# Distributed Computing Project

Grupo 3, Turma 6:

- Joana Mesquita - up201907878
- Diogo Pereira - up201906422
- Miguel Freitas - up201906159

# Membership Service

## JOIN/LEAVE MULTICAST:

As required, upon joining/leaving the cluster, a node will multicast a message of this type:

```
> JOIN REQUEST FROM STORE on address, port,  membership counter, accepting TCP
connections on : 127.0.0.2 3002 0 54982
```

The last argument is the port where it will be accepting the membership initialization info that is described in the next section.

The above message shall be multicast 3 times; after this, the TCP connection listening on the last argument shall be terminated. This behavior is implemented in TCPGetMembershipInfo.java .

## CLUSTER MEMBERSHIP INITIALIZATION:

```
> received TCP info: START TCP MEMBERSHIP INFO TRANSFER FROM IP_ADRESS, PORT :
127.0.0.1 3001
> received TCP info: 127.0.0.1 3001 2022-06-03,19:08:38 127.0.0.1:3001 0
> received TCP info: 127.0.0.1 3001 2022-06-03,19:08:40 127.0.0.2:3002 0
> received TCP info: ENDED TCP MEMBERSHIP INFO TRANSFER FROM : 127.0.0.1 3001
> Finished receiving the 32 most recent log entries
> received TCP info: START TCP ACTIVE NODES INFO TRANSFER FROM IP_ADRESS, PORT :
127.0.0.1 3001
> received TCP info: ACTIVE NODE INFO FROM : 127.0.0.1 3001
2b6ff2667361c5c866037c410876b78d1a3acd43544b98108c2653be2f8f6be3 127.0.0.1:3001
> received TCP info: ACTIVE NODE INFO FROM : 127.0.0.1 3001
3edc79880660c36504a40f03ad4eb1a97c7a6cad8974c482d226b3658b69412d 127.0.0.2:3002
> received TCP info: ENDED TCP TRANSFER FROM : 127.0.0.1 3001
> ended tcp active node membership information FROM: 127.0.0.1:3001
```

The above showcases the sending of the latest log entries and active node information from node 127.0.0.1:3001 to the joining node which has the id of 127.0.0.2:3002 .

## Relevant data structures:

We made use of java's TreeMap in order to guarantee that we have our node's hashIds permanently ordered.

```
 private TreeMap<String, String> activeNodesCluster = new TreeMap<>(new
SortbyHex());
```

## Log file:

The log file keeps only the most recent event for each node.

## Stale information prevention:

In order to prevent the propagation of stale information, we choose from our active nodes in the cluster one to be a LEADER. This LEADER is initially the first node to join the cluster (he knows that he is the first because he multicasts his JOIN message 3 times). After this, every 10 seconds, this node shall multicast various LEADERSHIP INFO messages of type:

```
> LEADER MEMBERSHIP INFO FROM IP ADDRESS, PORT : 127.0.0.1 3001 2022-06-03,21:58:11
127.0.0.1:3001 0
> LEADER MEMBERSHIP INFO FROM IP ADDRESS, PORT : 127.0.0.1 3001 2022-06-03,21:58:20
127.0.0.2:3002 0
```

LEADER sends at maximum 32 LEADER MEMBERSHIP messages (with the most recent 32 membership events in its log).

Cluster nodes, upon receiving these LEADER MEMBERSHIP messages, may notice that their entries have more updated membership information than the messages they are receiving from LEADER. In this case, a node will send a LEADER CHANGE message of type:

```
> LEADER CHANGE MESSAGE, NEW LEADER IS STORE ON IP ADDRESS, PORT : 127.0.0.1 3001
```

Upon receiving this message, every cluster node updates the current cluster leader.

# Storage Service

## Put Request Message:

```
PUT REQUEST FROM : 127.0.0.1 3001
4fe157558bb127fbaf5b4dd0d4719d67520c753bfaff83c16ada67dd8d1cab2b
REFUSE_IF_NOT_RESPONSIBLE
```

A node that currently belongs to the cluster, upon receiving the above message will, depending on if the hashValue is closest to his hashed id, do one of two things:
CASE 1: the hashValue is closest to his hashed Id, out every other hashedId in the cluster currently:

- the node stores this key-value pair in its filesystem

CASE 2: the hashValue is closest to another node's hashed Id:

- the node re-sends the above PUT REQUEST to the responsible node.

This behavior is executed by PutRunnable.java.

## Get and Delete Request Message:

```
GET REQUEST FROM : 127.0.0.1 3001
4fe157558bb127fbaf5b4dd0d4719d67520c753bfaff83c16ada67dd8d1cab2b
```

A node belonging to the cluster, upon getting the above message shall check his own fileSystem for the existence of the file. If he finds it, then return its contents to the Client.
This is the first check, seeing as replication is implemented, so while this node may not be the closest node to the hashValue, it may still hold a copy of the value.

If the node does not find the file within its file system, it will check if he is the responsible for this file (checks if he has the closest Id to the hash Id). If he is the responsible and the file is not here, that means it does not exist. If he is not the responsible, then we redirect this GET message to the responsible.

## Join Event:

After receiving the multicast JOIN message, every node shall check if he is the successor of the now joining node. If he is, the node shall go through his key-value pairs and send those keys which are now the responsibility of this new node to the JOINING node.

```
//check if we are the joining node's sucessor
```

```
if(this.membershipInfo.getOwnSucessor().equals(TCPmembershipAddress +
":" + joiningNodePort)){
          System.out.println("WE ARE THE JOINING NODE'S NEW SUCESSOR,
SENDING THEM THEIR RESPECTIVE KEYS");
          TCPKeyTransfer keySender = new
TCPKeyTransfer(membershipInfo);
          new Thread(keySender).start();
        }else{
          System.out.println("JOINING NODE IS NOT OUR SUCESSOR");
        }
```

### Leave Event:

Before leaving the cluster, the LEAVING node shall transfer all of his key-value pairs to his successor.

```
 membershipInfo.incrementMembershipCounter();
 this.threadPool.execute( new LeaveRunnable(clientSocket,
                    clientString));

this.threadPool.execute(new TCPKeyTransfer(membershipInfo, true));
                //if the cluster leader is leaving, send a change
cluster leader message to the second most updated activeNode
membershipInfo.onLeaderLeave();
membershipInfo.deleteAllActiveNodes();
multicastClusterListener.stop();
```

TCPKeyTransfer.java handles the protocol for transferring key-value pairs upon a membership event. It sends a PUT request with the flag “DO_NOT_REFUSE_IF_NOT_RESPONSIBLE” to the successor.

## Replication

We implemented a crude version of replication in our project.

After a PUT request on a responsible node, it will try and send that same PUT request to two other nodes in the cluster (if there are that many), but this time with the flag “DO_NOT_REFUSE_IF_NOT_RESPONSIBLE” which tells the receiving node to NOT check if he is the responsible node for this information - he shall store it within his file system regardless.

```
//REPLICATION - ensure we propagate this FILE to another 2 stores
```

```
 List<String> replicateNodeIds = membershipInfo.getTwoActiveNodeIds();
 for(String nodeId : replicateNodeIds){
           String[] partsId = nodeId.split(":");
           String nodeAddress = partsId[0];
           int nodePort = Integer.parseInt(partsId[1]);
           redirectRequest(fileContents, nodeAddress, nodePort,
"DO_NOT_REFUSE_IF_NOT_RESPONSIBLE");
               }
```

Upon receiving a successful DELETE request (this means that the file to delete did indeed exist), a node shall multicast that request to all cluster nodes, so that they may also delete it should they have that specific file within their file systems.

# Thread Pools

We used thread pools in our project to address two different problems:
To provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and to provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. We also took the teacher's advice to use a number of threads equal to the number of cores in the machine.

```
 protected ExecutorService threadPool =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors()
;
```

# Known Bugs

Although all of the above features were implemented, at the time of the deadline for submission there is a bug in our code. It can be replicated if we start up two Stores, make them both join the cluster and then make a put request to the one who joined the cluster last - for an unknown reason, at this time, the second Store will not accept the TCP connection.