Parallel and Distributed Computation

Distributed and Partitioned Key-Value Store

João Baltazar[1], Nuno Costa[1], and Pedro Correia[1]

[1]Faculty of Engineering of the University of Porto

June 3, 2022

Abstract

The goal of the project is to create a distributed key-value persistent store for a large cluster.

This goal was achieved - the distributed system meets the specifications. Furthermore, the hands-on approach and further reflection on what such a system entails allowed for a deeper understanding of the main problems of distributed systems, as well as ways of tackling them.

1 Introduction

This assignment has two main components: a membership service and a storage service. The latter is supported with a replication mechanism to make the system more robust to unexpected downtime. In the same line of thinking, the system also has fault tolerance mechanisms and supports concurrency for better performance and responsiveness. All topics mentioned will be explored with more detail below.

2 Membership Service

For a cluster-based distributed system to function correctly, it's important to establish a way of ensuring that the cluster information is up to date on all active nodes. For that, a membership protocol is established.

2.1 Membership Protocol

A node can either JOIN or LEAVE the cluster. This message is sent via multicast to all nodes currently connected to the group. Whenever a node joins the cluster, it expects to receive 3 MEMBERSHIP messages, from 3 different nodes, to set up its cluster information. If it does not receive them within a given time-frame (precisely 3 timeouts, 1 second each), it considers itself as part of the cluster.

In order not to flood the joining node, a node selection must be done. Given a parameter MEMBERSHIP_THRESHOLD_SEND, which represents the number of nodes that should send the message, a probability function was defined. On each node, this probability function, which is defined below, is executed and determines if the node should or should not send the message. It has an expected value of MEMBERSHIP_THRESHOLD_SEND, which attempts to assure, with high probability, that approximately MEMBERSHIP_THRESHOLD_SEND nodes send the message.

$$P(S) = MIN(1, T/(S-1))$$

where

S is the size of the cluster

T is MEMBERSHIP_THRESHOLD_SEND

To assure that not all messages reach the joining node at the same time, they are randomly spaced out, with a time delay ranging from 0 to MAX_SEND_MEMBERSHIP_VIEW_DELAY_MILLISECONDS ms.

To guarantee that the whole cluster is up to date, a MEMBERSHIP log message is sent every second by an elected node. For this particular mechanism, some approaches were evaluated.

Firstly, an election algorithm was considered, such as the Bully Algorithm - which made sense

for a complete network like this. Even though this approach would work, there were two main problems:

1. it's a heavy algorithm, requiring the exchange of a lot of messages between nodes
2. a good heuristic for what is an 'up-to-date' node would be required, and given the reduced information a node stores about itself and the others, this heuristic could end up not being admissible or leading to bad elections

Another approach considered was to elect the most recently joined node. This mechanism would make sense since a recently joined node is thought to be one of the most updated ones - since it receives 3 MEMBERSHIP messages from other nodes of the network. This, however, would not be guaranteed, the mechanism would not be as resilient as the first one considered and it would still require the exchange of messages between nodes.

In fact, exchanging messages altogether seems to be unnecessary. Given the periodicity of the MEMBERSHIP messages, it is possible to select an arbitrary node to send them and, whenever another node detects that the node echoing these messages is not up to date (its log is more recent, which can be seen if the membership counter of an entry is higher), it informs the elected node and all others via multicast to update their membership counter by sending a MEMBERSHIP log message.

As such, the chosen method to elect the node was to select the node with the lowest ID and execute a silencing mechanism - whenever a node detects that a lower ID node is echoing MEMBERSHIP messages, it stops sending them. This method is also resilient to crashes, which will be explained below.

2.2 Message Format

For proper communication between nodes, a message format was designed. This format is human readable and composed of a body and headers. The headers specify metadata of the message, while the body contains the content. The first header always contains the size of the message as well as the type of message.

There are 5 main messages used in this protocol: JOIN, LEAVE, MEMBERSHIP and REINITIALIZE. The specifics of each message are detailed below.

2.2.1 JOIN

The JOIN message has 3 headers: the ID of the node sending the message, the port on which it will communicate and its membership counter.

It also has an optional header: blacklist. This blacklist header entry defines a node which has already sent the MEMBERSHIP message, so it won't re-send it. It is added when the sending node re-sends a JOIN message - it had not received 3 MEMBERSHIP messages and timed out.

The JOIN message has no body.

2.2.2 LEAVE

The LEAVE message has 2 headers: the ID of the node sending the message and its membership counter.

The LEAVE message has no body.

2.2.3 MEMBERSHIP

The MEMBERSHIP message has 2 obligatory headers: the ID of the node sending the message the number of logs it will send.

It also has an optional header (only used upon JOIN and REINITIALIZE messages): the number of nodes in it's cluster.

The MEMBERSHIP message body is composed by the list of nodes in the cluster of the sending node (if the header is present) followed by the logs to send. The information is separated by newlines.

2.2.4 REINITIALIZE

The REINITIALIZE message has 2 headers: the ID of the node sending the message and the port on which it will communicate.

It also has the optional blacklist header.

The REINITIALIZE message has no body.

Its usage is described below.

2.3 Stale Info Avoidance

Since there are periodic updates, stale information must occur upon a node crash/unavailability.

Upon a crash, a REINITIALIZE multicast message is sent. This happens when a node detects that it's being initialized but its membership counter indicates that it already joined the cluster. As such, a crash must've occurred, and this message proceeds to execute the same procedure as the JOIN message - receiving 3 MEMBERSHIP

messages to update the node information accordingly.

Upon a possible unavailability - such as the network becoming unavailable, thus preventing the node from receiving messages - the echoing mechanism still assures that no stale information is passed on to other nodes, since an update multicast message will reach the stale node either when it receives the MEMBERSHIP log periodically or when another node detects that it has fresher information.

2.4 RMI

RMI was properly implemented to allow server-side execution of the TestClient commands JOIN and LEAVE. Its implementation is in the TestClient class.

2.5 Unresolved Edge Cases

3 Storage Service

The main purpose of this distributed system is to store key-value pairs in persistent storage. As such, a storage service was implemented to assure that it worked properly.

3.1 Storage Protocol

The storage service must allow 3 different operations: a PUT operation, to insert into the cluster a key-value pair; a GET operation, to obtain the value associated to a given key; and a DELETE operation, to remove information from the cluster.

To decide if a node is responsible or not for storing a key-value pair of the distributed hash table, consistent hashing is used. For that, the node's hashed value is used to define its position in a circular-like structure, where it is possible to easily define a node's successor and predecessor, knowing that position 0 and n, given n nodes, represent the same node.

With this node structuring, it is possible to associate a bucket of key-value pairs to each node, which facilitates changes in what key-value pairs each node stores upon membership changes, which is explained further below.

Whenever a PUT operation occurs, the node verifies if it is responsible for storing that same key-value pair. If it is, it stores the information and answers with a REDIRECT message telling the client what are the other replica nodes (replication is explained more in-depth below). If not, it sends a REDIRECT message to the client indicating all nodes responsible for storing that pair. It must be noted that the client should implement a way of traversing the nodes according to the REDIRECT message responses until all replica nodes are visited.

This same logic applies to the GET operation and DELETE operations as well, only differing in the fact that the DONE response for the GET operation has the value for the requested key in its body.

This protocol must also take into account membership changes of the cluster, since the bucket partitioning also changes. Upon receiving JOIN message, each node verifies if it has keys that it should send to the joining node. That is evaluated by seeing if it is its successor (with replication it verifies if it is one of its REPLICATION_FACTOR successors). If it is, it sends all key-value pairs on which the key is smaller or equal to the joining node's ID. After sending that information to the joining node, it can delete its key-value pairs (with replication it only does so if it is no longer a replica of those key-value pairs).

To transfer the information, the node transfers, for each key-value pair, a TRANSFER message, that works just like a PUT message.

3.2 Message Format

For proper communication between nodes, a message format was designed as for the Membership Service.

There are 6 main messages used in this protocol: GET, DELETE, PUT, TRANSFER, REDIRECT, DONE and ERROR. The specifics of each message are detailed below.

3.2.1 GET

The GET message has only one header, represented by the key it intends to get the value of.

The GET message has no body.

3.2.2 DELETE

The DELETE message has only one header, represented by the key it intends to delete the key-value pair of.

The DELETE message has no body.

3.2.3 PUT

The PUT message has only one header, represented by the key it intends to put the value in.

The PUT message body contains the value to insert in the key-value pair.

3.3 TRANSFER

The TRANSFER message is similar to a PUT message.

It only differs from PUT on the context it is used.

3.3.1 REDIRECT

The REDIRECT message has only one header with possibly several entries (depending on the replication factor). Each header entry refers to a node which contains the key-value pair previously asked for in a GET message.

The REDIRECT message has no body.

3.3.2 DONE

The DONE message has no headers.

The DONE message body may contain the value corresponding to a key previously asked for in a GET message or it may be empty if not prompted by a GET message.

3.3.3 ERROR

The ERROR message has no headers.

The ERROR message body contains the error message which describes what went wrong after a GET, PUT or DELETE message.

3.4 Main hurdles

4 Replication

In order to keep the system available and resistant to node crashes, a replication mechanism was implemented. It consists in replicating the key-value pairs in a REPLICATION_FACTOR number of nodes (in this implementation, the factor was set to 3), allowing the system to have more than one copy of a key-value pair. With that, whenever a GET request is made for a key-value pair stored in a node that is currently unavailable, it is possible to request that same key-value pair to one of the other nodes which hold the same information.

TRADE OFF ENTRE MAXIMIZAR REDUNDANCIA OU CONSISTENCIA

4.1 Implications on Membership and Storage services

The previously described services fit nicely with the implemented replication mechanism.

For example, when a node crashes and reinitializes, it sends a REINITIALIZE message as part of the Membership service protocol, to ensure that the Membership service information is up to date. This also means that the node might have lost information as part of the Storage service. As such, this message can trigger the key reposition protocol, // **TODO**

The implementation of the replication mechanism also leads to changes in the way the storage service works. One such change is that the PUT and DELETE messages send a REDIRECT message to the client indicating what are the other replica nodes on which a PUT/DELETE should be executed. To be noted that this emerging similarity is no coincidence - the implementation of tombstones effectively turns the DELETE operation into a special PUT, tagging the key as dead (on the filesystem it's represented as having the file extension ".dead").

Tombstones were implemented to eliminate the possibility of a replica node detecting that it has more information than another - more key-value pairs - and sending that "missing" information, while in fact that information should have been deleted and wasn't due to a DELETE message that couldn't reach it. With tombstones, key-value pairs are not actually deleted and instead leave a flag behind, indicating that the value for that key has been removed. Tombstone transfer was also changed and, when it happens, it is dealt on the receiving node as a DELETE operation, instead of a PUT operation like the rest of the TRANSFER messages are.

5 Fault-Tolerance

Whenever a node receives a GET message for a key-value pair it does not possess, it sends back to the client a REDIRECT message indicating to which nodes the client must connect to to actually receive that pair. With replication, that message has a REPLICATION_FACTOR number of nodes

listed.

This could lead to problems if, in fact, the membership view of the node that received the request was not up to date. One must note, however, that the Membership service seeks to keep nodes up to date, so it is highly likely that most active nodes are up to date. With this assumption, even if the REDIRECT message has wrong information, it will most likely send the request to a node that will correctly redirect the client to the key-value pair it seeks. With enough hops traversing the redirections, a successful GET is nearly guaranteed, even on a severely fractured cluster.

Another fault-tolerance mechanisms were implemented and described in the Membership and Storage service chapters.

6 Concurrency

Given the nature of the built system, it was important to assure that concurrent requests were properly handled and that the system did not become unavailable upon request handling. To solve this, Asynchronous I/O was used and Thread Pools assured the best possible usage of threads for a better performance.

6.1 Thread Pools

The system has to guarantee that the processing of requests, from either the Membership Service or the Storage Service, do not put the system availability at risk. As such, it makes sense to build a multi-threaded system.

That, however, might become problematic when a given node has to process a high number of requests. If a thread is created to process each request, the overhead associated with the creation and deletion of the thread, both in memory and time consumption - associated with the change of context - might reduce the performance of the system severely.

To solve this issue, Thread Pools were used. As such, whenever a new task needs to be handled, it is submitted to the Thread Pool, which has a defined limit of threads (we used the number of available processors on the machine times NUM_THREADS_PER_CORE, which is a constant we set to 4). When a task finishes executing in a thread of the Thread Pool, it is not destroyed, and thus the associated overhead is reduced. If all threads in the pool are already being used, new tasks are put on a queue and are later assigned a thread whenever one becomes available.

In our implementation, we used a ScheduledThreadPoolExecutor, since it allowed to schedule tasks (such as the MEMBERSHIP response upon a JOIN message with random delay) without having to use busy waiting or CompletableFutures. This executor is created in the Node class, and passed down to other classes which may need to submit other tasks.

6.2 Asynchronous I/O

The usage of Thread Pools does not solve the availability problem of the system. While it does optimize the thread usage, it does not assure that these threads are used only when needed. In fact, whenever a I/O operation occurs, such as reading or writing to a socket connection, the used thread will block while waiting for the result/action to be complete.

When a spike in requests occurs, the time taken to complete the I/O operations can be significant and may lead to a situation where the threads will be taken out of the thread pool faster than they are returned, since the thread will block. This would lead to the creation of more threads or, in this system that uses Thread Pools, it would lead to the pool queue to quickly grow. To avoid such a problem, Asynchronous I/O is a good solution.

Since these operations depend on kernel calls and are not a responsibility of the program itself, a callback system is used with Asynchronous I/O to free the thread responsible for the call. As such, after calling a read/write call asynchronously, the thread will return to the Thread Pool, where it can be used for another task. When the callback is called, the Thread Pool assigns another thread to complete the process. With this flow of execution, the problem presented above will not occur, since the threads will no longer block.

Asynchronous I/O is used in this system in all TCP Socket Connections as well as all Local Storage writes/reads. For all operations of this kind, an AsynchronousSocketChannel or AsynchronousFileChannel is assigned, to which the message to send/receive is sent to. Depending on the operation type, a ReadHandler or a WriteHandler is defined, which is executed when the asynchronous call is completed. This indication is given through a Java CompletionHandler, which functions as a

callback for the read/write operations. Both of these read and write handlers refer to a interface, which has two overridable functions: one that is called when the callback returns successfully and another that is called when an error occurs. This allows the handler to adapt according to the context it is called in.

7 Conclusions

All of the project's goals were achieved - all features were properly implemented and are complete, well designed and working as intended; the design and implementation of a distributed systems made clear the problems these systems may have and reinforces what was discussed in class on a theoretical level; and the development of the project as a whole was a solid practical learning experience.