Distributed and Partitioned Key-Value Store

2nd project of the Parallel and Distributed Computation course

Informatics and Computer Engineering

Class 6, Group 7

Lucas Santos (up201904517)
José Pedro Ferreira (up201904515)
Sérgio Rodrigues da Gama (up201906690)

2021/22

Contents

1 Introduction **3**

2 Membership Service **4**
2.1 Messages Format 4
2.1.1 Join 4
2.1.2 Leave 5
2.1.3 Membership 5
2.2 Avoidance of Stale Information 5
2.3 Fault-tolerance 5
2.4 Remote Method Invocation (RMI) 6

3 Storage Service **6**
3.1 Messages Format 6
3.1.1 Get 7
3.1.2 Put 7
3.1.3 Delete 7
3.2 Fault-Tolerance 7

4 Replication **8**

5 Concurrency **8**
5.1 Thread-Pools 9

6 Conclusion **10**

7 References **10**

1 Introduction

In this project we had to do a simple storage system to store arbitrary files. In order to access and manipulate these file (save, retrieve and delete them), each of them should have a unique identifier, which would be used like a key in a hash table. All the data should be stored persistently, so the keys and the data should be stored in the hard drive, rather than in the RAM memory.

In addition to that, we had to make this storage system distributed, which means that the key-value pairs should be stored partitioned among different nodes inside a cluster. However, this partitioning came with more challenges, because all the nodes should be synchronized with a consistence hashing algorithm and should know about each others existence.

Indeed, to manage and synchronize the nodes in a cluster, all the nodes should follow a protocol that we also had to develop, the membership protocol. This last one, should give a node the ability to join and leave from a particular a cluster, so it had to keep a membership log with all the active nodes and their activities.

Finally, we had to build a test client, which behaves as an interface for the storage system (store). This client should have the capabilities to control the functionalities described before, supporting 5 types of operations, put, get, delete, join and leave. The first 3 are relative to the management of files and the last 2 to the membership protocol.

2 Membership Service

The algorithm used by the storage system requires every node to know about each other. So, it was needed to implement a membership service that ensured the nodes synchronization. This protocol is implemented in the MembershipNode class, which is in the protocol folder.

2.1 Messages Format

In a matter of fact, the messages used to make the communication between nodes were kept as simple and smaller as possible, only having one space character separating each message element. In addition to that, they are all composed by an initial character that states the message purpose, for example, J for join. After that character is the node_id, which is the identifier or the IP address of the node that is sending the message. Followed by said node_id are some specific fields for each of the messages. The overall performance of the membership protocol benefits from this design, as less bytes are sent in each message. Therefore, the message used were the following:

Message	**Format**	**Size (items)**
Join	J node_id node_counter port_to_receive_logs	4
Leave	L node_id node_counter	3
Membership	M node_id /n logs_file	2 + no. of logs

2.1.1 Join

This message is transmitted to all the active nodes when a given node receives a join command from the test client. This transmission is part of the node initialization process, which is done when a node is joining a cluster.

Inside this message is a node_counter, which is both used to count the number of times a node joined and leaved the cluster, as well as see it's current state. As the counter is initially 0 and it is increased by one unity when it is joining and leaving the cluster, when the counter is even, the node is inside the cluster and when it is odd, it's not.

The initialization process consist of a node creating a TCP connection with an available port (the port that is also sent in the message, port_to_receive_logs) and start listening for connections, accepting up to 3 of them. When the node is already listening in that TCP socket created, it then multicasts the join message, using a UDP socket, and waits for the 3 connections.

On the other hand, when a node receives a join message, they all send their version of the last 32 membership logs, or more if needed to reconstruct the active members view. Then, the node joining the cluster compares entry by entry, all the logs received and creates the most up to date group of logs.

Lastly, when the node making the initialization process doesn't receive the 3 connections, it multicasts the message again and repeats the process. This

retrial happens up to 3 times. In the failure of the third re-transmission, the node assumes that it is alone, or that are only 1 or 2 active nodes, besides itself.

2.1.2 Leave

In contrary to the last message, this one only needs the message purpose identifier, L in this case, the node_id, and the node_counter. Indeed, this message is only used when a node wants to start the leaving process, which is when it receives a leave command from the test client. Also, like in the join initialization process, it is summed one unity to the current value of the node_counter.

To leave the cluster the node then transmits to all the other nodes a leave message, like in the join operation, using a UDP socket. When the other nodes receive the leave message, they just update their membership logs.

2.1.3 Membership

This message is used both in the update and in the join processes. Indeed, when a node is joining a cluster it waits for other nodes to send membership logs in response. The other situation is in the periodic update, explained in the next topic (Avoidance of Stale Information). Those membership logs are then sent using this simple message format, which is identified by an M character, followed by the node_id and the logs themselves.

The logs are parsed in way that each line following the node_id corresponds to a log register. When a line is equal to "end", the node knows it has read all the log lines from the membership logs.

2.2 Avoidance of Stale Information

In order to avoid stale information, so that the nodes keep their information about each other updated, the membership protocol ensures that, every second, one node transmits to all the other nodes their membership logs. This scheduled transmission is done via the ScheduleExecuterService interface created by Jabok Jenkov [1].

The node that does this transmission should be the most probable to be updated at all times, so we give this responsibility to the node first joining the cluster.

2.3 Fault-tolerance

So as to achieve fault tolerance in the membership service, we developed a mechanism that keeps the nodes with their membership logs update in the eventuality of a crash or any other event that out-dates a given node logs.

The mechanism works by checking the node's logs against the ones received

in the periodic message and see if there are anything out of order.

In the case where the received logs are more complete, the node is forced to leave and rejoin into the cluster, going through the initialization process again and receiving and creating the updated logs on itself.

On the opposite side, if the received logs are outdated in relation to the ones the node has, he immediately multicasts the membership message itself to update the other nodes. Moreover, it is up to that said node to keep the transmission of the periodic membership message.

2.4 Remote Method Invocation (RMI)

In alternative to the TCP socket listening to the client commands in the store program, we have also implemented RMI, but only for the membership commands, join and leave. This is a protocol to call methods belonging to a remotely registered object, that implements a specific interface. Moreover, the client only has to know that said interface in order to remotely invoke the methods wanted, that belong to the object. This interface is located in the 'rmi' package inside the root 'src' folder, named MembershipRmi.

3 Storage Service

This service covers the actual system storage functionalities that allow the retrieval, insertion and deletion of files. In fact, the key-value store takes advantage of a consistent hashing algorithm, in which the nodes correspond to a bucket and a change in the number of buckets doesn't obligate a complete remapping of the hash table. The nodes responsible for the storage of a given pair are the three whose hashed IDs are closest to the key, as we are using a replication factor of three. This service is mainly implemented in the StorageProtocol class, which is in the protocol package.

3.1 Messages Format

x Following the same logic as in the membership service, all the messages where kept simple and the smaller possible and the message elements where separated by a space character. Thus, the messages are composed of an initial character which indicates what operation to perform in the store, P for put (or F to force the storage of the exact number of new pairs given by the factor), G for get and D for delete.

Afterwards, there is a factor, which depends on the operation, being a replication factor for put and delete operations and a retry factor in the get operations. The last element is the actual value (or file content) in the put operation and the key (hash) for the get and delete operations.

Message	Format	Size (items)
Get	G retrial_factor key	3
Put	[P or F] replication_factor value /n END	2 + no. of lines
Delete	D replication_factor key	3

3.1.1 Get

The get operation is responsible for the retrieval of the value when given the key. So when the client sends the get command to any node, it starts by checking if (by chance) has the necessary value stored. If not, sends a get message to the closest node (in its perspective). This one, in its turn, either returns the value to the client, or, if the pair isn't in its possession, starts the retry mechanism, explained in the Fault-Tolerance below.

3.1.2 Put

When a client gives the put command to a node, three pairs key-value are stored, according to the desired replication factor. The first node sends the put command to the store he deems to be the closest to the key in question. The following nodes store the value if it isn't stored already and send to the node they think is next in line. There is a check in place to ensure that there are enough nodes to store the three copies. If this check fails, the nodes notify the client that not every copy was created and rely on the storage operations that happen automatically when a join happens, to restore the number of copies in the store to the optimal, three.

3.1.3 Delete

Deleting a key-value pair entails replacing every copy of said pair with its tombstone. A tombstone is a special key-pair, or marker, indicating that a pair with the corresponding key has been deleted. This is useful for when a node may jump a delete operation, because this way it realizes that a given pair has been purposefully deleted. Upon the reception of a delete message, the nodes start by checking if they have an active pair of the target key-value and replacing it with the tombstone if they do. They proceed to send a delete command to the closest node to the key or the next in line. If there are less pairs than the usual three, then the existing ones are deleted. In case the value isn't even stored to begin with, a get request is sent and if it turns out negative, the store informs the client that they are performing a delete on a non-existing pair, also avoiding any further resource waste.

3.2 Fault-Tolerance

We implement multiple fault tolerance measures, so that the file system stays as robust possible. The use of the replication policy is naturally the most important fault-tolerant mechanism. However, we also have some others, such as:

- **Get retrials**: When a get operation is executed and the closest node doesn't have the desired value, the node which didn't have the value makes the same operation in a random chosen node. When doing this retrials the retrial factor is passed and is decreased in order to prevent the cluster to be flooded with retrials. This less conventional approach allows to escape some possible dead end cycles, caused by faulty membership logs. Now if, for example, a given node thinks the closest node to a key is one that's not even one of the three responsible for said key, it's still possible to get the target value, given that one of the random ones chose at least has as closest one member of the referred trio.
- **Delete persistence**: The delete processor has a counter that starts with the replication factor and is only decremented when an active key-value pair is replaced by its tombstone. And the delete operation is only concluded when said counter reaches zero or all nodes have had their respective pairs deleted (when there are fewer nodes than the value of the replication factor, in this case, when there's either just one or two nodes), or finally, if the pair doesn't exist at all, fact which is double checked using a get request, to avoid further resource waste trying to delete non-existing files.

4 Replication

Without replication if a node goes down and becomes unavailable, the key-value pairs that were stored in that node also become unavailable. Therefore, by replicating the value-pairs by a determined replication factor, we avoid that problem.

In order to achieve that, we maintain three copies of each key-value pair stored. The operations are a bit more complex, as we explained above, but essentially, both put and delete are sent to at least three nodes, with the get operation being unaffected.

So when a node is down, there are, usually, other two still up and capable of retrieving the value for the client.

5 Concurrency

With the aim of not deadlocking the store, we made extensive use of Threads and Thread-pools. In a matter of fact, delegating tasks that deal with blocking sockets or other blocking operations, to be executed in threads, makes the store more robust and with the ability to process more concurrent requests.

When an instance of the store program is initialized, it creates a Node Thread, which itself creates a StorageProtocol Thread that deals with the reception of the commands from the client. This separation was done to maintain

the Object Oriented Paradigm used throughout the entire project.

Moreover, when the StorageProtocol captures a join message command from the client, besides the processing of the command being sent to a thread in its thread pool, it also creates a new standalone thread to start listening and deal with membership messages.

5.1 Thread-Pools

Firstly, we used the Jakob Jenkov's [2] implementation of Thread pools, which we putted in the servers package. The use of this thread pools eases the development in the sense that we do not have to bother on reusing and creating new threads, as the pool does it for us, and even in a more efficient way. Therefore, we create a thread pool in the Node class that is passed to the down to the following classes. Even further, the thread pool has a maximum of threads to be created in the pool equal to the number of cores, so that we keep the cost of creating a managing threads the lowest possible.

- **StorageProtocol**: In this case, each individual client command issued to the store is processed separately in a thread that is pushed into the pool previously created. This facilitates the processing of concurrent commands issued by the same, or even different clients, into the store.
- **MembershipNode**: Additionally, in this class we deal with all the messages received by other nodes. This is where a multicast UDP socket is actively listening and processing join, leave and membership types of messages. Thus, to keep receiving messages without blocking, we do all the processing for each message received in a different thread pushed to the pool.

6 Conclusion

In conclusion, this project was able to bring together the topics of distributed systems and the parallel computation, as we had to build a distributed storage system that dealt with multiple concurrent requests, with the help of threads.

We also, more or less, followed the work division given in the project specification [3], which lead us to a good balance in the team effort and environment. Although there are always improvements we can make to the project, we are really satisfied with the outcome and the overall performance of our storage system.

Finally, we really enjoyed working in this project, besides the tones of refactors we had to make, because we were always looking for the most efficient way to divide the code and make the best out of the available processing power.

7 References

[1] Jakob Jenkov. Scheduledexecutorservice. Technical report, 2014.

[2] Jakob Jenkov. Thread pools. Technical report, 2020.

[3] Pedro Souto. Parallel and distributed computation - 2nd semester proj. 2: Distributed and partitioned key-value store. 2022.