Parallel and Distributed Computing

Report - Distributed and Partitioned Key-Value Store

Catarina Gonçalves (up201906638)
Mariana Monteiro (up202003480)
Vasco Gomes (up201906617)

June, 2022

Membership Service

The goal is that every node knows every other node in the cluster, to keep this information up to date, we resort to sending messages via TCP and UDP.

In particular, every time a node joins or leaves the cluster, a JOIN/LEAVE message must be sent via IP multicast, respectively.

When a node joins the cluster, it must initialize the cluster membership. Therefore, some of the cluster members (only the nodes whose membership information is up to date) will send a MEMBERSHIP message to the new member, including its membership log, in order to update the cluster membership view and its own membership log.

The membership counter and membership log are stored in non-volatile memory, as files in the node's respective directory. The first file only contains an integer with its current counter and the latter contains the most recent 32 events of all cluster members(one line for each member).

The cluster membership view is stored as a binary search tree, represented by the class ClusterMembership. This choice of representation allows us to find successors efficiently with binary search.

ClusterMembership has useful functions that are used in the protocol like show the cluster membership view, find successor of a given key, find predecessor of a given key, insert a new member or increase a member's counter. Each member of the binary tree is a Member instance, which contains attributes like the counter, the key and the respective port.

All nodes have 3 main threads, one responsible to listen to multicast messages via UDP, another responsible to listen to unicast messages via TCP and the other is bound to the RMI in order to execute operations(the latter creates sub-threads to increase the efficiency of the operations)

In order to avoid the propagation of stale information, nodes with an empty membership log can't send MEMBERSHIP messages and the smaller the file is, the longer the thread sleeps before sending MEMBERSHIP messages. This happens both when a node receives a JOIN and in the cyclic MEMBERSHIP message and allows the nodes with more information about the cluster to send messages more frequently than the others and keep the information updated across all nodes.

Before a node tries to JOIN, we verify if its respective counter is even and if the number of tries to resend a JOIN is not exceeded. If it meets these conditions, the node creates a serverSocket to accept MEMBERSHIP messages on a new port, randomly chosen. That port is sent in the JOIN message, as explained below so that the others nodes can contact him. When the node receives the MEMBERSHIP messages, it will update its own log and membership view if any event is new or younger(higher counter). If the new node does not receive 3 MEMBERSHIP messages within 3 seconds, he tries to resend the JOIN message until it reaches 3 tries. In another thread, after he creates the serverSocket, he sends the multicast JOIN and increases its own counter.

When a node receives a JOIN message, it updates its own log and membership view, sends a MEMBERSHIP message if it has updated information as explained before and updates the files as it will be explained in the Storage Service below.

When a node tries do LEAVE, we verify if its respective counter is odd. If so, he will do some put and delete operations to keep the replication factor of all files equal to 3. At the same time, there is another thread responsible to send the multicast LEAVE message and increase its own counter.

The cyclic MEMBERSHIP messages are sent in the ReceiverUDP file with the help of Java's Timer. While the ReceiverUDP's main thread is listening and receiving multicast messages, the class has another thread running only to send MEMBERSHIP messages when it passes 5 seconds of the last received MEMBERSHIP multicast. As explained before, in this thread we have a sleep based on the content of the node's membership log, which allows nodes with more and updated information to send the message first.

Every time the main thread receives a MEMBERSHIP message, it schedules a task for 5 seconds and puts true in a variable, receivedMembership. After 5 seconds, the task that is executed is simply to put that variable false, in order to let the secondary thread stop busy waiting in a while loop and to send a MEMBERSHIP message if it is fast enough.

Implemented Messages

In this subsection, we describe the message format used in the protocol.

Membership

MEMBERSHIP <nodeid> <nodeport> <crlf> <log> <lastcrlf>

This message is sent by a node after receiving a JOIN message or when after 5 seconds without receiving a multicast MEMBERSHIP message.

```
MEMBERSHIP 224.0.0.0 4003
172.0.0.1-0-8001
172.0.0.2-0-8002
172.0.0.3-0-8003
172.0.0.4-1-8004
172.0.0.5-0-8005
```

Figure 1: Membership message.

Join

JOIN <nodeid> <nodeport> <counter> <joinport> <lastcrlf>

This message is sent by the node that wants to join to all nodes.

```
JOIN 172.0.0.5 8005 0 49954
```

Figure 2: Join message.

Leave

LEAVE <nodeid> <nodeport> <counter> <lastcrlf>

This message is sent by the node that wants to leave to all nodes, or by the node that tries to communicate with another node that is not available.

```
LEAVE 172.0.0.4 8004 1
```

Figure 3: Leave message.

RMI

The definition of the remote interface is implemented in RMIServer.java

Storage Service

In order to allow testing the several nodes on a single computer, each node uses its own folder to keep the values of the items it is responsible for, the originals and the replicates ones.

Put Operation

To implement the put operation, we first verify the responsible node for saving the received file. For this verification, we create an auxiliary function to search the successor node of the key, which is calculated through the hash calculation of the file name that we received in the TestClient.

After getting the successor node, we check if its id matches the id we received in the TestClient. On the one hand, if so, it creates the file with the name of the key and the same content in its directory. After that, a TCP message, PUT, is sent to the predecessor and successor to create replications of the file. On the other hand, if the id of the successor node does not match with the id that received the put operation from the client, a TCP message, PUTREPLICATE, is sent to the responsible node so that it performs a put operation of the file.

So, in the ReceiverTCP file, the put operation is made according to the messages that have been received.

Delete Operation

To implement the delete operation, we first verify the responsible node for saving the received file, as we explained previously.

After that, we check if the file we want to delete exists and if so, if the node that received the delete operation from the client is responsible for that file, we delete the file in its directory and send a TCP message, DELETE, to delete the files that were replicated in the successor and predecessor, as we made in operation put.

On the other hand, as explained in the put operation, if the nodes is not the responsible for the file, a TCP message, DELETEREPLICATE, is sent to the responsible node so that it can delete the file in its directory and warn other nodes that have this file, successor and predecessor nodes.

In the ReceiverTCP file, we also made the delete operation according to the messages that we received.

Get Operation

To implement the get operation, we first verify the responsible node for saving the received file, as we explained in put operation.

If the node is the responsible or has the replicate files, we read the content of the file and return this to the client that requested.

Opposite to the operations explained above, if the node doesn't have the file, we answer the client with the id that he needs to contact to get the content of the file that he wants.

Pair transfers on Membership changes

Every time a nodes enters or leaves the cluster, the files are reorganized in order to keep all files replication factor equal to 3 as soon as possible and to prevent faults.

When a node joins, the successor of this new node will send to him the files that now have the new node as their successor.

In the other hand, when a node leaves the cluster, it sends his file to his successor, as that node will be the new successor of that files.

The implementation of the replication will be explained with detail in the section below.

Implemented Messages

In this subsection, we describe the message format used.

Put

PUT <nodeid> <nodeport> <key> <crlf> <value> <lastcrlf>

This message is used to have the predecessor and successor create a file in its directory.

```
PUT 172.0.0.1 8001 a4480283519b5d4997477fca5efddb6d3ed13d15cfa7515de5cfd70721aea029
```

Figure 4: Put message without the value.

Delete

DELETE <nodeid> <nodeport> <key> <lastcrlf>

This message is used to have the predecessor and successor delete a file in its directory.

```
DELETE 172.0.0.3 8003 912cfc4a41c3f89241b426aebf67c9f4e47437c83f7549411e3a1b0678a0e612
```

Figure 5: Delete message.

Replication

When a node that stores a key-value pair goes down, the key-value pair becomes unavailable as well. To solve this problem, we implemented a replication of these pairs. More specifically, key-value pairs are replicated using a replication factor of 3, which causes each key-value pair to be stored in three different cluster nodes. In this way, we are guaranteeing greater availability for key-value pairs.

Replication implementation

To implement the replication, whenever a node is joining or leaving, there are some operations that are required to keep all files replication factors. That happens when a node "sees" another node joining and needs to send files if necessary, when a node is leaving and needs to send files to the new owners or when a node "sees" another node leaving and can be necessary to replicate some files.

In order to get more details on the exact operations that happen, we will describe below the pseudo code implemented in our project. The term "files belong to X" is used to simplify the concept that that file has X as the successor of his key, which means that X is the responsible to save the original file.

After a node receives a JOIN message:

function uponReceivingJoin
if I am successor of the new node **then**
Send to the new node all the files that belong to him or still belong to me
Delete all files that belong to the predecessor of new node (because I will no longer need to replicate that files)
else if I am predecessor of the new node **then**
Send to the new node all the files that belong to me
Delete all files that belong to the successor of the new node(same reason as above)
else if I am successor of the successor of the new node **then**
Delete all files that now belong to the new node(same reason as above)
end if
end function

Before a node sends a LEAVE message:

function beforeSendingLeave
Send to the successor of my successor all files that belong to me
end function

After a node receives a LEAVE message:

function uponReceivingLeave
if I am successor of the old node **then**
Send to predecessor of the old node all the files that belong to me (because that node needs to replicate that files now)
else if I am predecessor of the old node **then**
Send to the successor of the old node all the files that belong to me(same reason as above)
end if
end function

Implemented Messages

In this subsection, we describe the message format used.

PutReplicate

PUTREPLICATE <nodeid> <nodeport> <key> <crlf> <value> <lastcrlf>

This message is used for the responsible node to create a file in its directory and send a TCP message, PUT, to its predecessor and successor to perform the put operation and save the replication of their files.

```
PUTREPLICATE 172.0.0.1 8001 912cfc4a41c3f89241b426aebf67c9f4e47437c83f7549411e3a1b0678a0e612
```

Figure 6: Put replicate message without the value.

DeleteReplicate

DELETEREPLICATE <nodeid> <nodeport> <key> <lastcrlf>

This message is used for the responsible node to delete a file in its directory and send a TCP message, DELETE, to its predecessor and successor to perform the delete operation of the replicated files.

```
DELETEREPLICATE 172.0.0.4 8004 912cfc4a41c3f89241b426aebf67c9f4e47437c83f7549411e3a1b0678a0e612
```

Figure 7: Delete replicate message.

Others Fault-Tolerances

Beyond the fault tolerances in the membership service and in the storage service, we decided to implement another one that happens when a node is suddenly offline.

When a node tries to contact another via TCP (PUT, PUTREPLICATE, DELETE or DELETEREPLICATE) but the other node is down, for some reason, and didn't LEAVE correctly, the node sends a multicast LEAVE message in order to let the cluster know that that node is no longer available.

Following that, all files that were replicated by that node are sent to the new nodes responsible for that replication as explained before. This allows, at least, to keep all files replication factor at 3 and prevent nodes to send stale information about that node.