I also helped with the handling of the failure on the node itself.



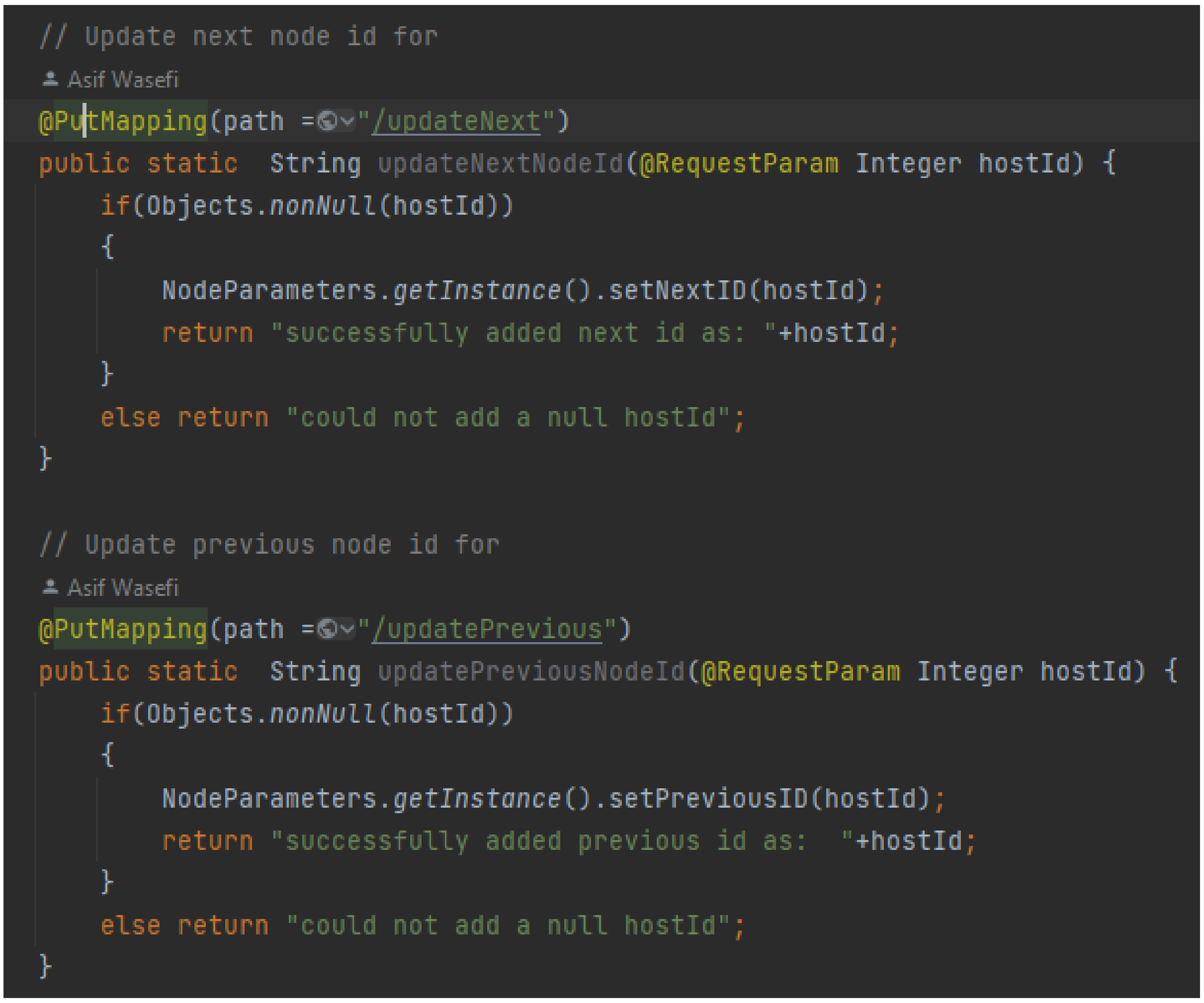
```java
public void nodeFailure(int ID)  {
    HttpClient httpClient = HttpClient.newBuilder().build();

    // create a request
    HttpRequest request = HttpRequest.newBuilder()
            .POST(HttpRequest.BodyPublishers.ofString( body: ""))
            .uri(URI.create("http://"+NodeParameters.nameServerIp.getHostAddress() + ":8080/naming/failure?id=" + ID))
            .build();
    if(NodeParameters.DEBUG) {
        System.out.println(request);
    }
    // use the client to send the request
    HttpResponse<String> response = null;
    try {
        response = httpClient.send(request, HttpResponse.BodyHandlers.ofString());
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }

    TreeMap<String, Integer> responseMap = null;
    try {
        responseMap = new ObjectMapper().readValue( response.body(), TreeMap.class);
    } catch (JsonProcessingException e) {
        throw new RuntimeException(e);
    }

    if(responseMap.size() > 0) {
        int previousNode = responseMap.get("previous");
        int nextNode = responseMap.get("next");
        if (NodeParameters.DEBUG) {
            System.out.println(previousNode + " =prev, next= " + nextNode);
        }
        //update nodes
        try{
        if(previousNode == NodeParameters.id){
            NodeParameters.nextID = nextNode;
            updatePreviousIdOfNextNode(previousNode, nextNode);
        } else if (nextNode == NodeParameters.id) {
```

I implemented the sequence that occurs at failure. First, it sends a REST request to the naming server. (The one I just talked about above). Then, the naming server answers with then next and previous or nothing as mentioned before. In that case the node does nothing and lets the other node handle the failure.

When it receives the next and previous node of the failed one. It first checks whether one of these is the this node itself. In that case it doesn't have to send an REST request. Then, it sends a REST request to the other node telling it to change its previous or next node. In the end, I did most of the work on failure. However, I must say Vital was doing its best, he just didn't seem to be very skilled at programming.

With the knowledge of the failure I also fixed the shutdown which wasn't finished in time. Since the shutdown is only part of the failure without the extra overhead it was really easy.

Besides helping on the failure and fixing the shutdown, I also served as extra pair of eyes for the work Louis was doing. In this way we were able to debug his code really quickly.

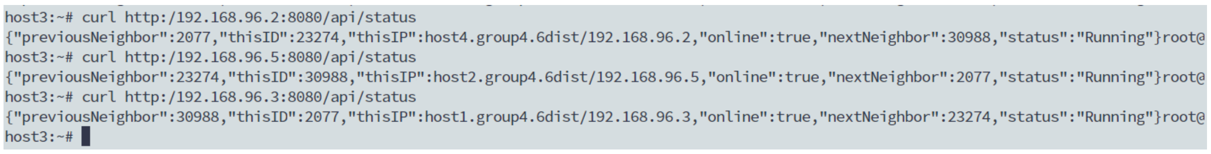


```java
// Update next node id for
Asif Wasefi
@PutMapping(path = "/updateNext")
public static  String updateNextNodeId(@RequestParam Integer hostId) {
    if(Objects.nonNull(hostId))
    {
        NodeParameters.getInstance().setNextID(hostId);
        return "successfully added next id as: "+hostId;
    }
    else return "could not add a null hostId";
}

// Update previous node id for
Asif Wasefi
@PutMapping(path = "/updatePrevious")
public static  String updatePreviousNodeId(@RequestParam Integer hostId) {
    if(Objects.nonNull(hostId))
    {
        NodeParameters.getInstance().setPreviousID(hostId);
        return "successfully added previous id as:  "+hostId;
    }
    else return "could not add a null hostId";
}
```

REST request for updating the previous and next node.

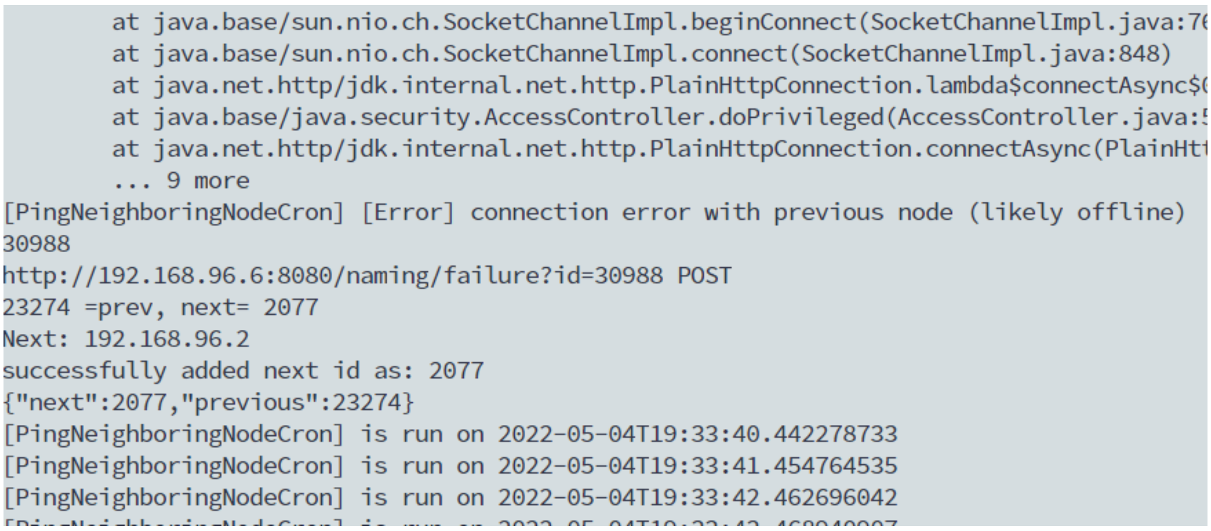
Failure tests

We start from the scenario with 3 nodes running We stop one using CTRL+C



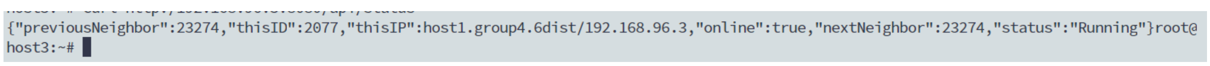
```
host3:~# curl http:/192.168.96.2:8080/api/status
{"previousNeighbor":2077,"thisID":23274,"thisIP":host4.group4.6dist/192.168.96.2,"online":true,"nextNeighbor":30988,"status":"Running"}root@
host3:~# curl http:/192.168.96.5:8080/api/status
{"previousNeighbor":23274,"thisID":30988,"thisIP":host2.group4.6dist/192.168.96.5,"online":true,"nextNeighbor":2077,"status":"Running"}root@
host3:~# curl http:/192.168.96.3:8080/api/status
{"previousNeighbor":30988,"thisID":2077,"thisIP":host1.group4.6dist/192.168.96.3,"online":true,"nextNeighbor":23274,"status":"Running"}root@
host3:~# █
```

How the nodes are configured before shutting down one



```
        at java.base/sun.nio.ch.SocketChannelImpl.beginConnect(SocketChannelImpl.java:76
        at java.base/sun.nio.ch.SocketChannelImpl.connect(SocketChannelImpl.java:848)
        at java.net.http/jdk.internal.net.http.PlainHttpConnection.lambda$connectAsync$0
        at java.base/java.security.AccessController.doPrivileged(AccessController.java:5
        at java.net.http/jdk.internal.net.http.PlainHttpConnection.connectAsync(PlainHtt
        ... 9 more
[PingNeighboringNodeCron] [Error] connection error with previous node (likely offline)
30988
http://192.168.96.6:8080/naming/failure?id=30988 POST
23274 =prev, next= 2077
Next: 192.168.96.2
successfully added next id as: 2077
{"next":2077,"previous":23274}
[PingNeighboringNodeCron] is run on 2022-05-04T19:33:40.442278733
[PingNeighboringNodeCron] is run on 2022-05-04T19:33:41.454764535
[PingNeighboringNodeCron] is run on 2022-05-04T19:33:42.462696042
```

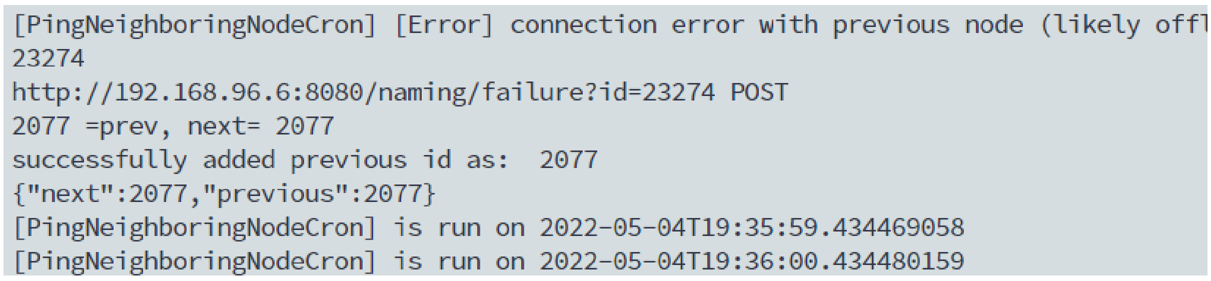
The other nodes notice this and they receive the neighbours from the naming service. Then they send the correct responses to the nodes.



```
{"previousNeighbor":23274,"thisID":2077,"thisIP":host1.group4.6dist/192.168.96.3,"online":true,"nextNeighbor":23274,"status":"Running"}root@
host3:~#
```

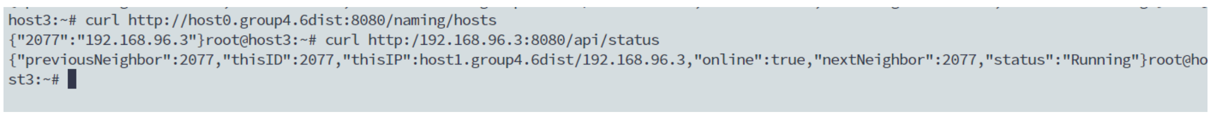
The nodes are updated/configured correctly again.

Say we stop another node:



```
[PingNeighboringNodeCron] [Error] connection error with previous node (likely offl
23274
http://192.168.96.6:8080/naming/failure?id=23274 POST
2077 =prev, next= 2077
successfully added previous id as:  2077
{"next":2077,"previous":2077}
[PingNeighboringNodeCron] is run on 2022-05-04T19:35:59.434469058
[PingNeighboringNodeCron] is run on 2022-05-04T19:36:00.434480159
```

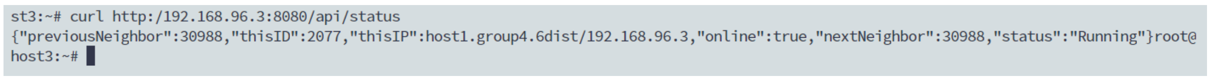
Again it is noticed by the one node still alive.



```
host3:~# curl http://host0.group4.6dist:8080/naming/hosts
{"2077":"192.168.96.3"}root@host3:~# curl http:/192.168.96.3:8080/api/status
{"previousNeighbor":2077,"thisID":2077,"thisIP":host1.group4.6dist/192.168.96.3,"online":true,"nextNeighbor":2077,"status":"Running"}root@ho
st3:~#
```

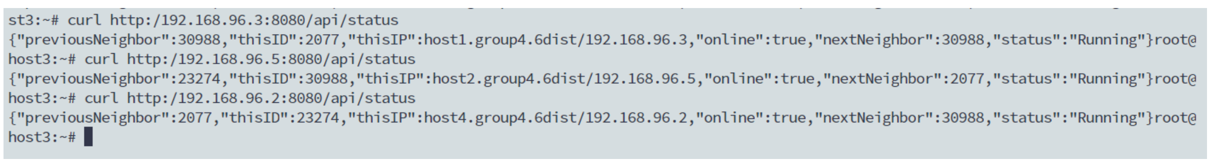
The node is configured correctly and as can be seen, the previous nodes are removed from the list.

Now if we start these nodes again



```
st3:~# curl http:/192.168.96.3:8080/api/status
{"previousNeighbor":30988,"thisID":2077,"thisIP":host1.group4.6dist/192.168.96.3,"online":true,"nextNeighbor":30988,"status":"Running"}root@
host3:~#
```

The settings get updated again. When we add our third node back. Everything is just like it was before



```
st3:~# curl http:/192.168.96.3:8080/api/status
{"previousNeighbor":30988,"thisID":2077,"thisIP":host1.group4.6dist/192.168.96.3,"online":true,"nextNeighbor":30988,"status":"Running"}root@
host3:~# curl http:/192.168.96.5:8080/api/status
{"previousNeighbor":23274,"thisID":30988,"thisIP":host2.group4.6dist/192.168.96.5,"online":true,"nextNeighbor":2077,"status":"Running"}root@
host3:~# curl http:/192.168.96.2:8080/api/status
{"previousNeighbor":2077,"thisID":23274,"thisIP":host4.group4.6dist/192.168.96.2,"online":true,"nextNeighbor":30988,"status":"Running"}root@
host3:~#
```