Distributed and Partitioned Key-Value Store

Parallel and Distributed Computing

Faculty of Engineering of University of Porto

André Pereira, up201905650
Margarida Vieira, up201907907
Matilde Oliveira, up201906954

1 Introduction

The goal 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

The key aspects of the membership service will be described in this section, including message format and explanation of other implementation details.

2.1 Membership Information Storage

Regarding the membership information, every node keeps himself some different files with data about the counters of each node in the cluster. Therefore, there are three types of files regarding the belonging of a node in his cluster. This data is essential for a correct messages flow and membership protocol functioning.

All files are kept inside the directory **\store**. And inside it each nodes has the respective folder: **\store\node<node IP>_<node port>mcast<multicast address>_<multicast port>**.

First of, each node keeps its **counter** in a file named upon its address, port and respective multicast address (**\store\node<node IP>_<node port>mcast<multicast address>_<multicast port> \counter.txt**).

All ever **membership logs**, are kept in a file, with a list of addresses, ports and each node counter (**\store\node<node IP>_<node port>mcast<multicast address>_<multicast port>** \ **membership.txt**).

Moreover, we have another file, that contains a list of **nodes**, represented by it's addresses and respective ports, indicating the nodes that are currently in the cluster, and that is calculated based on the current membership counter log of every node (**\store\node<node IP>_<node port>mcast<multicast address>_<multicast port> \members.txt**).

Every node is always represented by its address and port in this format: **<node IP>:<node port>** .

2.2 Implementation

JOIN

Upon a node creation, and when a join event is invoked by the Test Client, all multicast and unicast sockets are subscripted for that address and port, in new threads, and a JOIN Message is sent by multicast to the network.

When a node receives a JOIN message, he verifies whether its counter is coherent to the information it has, and updates it if it makes sense to him (only adding one to the current one), and then if it is the one responsible for sending him the updated information it sends it, via unicast.

Note that, a new node is only updated and successfully inside the cluster, when one of the following conditions is met: has received three MEMBERSHIP messages or has sent three JOIN messages asking for the membership information. After the JOIN message, the node waits a small interval of time to resend, in order for the other nodes to have time to send the information back.

After this, it can finally receive and hear the network messages sent by multicast and by unicast to him and, it can successfully store information regarding the key-value storing service and what that protocol may demand.

Join message format: JOIN;<node IP>:<node port>;<actual membership counter>;<number of JOIN messages sent>

Most relevant source files: Node.java, NodeInfo.java and JoinMessage.java

LEAVE

When a leave event is invoked by the Test Client, the node sends a multicast LEAVE message to the network.

When a node receives a LEAVE message, he verifies whether its counter is coherent to the information it has, and updates all information in the membership log file or in the cluster member, by deleting this member or updating the counter. When a node leaves the cluster, the cluster structure is going to be reorganized and this relapses over the membership protocol, as well as, delegates the key-value storage protocol to find a new node to be responsible for the data.

Leave message format: LEAVE;<node IP>;<node port>;<actual membership counter>

Most relevant source files: Node.java, NodeInfo.java and LeaveMessage.java

2.3 Election and Membership Messages

One main objective of our service is to **avoid the propagation of false information**. In order to fulfill it, we need to propagate membership information, with high probability of it being correct.

This process was implemented by having a temporary cluster leader who sends its known information about the cluster membership every second, so that everyone in the cluster is updated.

This leader is chosen by an election method that chooses the node with the **highest counters sum of all membership logs** in its cluster membership file.

Throughout the election, we first compare the counters sum of all nodes and build a sorted list by this number, assuming that the node with the highest counter is the one with the **highest probability of being correct**. In case of the counters sum being equal we just break the tie by comparing their hashes alphabetically/ numerically.

This enables us to have a sorted **list** with the Priority Member nodes **ranked by the probability of being updated**, and then be able to understand who is the leader and the next nodes. This list is also helpful for the mechanism of sending the membership messages, to a **new node upon its entry on the cluster**, by the three most updated nodes.

This is all only possible because of an election algorithm based on the known **Bully Election Algorithm**. Throughout the election, each node can be in one of the following states: Follower, Candidate or Leader.

At the beginning of the election, everyone is a candidate to be the leader, so everyone sends an ELECTION message, via multicast, with its IP address and port, as well as its actual counters sum of its membership log file. Everyone receives these messages and updates the counters sum of the nodes in its priority list.

After a small interval, each node sends an ALIVE message, via TCP, to the nodes which have an higher probability of being right, according to its priority list. If the **nodes with higher probability answer the message**, the node becomes **Follower**, otherwise it becomes **Leader** and sends a LEADER message via multicast to the network.

After becoming the leader, that node's job is to send the MEMBERSHIP message with the most recent logs of the cluster every second. When he sends 20 sequenced membership messages, he triggers an election, becoming himself a Candidate.

If it happens for the leader to crash or not send a MEMBERSHIP message when it is supposed to, the nodes who are expecting it, become candidates and trigger an election by sending an ELECTION message.

Membership message format: MEMBERSHIP;<node IP>;<node port>;<list of cluster members (separated by comma)>

Election message format: ELECTION;<node IP>;<node port>;<counters sum>

Leader message format: LEADER;<node IP>;<node port>;<counters sum>

Alive message format: ALIVE;<node IP>:<node port>

Most relevant source file: Priority Member, LeaderElectionState.java, Candidate.java, Follower.java, Leader.java, LeaderMessage.java, ElectionMessage.java, and AliveMessage.java.

2.4 RMI

In order for the Client to communicate with the nodes, the Node class implements the RMI interface, called Membership Service. When running a client, one of the Membership functions can be called in the client side (Test Client):

- **join()** to enter the cluster;
- **leave()** to leave the cluster.

Most relevant source file: Node.java, MembershipService.java and TestClient.java

3 Storage service

In this section will be explained the key aspects of the storage service.

3.1 Information Storage

Upon a Node instantiation, a NodeStore object is instantiated as well with the purpose of managing the storage of the key-value pairs.

Each node's NodeStore keeps information regarding the cluster members that the node is aware of in a TreeMap data structure, with the member's hash as key and the member himself as value. This data structure is particularly appealing because it guarantees log(n) time cost for the containsKey, get, put,

and remove operations, which is in line with the storage service's required functionality. Furthermore, the NodeStore sets a path and creates a directory for the node to persistently store the data under his management when it is initialized. The format followed for storing the data is **\store\node<node IP>_<node port>mcast<multicast address>_<multicast port>\storage\<key>.txt**.

Each operation offered by the storage service - get, put and delete - triggers a well defined sequence of events which will be described later in this section.

Most relevant source file: Node.java and NodeStore.java.

3.2 Implementation

In this section will be explained the program's behaviour upon receiving a GET, PUT or DELETE message, as well as the key aspects of Consistent hashing and how exactly they were put into practice.

GET

When a node receives a GET message, he verifies whether he is the node where the data is stored or one of the nodes where the data is replicated, in which case he sends an OK message, followed by the data itself. Otherwise, a REDIRECT message is issued with the replication nodes.

Get message format: GET;< key >

Most relevant source files: Node.java, NodeStore.java and GetMessage.java

PUT

Similarly to the aforementioned message type, when a node receives a PUT message, he verifies if he is the node where the data is supposed to be stored or one of the nodes where the data is supposed to be replicated. There are two possible outcomes here: whether an equal key-value pair already exists and the node simply dismisses the request, or not, in which case the node transmits an OK message and effectively stores the data sent afterwards. Otherwise, a REDIRECT message is issued to the appropriate nodes.

Put message format: PUT;< key >

Most relevant source files: Node.java, NodeStore.java and PutMessage.java

DELETE

When a node receives a DELETE message, he checks to see if he is the data's storage node or one of the data's replication nodes. There are two possible scenarios in this circumstance: whether he has the data in storage, in which case he sends an OK message, followed by the deletion, which occurs in all nodes where the data is replicated, or not and he proceeds with sending a REDIRECT message to the appropriate nodes. However, it is important to note that, in this case, deletion is not what it appears to be, and the implementation details for this aspect will be discussed further below in the Replication section.

Delete message format: DELETE;< key >

Most relevant source files: Node.java, NodeStore.java and DeleteMessage.java

Other relevant messages format

Redirect message format: REDIRECT;<list of at most 3 chosen members (separated by comma)>

Replication message format: REPLICATION;<key>

Sync message format: SYNC;<node IP>:<node port>;<membership counter>

Most relevant source file: StoreService.java, ReplicationMessage.java and SyncMessage.java

3.3 Consistent hashing

Consistent hashing is a distributed hashing scheme that assigns nodes to a place on an abstract circle, or hash ring, regardless of the number of nodes in the distributed store (hash table). This has a substantial impact on system scalability since it allows nodes to be introduced without affecting the overall system.

The key-value store uses SHA-256 to generate keys, so one can assume that there are no hash collisions. Besides, the hash function used to compute the keys is also used to hash the id of each node in the cluster.

Whenever a client requests a PUT operation, the key-value pair is assigned to the cluster node with the hashed id that is closest to the pair's key in a clockwise direction. As a result, moving clockwise until a first node is located is the way to find the node that will be accountable for storing that data. The same applies for a GET request - given the request's key, the corresponding storage node will be the first to appear on the ring in the said direction. This implies that the storage node responsible for a specific key can be efficiently found through a binary search.

This technique is quite effective when it comes to resizing the cluster, either by adding or removing nodes - either way, all that is necessary is to reallocate the data (by recalculating which will be the new storage node) that falls between the node that has just joined or left the cluster, respectively, and the first node appearing in a counterclockwise direction.

In this project, it was used a TreeMap data structure to store the key-value pairs and the Java built-in implementation of the binary search algorithm to effectively search for a key's value.

Most relevant source file: NodeStore.java

4 Replication

What happens if the node storing a key-value pair goes down? That key-value pair becomes unavailable? This section will discuss how replication was implemented in the project to prevent the aforementioned scenario from happening.

4.1 Implementation

Each key-value pair is replicated with a replication factor of three, i.e. it is stored in three distinct cluster nodes, unless there are not enough nodes in the cluster. Therefore, when a PUT message is received and all of the conditions regarding data storage are met, the data is stored not only in the original storage node, but also in the other two replication nodes. To accomplish this, when the message is completely received, the event is replicated to the other previously computed nodes, by their hashes. For example, assuming that a node receives a PUT message and he must save the file given, then, upon receiving all the data, two new PUT messages are sent by him, to the other nodes where the data must be replicated.

Most relevant source file: NodeStore.java

4.2 Implications on membership service

It is extremely important to guarantee that the storage service works and that the maintenance of the membership information is constantly updated. This means that, in order to make the protocol work, is really central to have the list of cluster members updated.

The replication mechanism that was implemented implicates that upon a node joining, if its hash is indicative of a replication position (being one of the three nodes that's next in the hash ring), then the files are transferred by TCP from one of the nodes to it, after a successful response to a REPLICATION message.

In case of a failure of a node, the synchronization, is done with a SYNC message, in case there are files that are supposed to be of its knowledge, upon receiving the sync message with the hash, then the nodes with the files should send every missing file of that node.

On the other hand, when a node leaves the cluster, it has the obligation of transferring its files to the correspondent node(s) indicated by the hashes circular ring.

Most relevant source file: NodeStore.java, ReplicationMessage.java and SyncMessage.java

4.3 Implications on storage service

The potential of one of the nodes where the data would be stored/replicated missing a DELETE operation is a problem that emerges from the use of replication. The usage of "tombstones" for deletion - a distinctive marker indicating that a pair with the matching key has been deleted - was implemented as a solution to this problem. As a result, rather than actually deleting the key-value combination upon its deletion, it is replaced with its "tombstone".

Moreover, when a node receives a REPLICATION message and the data is marked as tombstone, he denies the replication by notifying the sender node that the information that he is trying to replicate has been deleted.

Other detail we also acted on, is the fact that we always use a different hash to every file, because it not only depends on the file's content, but on the date and time it was first inserted in the system. This helps to avoid problems with ever existing tombstones with the same name.

Most relevant source file: NodeStore.java and ReplicationMessage.java

5 Fault tolerance

Apart from those already mentioned in the previous sections, there are a few things to note when it comes to the program's fault tolerance.

The REDIRECT messages can be interpreted as a fault tolerance mechanism because, in addition to ensuring data persistence when a node fails, it also prevents the program's flow of execution from crashing - when a node receives a request that, for whatever reason, he can't process, he responds with a REDIRECT message to those who will be able to properly handle it.

Additionally, both SYNC and REPLICATION messages are also worth mentioning in this section. When a node tries to join the cluster without ever leaving it, as previously stated, that is recognized and a SYNC message is issued instead. If another node in the cluster identifies that the node syncing should be one of the nodes where the data he holds should be replicated, he sends a REPLICATION message with that same data.

That message being received, the node syncing can accept it, if the data was not indeed in his possession, or reject it, if the data was already in his storage. This ensures that all cluster information is kept in the correct nodes, even when an inconvenient occurs.

In order to avoid considering wrong information about the membership members and logs, upon receiving messages with data about the members, every node tries to merge the information received with the actual information it has. This merge is done, starting by checking the common nodes and choosing the highest counter from the file or the message received. Then, when there are not in common members, we always check the current log counter to decided whether it is in or out of the cluster.

Most relevant source file: NodeInfo.java, MembershipMessage.java, ReplicationMessage.java and SyncMessage.java

6 Concurrency

In this section, will be discussed the key aspects of the concurrency design of the implementation.

6.1 Thread-pools

In order to run specific parts of the service in parallel, thus avoiding blocking calls from delaying the entire program's execution, we used thread-pools, which were crucial to parallelize the sending and receiving of messages, as well as the processing required for some operation.

Thread-pools were used in particular **to receive and handle multicast and unicast messages**, as well as for all jobs that did not last the entire node's lifetime in the cluster (which ends up being what characterizes a job, with exception to the ConsistencyJob) - JoinJob, LeaveJob, SyncJob, InitializeStoreJob and CloseStoreJob. It is especially important to note that all these jobs will eventually terminate and they can be executed in parallel with each other and with the main flow of the program.

Furthermore, some operations that contribute to nodes' integrity within the respective cluster, such as the UnicastExecutor, MulticastExecutor and ConsistencyJob, also made use of thread in their implementation. However they are execute as a simple thread and not by the thread-pool since they will be running until the node leaves the cluster.

Additionally due to all the processes concurrency he had the need, in some cases, to use mechanisms of thread synchronization in order to guarantee the integrity of the data.

Most relevant source file: Node.java, UnicastExecutor.java, MulticastExecutor.java and ConsistencyJob.java

7 Conclusion

Throughout this challenging project, we learned more about the concerns and challenges of a distributed system, while trying to implement a consistent program that supports a key-value storage system. It had great results in our learning and understanding process, that lead to an interesting final product.