Distributed and Partitioned Key-Value Store

Computação Paralela e Distribuída, L.EIC

João Marinho up201905952

Tiago Silva up201906045

1 Introduction

The objective of this project was to develop a distributed key-value persistent store for a large cluster.

A key-value store is a simple storage system that stores arbitrary data objects, the **values**, each of which is accessed by means of a **key**, very much like in a hash table.

To ensure persistence, the data items and their keys are stored in persistent storage (HDD/SSD), rather than in volatile storage (RAM).

By **distributed**, is meant that the data items in the key-value store are **partitioned** among different cluster nodes.

The project design was loosely based on Amazon's Dynamo, in that it uses **consistent-hashing** to partition the key-value pairs among the different nodes.

2 Membership Service

2.1 Message Format

All messages are char-based, with a JSON-like format, in which the 'header' attribute distinguishes the message type and content.

In this regard in terms of membership there are the following messages:

- **JOIN**: Symbolises the arrival of a node into the cluster, contains the **id** of the node, the **membership counter** and the **port number** to which it is ready to receive the initial *MEMBERSHIP* messages.
- **LEAVE**: Represents the removal of a node from the cluster, like the *JOIN* message contains the node's **id** and **membership counter**.
- **MEMBERSHIP**: Is the protocol's most recurring message, that includes a node's view of the membership, i.e. the 32 most recent **membership events** in its log, as well as the **id** of the sender.
- **ELECTION**: Denotes a node's application to the coordinator role, informing the remaining nodes of its **priority** value and **id**.
- **COORDINATOR**: Possesses the same content as the *ELECTION* message, although it tells the cluster members the acceptance of the coordinator's role.

2.2 Membership Protocol

The lifetime of a node within the cluster starts upon a *JOIN* request by the client, this request triggers a set of actions defined in the membership protocol:

1. If the client inquiry is valid (even value counter), the membership counter of the node is updated accordingly.
2. It is then necessary to initialise the cluster membership. For this, the node accepts **unicast** connections (**via TCP**) that allow it to receive *MEMBERSHIP* messages from the other members of the cluster.
3. A *JOIN* message is sent **via multicast** which, as seen previously, contains the port associated to the created socket, so that the members with greater knowledge [2.3] can make a connection.
4. A total of 3 connections must be established, where after 3 seconds a retransmission of the message already sent is made up to a maximum of 3 retransmissions, in the hope of receiving the desired number of messages. In case of failure to connect with 3 other nodes, the current node carries on with the process.

5. From this point on, it will subscribe to the multicast channel, becoming ready for any request by the client or the rest of the cluster members (**via multicast** or **unicast**).

We can verify that many messages can be sent within the cluster, this high number of transmissions can lead to failures and so, there is a node, called the *Coordinator*, which has the highest priority among all nodes in the cluster. This node is therefore responsible for, every second, sending a *MEMBERSHIP* message so that nodes with a higher latency can monitor the new entries and exits of the cluster (more on the *Coordinator* topic in the section bellow).

Regarding the nodes that receive the respective JOIN message, the following procedures have been established:

1. Update of the cluster membership view, adding the respective event to the **membership log** and also to the respective logs file allowing the information to be stored persistently.

2. Attempt to establish a connection with the node intending to join, according to the priority value. For this matter a heuristic was therefore created in which only nodes with a maximum difference of 3 units with the Coordinator's priority can send the *MEMBERSHIP* message, in order to favour the highest priority nodes, a time delay is imposed in such a way that the greater the difference the greater the interval.

Not only do we need to define the protocol actions for entering the cluster, but also those for exiting the same. Thus, the node's exit of the cluster process starts with a *LEAVE* command from the client:

1. Just like joining the cluster, the given membership counter, in the client request, must be valid (odd), leading to its update.

2. The node informs the rest of the cluster that it is about to leave with a *LEAVE* multicast message.

3. Finally it stops receiving/sending requests and closes its connections both multicast and **unicast**.

The remaining members of the cluster upon receiving the message update their **membership log** and respective file, removing the node from the active nodes.

2.3 Election Algorithm

As previously mentioned, a node of the cluster, *Coordinator*, is responsible for periodic *MEMBERSHIP* messages. The way the node is chosen, goes through a two-stage process which is based on the Bully Election Algorithm.

The first phase, election phase, starts as soon as one of the cluster nodes stops receiving one of the periodic messages for a time interval of 2 seconds. Thereafter, this node will inform the others that it wishes to be the *Coordinator* by means of an *ELECTION* message. The parse of this type of message leads to two possible outcomes, either the node has higher priority and maintains its state, or the node has lower priority and accepts that it will not be the *Coordinator*.

Once the time limit is reached, without receiving any *ELECTION* message, phase two begins, the Coordinator's phase. Nodes that have only received the previous messages with a priority value lower than their own (low probability of being more than one node, however possible due to failures in sending), shall forward a *COORDINATOR* message, representing their acceptance for the position. If by chance any of the nodes has a higher priority than the node that wants to be *Coordinator*, it will reply to it, stating that it is the *Coordinator*, until there are no more replies. Once the Coordinator's phase is over, and the node chosen, *MEMBERSHIP* messages are automatically resent.

2.4 RMI

RMI is used in this protocol whenever a client wants to send a *JOIN* or a *LEAVE* to a given node.

So, with the help of the documentation we implement an **RMI interface**, which allows us to perform the desired operations.

```java
package interfaces;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RMI extends Remote {
    void join() throws RemoteException;

    void leave() throws RemoteException;
}
```

Figure 1: RMI interface

2.5 Missing Details

Did not implemented the referred feature in the fault-tolerance topic of the project's proposal:

- If a node is down for a long time and it misses many membership events, the periodic *MEMBERSHIP* messages may not be enough for the node to learn the current membership of the cluster.

3 Storage service

3.1 Message Format

Likewise for the **membership protocol**, all messages are char-based, with a JSON-like format, in which the 'header' attribute distinguishes the message type and content.

In addition to the already mentioned *JOIN* and *LEAVE* messages, the following were also introduced:

- **PUT**: Representative of the operation of storing a file whose name is equal to the **key** attribute and **value** content.
- **GET**: Request to retrieve the content of a file specified in **fileName**.
- **DELETE**: Message with the same structure as the previous one, but with the objective of deleting the key-value pair.
- **FILETRANSFER**: Aims to transfer a file (key-value pair) onto another node, keeps its **key** and **value**.

3.2 Implementation

Implementation wise the cluster nodes are logically arranged in a **circular fashion**, with increasing value in a clockwise direction. The position of each one is defined by the hash function used by the client to compute the keys, using for this case the **id** of each member.

Every node holds an ordered list of the **currently active nodes** within the cluster, which allows them to know who is responsible for a particular key-value pair in an efficient way (binary search) and proceed with the appropriate request redirects.

Hence the three **fundamental requests**, sent **via TCP**, are treated as follows:

Put Whenever a request of this type is made, there is a **verification** of who is responsible for the pair sent.

The most straightforward scenario occurs when the node is not responsible, and so the message will be **forwarded** to the node which the receiving node assumes to be the actual responsible for keeping the key-value pair.

The second scenario happens when the receiver itself is the responsible, proceeding to the creation of the respective file inside its **storage** folder. It is also here that the appropriate replication is dealt with, which will be reviewed in the following topic.

Get In the same way as the previous request, the owner of the file is compared with the receiver. However, as an optimization the existence of the pair in the **replication** folder of the receiver node was included, so if it possesses the **demanded content**, it will send the following **via TCP**.

On the other hand, if the file is marked with a **tombstone**, a reply will be sent informing that the pair is not on the network (deleted).

Eventually, after all these verifications, if the receiver is not the holder, the client is warned that it must **redirect** the message to the **responsible** node, according to the information included in the message's content.

Finally, if for any reason the **owner** of the file does **not possess** it, the client is informed that the pair does not exist.

All these responses are **distinguished** by the respective **header** present in the message, which may have the value of *CONTENT* in case of success, *REDIRECT* along with the responsible node's address, *FILENOTFOUND* and *DELETED*.

Delete Requests of such kind cause a key-value pair to be **removed** from the cluster.

The message is **redirected**, **via TCP**, until it reaches the node responsible for the file, whereupon it will be **marked with a tombstone**, indicating its removal.

The owner will inform the other nodes of the event so that the **replicas** are also deleted, through a multicast message of type *REPLICATION_DELETE*.

The storage service also needs to take into account possible **membership changes**, i.e. the transfer of keys on the occasion of a join or leave event.

On that account, following a *join* event a node will check if the incoming node is behind it in the logical circle, implying a key-value pair transfer. If so, the files stored in the current node's **storage** folder will be **traversed** in an orderly fashion and **transferred** using *FILETRANSFER* messages, these same files will be passed to the **replication** folder, since the node is not the responsible for the given pair.

Meanwhile if a node wants to leave the cluster it proceeds to **send the keys** to the node that will succeed it, i.e., the node directly in front of him. Once again, the transfers are done through *FILETRANSFER* messages **via TCP**.

3.3 Fault-tolerance

A fault-tolerance provision was taken in the scenario where a particular node to which a *GET* request is made, is not up to date with the cluster information. Therefore, when processing the request, it may return an incorrect response. Trying to reduce the probability of an occurrence like this, if, despite being outdated, this node does not consider that it is responsible for the requested file, it forwards the client to another node that may already have the updated information. This new node can forward the client to the responsible, or reply with the file if it turns out to be the key-value pair owner.

4 Replication

Regarding replication, in order to increase availability, our protocol strives to replicate each key-value pair with a **factor of 3**, i.e. each key-value pair should be stored in 3 different cluster nodes.

4.1 Message Format

There are two types of replication messages, once again, since they are protocol messages, they differ from the others by the **header** field:

- **REPLICATION**: Consisting of the fields related to the file to be replicated, **key** (the hash of the file name) and **value** (the file content).
- **REPLICATION_DELETE**: Intended to spread the knowledge that a file with the given **key** has been deleted.

4.2 Message Parsing

When a node receives a *REPLICATION* message, its parsing is simple: the received file will be added to the **replication** directory that each node possesses. It's in this directory where all the replication files will be stored.

4.3 Implementation

This feature is supported in the different components of the protocol:

PUT Whenever a node receives a new file through a *PUT* message, in the course of its processing, an attempt to **replicate** the message to another two nodes will occur. Considering the cluster organization by the hash's order, the two chosen nodes will be the next and previous, according to the position of the first one. Thus, this node will try to establish a connection between them, ensuring a **replication factor of 3**.

However, there may exist cases where the number of cluster nodes known by the node is less than two (it can be alone in the cluster or with only another node). In these cases, the message will be added to a set of messages that are waiting to be replicated. This set will be saved in the **messagesToRetransmit** structure in the node **clusterInfo** field, which will keep track of the number of times each message needs to be replicated, proceeding to the same whenever possible, i.e. whenever someone **joins** the cluster.

JOIN When a node receives a *JOIN* message, **via multicast**, it will verify if there is any message that needs to be replicated. Hence, to avoid any fraud, i.e., prevent the node that is joining the cluster from receiving the message that needs to be replicated, tracking is made of to whom the message has already been sent. And so, at each received *JOIN* message, each of the *REPLICATION* messages that have not been already replicated **twice** will be sent, **via TCP**, remaining in the data structure if the new cluster node belongs to the list of already **replicated nodes**.

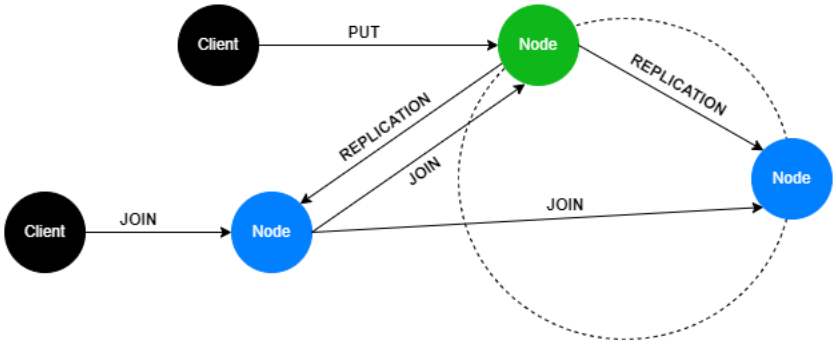

Figure 2: Example of a possible replication case

4.4 Implications on Membership and Storage Services

Therefore, due to replication, the parsing of a *GET* or *DELETE* message will need to **adopt other behaviours**.

When a node receives a *GET* message, if it is not responsible for storing the requested file, it will also check in the **replication folder**, if it does not have that file replicated, avoiding the need for the client to establish a new connection with another node (with a *REDIRECT* response).

As for *DELETE*, at the parsing time, it is necessary for the node to verify that it has the file in its **replication folder**, and if it does, it must mark it as deleted to keep the cluster information up to date.

5 Fault-tolerance

A possible case that could lead to the protocol failure would be if, for some reason, the **coordinator** node fails and stops sending periodic *MEMBERSHIP* messages. So, in order to solve this problem, as already explained, when a node spends a certain period of time (2 seconds) without receiving these messages, it automatically starts the election period, where, at the end of it, one of the nodes is again elected coordinator.

6 Concurrency

6.1 Thread-pools

So as to **reduce** the number of threads used, and **mitigate** some performance issues in our project, **3 thread-pools** were introduced, each with its own purpose. If used correctly, thread-pools, help not only to **separate** the execution of tasks and the creation/management of threads, but also **increase** the overall performance of the processes.

The idea of creating more than one thread-pool may seem **pointless**, however its purpose comes from the idea that different activities (tasks) **keep up with each other's speed**, avoiding thread starvation, the system becomes **more stable** as the load changes.

Message Parser Pool Pool of the type *newCachedThreadPool*, this type of pools generally **improve the performance** of programs that execute asynchronous tasks of short duration, **reusing threads** already used and that have finished their execution, being that threads that stay a certain time without running are removed from the cache.

Whenever a **message is received** in order not to take time with its parsing, i.e., to evaluate the message content and **execute actions accordingly**, a task is added to the pool to be executed later.

Therefore, this thread-pool is intended to parse all messages received by the node, both **via unicast** and **multicast**.

Message Sender Pool Likewise the previous pool, to send messages from one node to another, we also use a pool of the type *newCachedThreadPool*.

Whenever we want to **send a message**, we create a task depending on the type of message we want to send: **unicast** vs **multicast**. Thus, the task is added to the pool and the message will be sent when a thread is available.

The necessity to create one more thread pool of the same type as the previous one arose due to the vast amount of communications that the protocol may require. In this way, and to not disrupt its performance, a thread pool was created responsible for sending all types of messages.

Scheduler Thread Pool Unlike the other thread-pools, a pool of the type *newScheduledThreadPool* is used so that commands can be scheduled to run after a given delay, or to execute periodically.

Since this type of tasks are not so usual, but mandatory to happen during the process of a node, this pool has a predefined size of 1 thread.

Here tasks such as **checking for missing *MEMBERSHIP* messages** during a period of time triggering **timeouts during the election algorithm**, and finally the **dispatch of such messages** by the *Coordinator*, are all tasks assigned to the pool.

6.2 Asynchronous I/O

I/O operations relate to the exchange of information between two systems, with **Input** symbolising the signals received and **Output** symbolising the signals sent.

A simple way to act on these types of operations would be to wait or block (**synchronous I/O**) for an operation to finish. If the program does many operations of this type, however, the **processor may take up most of its time** waiting for them to complete.

Alternatively it is possible to establish a communication and **later perform processing** without having to wait for the end of the operation, this approach is called **asynchronous I/O**, and is implemented in the project to establish communications **via TCP**.

With the help of the Java package, nio.channels, we were able to **abstract** from the difficulties of implementing **asynchronous operations** by using the **AsynchronousServerSocketChannel** and **AsynchronousSocketChannel** to establish communications, **Future**'s that allow us to poll or wait for the result of an operation and better yet the use of **CompletionHandler's**, i.e. callbacks invoked upon success or failure of the operation.

It is possible to obtain a number of **operations called outstanding**, much higher than the number of threads in the **asynchronous channel pool**, while in the case of using synchronous channels it would be necessary a **thread for each operation**. This demonstrates the power of such channels over the **consumption/use** of threads.

At the project level, they are used to communicate **via TCP**, in the **unicast package**, inside the communication folder, where the two existing **handlers**, for the following situations, are also defined:

- **Initial Handler**: Callback used to establish the 3 initial connections when a node is joining the cluster.

- **Receiver Handler**: General tcp message receiver, that accepts new connections after one has been established.