CPD 2nd Project - Distributed and Partitioned Key-Value Store

Afonso Duarte de Carvalho Monteiro - up201907284
Miguel Azevedo Lopes - up201704590
Vasco Marinho Rodrigues Gomes Alves - up201808031

June 3rd, 2022

Contents

1 Problem Explanation

This report aims to present and explain how the distributed key-value persistent store for a large cluster was developed and implemented. The objective was to store arbitrary data objects, **values**, which are accessed by a **key**. The data items and their keys must be stored in persistent storage and they must be partitioned among different cluster nodes.

2 Membership Service

This service is initialized as a thread, once the Node receives a join request from the TestClient.

It creates a thread that starts accepting unicast (TCP) messages. It then creates another thread to send multicast JOIN messages, which it sends repeatedly until one of the two following conditions is met: it receives 3 JOIN_RESPONSE messages or it has sent 3 multicast messages already. After that, the thread created to receive TCP messages is closed and it uses the JOIN_RESPONSE messages to update its logs. It then creates a new thread which starts listening to multicast messages, and another that sends a periodic message every second. The periodic message is only sent if the node is considered "up to date", in other words, it believes to have reliable information. A node determines this by comparing its logs to the logs it receives from other nodes. If for some reason it finds that one of the membership counts in the logs received is superior to the one it has, it updates its logs and marks itself as "out of date", until the logs it receives from other nodes match the ones in its logs. The constant sharing of information through the exchange of periodic messages assures that the logs are kept updated and grants this service an elevated fault-tolerance.

Although it wasn't mentioned in the project requirements, we implemented a safeguard to check for "dead nodes". We consider a dead node to be any node that hasn't communicated with the other nodes in at least 20 seconds. This verification adds to the fault-tolerance of this protocol, even though it is redundant since we also implemented replication.

For the membership messages we implemented a class **MembershipMessage** which has a **type** that takes one of the values {**JOIN, JOIN_RESPONSE, PERIODIC, LEAVE**}, a copy of the **membership log**, which saves the current status of the storage membership, and a **membership view**, which contains information about the node. This message is used to signal the events that were previously mentioned (join, join responses, periodic update messages and leave). To make sure the message is human readable, we override the toString method and implement a constructor that uses a string to reconstruct the message on the receiver end.

2.1 RMI

Our implementation includes RMI which facilitates the communication with the nodes. The binding of the address to the actual node is done in the *KeyValueStoreServer.java* file. To simplify the communication we established that the name of the object is the hashed value of its IP address.

3 Storage Service

To ensure the storage service is implemented as a distributed partitioned hash table in which each cluster node stores the key-value pairs in a bucket and that they are laid out on a circumference we used a Java **TreeSet**, which stores the node hashes in ascending order. This data structure is particularly useful because of its **higher** method, that returns the first element that is higher than the one passed as an argument, which makes it easy to find the successor nodes. It is also useful to find the node responsible for a key, since the argument of the higher method doesn't need to be in the TreeSet.

If a node that receives an operation (put/get/delete) isn't the one responsible for executing it, we calculate the responsible node and then send a unicast (TCP) message to it. Once the responsible node receives the message, it performs the respective operation, storing key-value pair in its storage if it is a put message, retrieving the value bound to a key if it is a get message or deleting key-value pair in case of a delete message.

For the storage messages we used a class **StorageMessage** which has an attribute **type** that takes one of the values {**GET, PUT, PUT_REPLICATE, DELETE, DELETE_REPLICATE**} and an attribute **contents**, which is the content of the message (key or value). The **PUT_REPLICATE** and **DELETE_REPLICATE** were created so we didn't need to check if it was the right node receiving the message. They are sent between nodes so there is no room for error. To make sure the message is human readable, we override the toString method and compose it by appending the message **type**, then two empty lines and finally the **content**. It is sent as a string and reconstructed into a class using a constructor that takes a string as an argument.

When a node joins or leaves the cluster we also deal with the pair transfer by sending unicast storage messages to the nodes that will be affected.

4 Replication

When a key-value put/delete operation is performed on the storage it is first performed on the responsible node, which then sends a unicast message to its two successor nodes for them to perform the replicate operation. When a get operation is performed, we find the node responsible for the key and call the get method on that node using RMI. If it doesn't have the key, we search on its two successor nodes.

5 Fault-tolerance

Our protocol covers all the possibilities mentioned in the project requirements, including the ones mentioned in the fault-tolerance section.

If a node has an out of date log and sends the request to the wrong node, the receiving node will always check its own log first before acting upon that request, which ensures that the request always reaches the correct node.

As mentioned previously we also implemented a system which detects the presence of a dead node in the cluster. We consider a dead node any node that hasn't communicated with other nodes.

It should also be noted that we tested our program thoroughly by using a test library (JUnit) to make sure that every aspect of our implementation was solid. However, when running these tests **the**

user should run each test individually and one at a time (as opposed to all in a row as some IDEs allow) because, since we use loopback addresses and the same ports for different tests, we found that sometimes the TCP ports took a few seconds to close after each test was ran, which naturally interferes with the correct functioning of the tests.

6 Concurrency

We found that using thread pools wouldn't really add any value to our implementation since we were using a small number of threads.

7 Conclusion

The most important objectives of this assignment were achieved as we understood the advantages of the implementation of a distributed system in certain scenarios. We had to face and solve some challenges since it was the first time we had to work on a system like this.

We also consider that the realization of this project contributed to our development as future engineers because we had to learn new tools and use different strategies to solve our problems.