U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Licenciatura em Engenharia Informática
e Computação
2021/2022 | 2nd Semester

Distributed and Partitioned Key-Value Store

Parallel and Distributed Computing

Bruno Rosendo | up201906334@up.pt
João Mesquita | up201906682@up.pt
Rui Alves | up201905853@up.pt

03/06/2022

Index

Introduction

The main goal of this project is to develop a distributed key-value persistent store for a large cluster. The main features requested for this project were implemented: the membership service and the storage service, which include a third internal service responsible for transferring files between nodes in membership events.

Aside from those, the following enhancements were also implemented:

- **Replication:** To increase availability, key-value pairs should be replicated with a given factor.
- **Fault-Tolerance:** It's possible for a node to be down for a long time, therefore having an outdated view of the cluster. The system must handle these types of situations.
- **Concurrency:** Having a large distributed system with a potential great number of clients requires thread parallelization in each node. Solving possible concurrency problems (e.g. race conditions) is an important factor of the system.
- **RMI:** Remote Method Invocation provides an easy-to-use interface to the client, facilitating the integration of our system.

Membership Service

Message Format:

Membership Messages are composed of two parts: a header and the body. The header consists of a sequence of ASCII lines and it contains control information, such as the type of the message (request and reply) and an action or a status, according to its type, respectively. Each ASCII line terminates with the codes of the *CR* and *LF* chars. The header terminates with an empty line, separating it from the body. The body is an optional field containing data (a byte sequence). Since the cluster is derived from these messages, they also the node ip and its respective port from where he will listen to the TCP messages.

Membership Protocol:

Every time a node joins or leaves the cluster, the node sends via multicast a JOIN/LEAVE message, respectively, which includes the membership counter. In order to join, a node must receive membership messages from 3 other nodes. If it doesn't receive those 3 messages, then it waits for a random time and retransmits the JOIN message up to 3 times. After that, if he still didn't receive 3 messages, it assumes there are not 3 nodes in the cluster and joins it. In order to avoid overloading the network, nodes that have already replied with a membership message successfully, will not resend it, unless there was another membership change meanwhile.

Log Merging Algorithm:

Every node maintains a log file which contains the most recent action regarding each node. For this reason, this algorithm needs to verify previous occurrences of a node in the log upon new insertions. To achieve maximum efficiency, we iterate the current log file only once to build a HashMap with the nodes inside it. Afterwards, we go over the new logs and, for each one, compare the membership counter. If the new counter is higher, we then proceed to update the membership log file, removing any previous occurrences of the node and inserting the new occurrence at the top. Furthermore, one additional verification is made to confirm whether the membership log is coherent with the node map. Hence, if the new log indicates that the node should be part of the cluster, that is, the membership counter is even, then we add the node to the node map if it was not yet part of it. Likewise, if the log indicates the node left the map, we remove it from the node map. This is done in handle membership response and update membership info functions.

Election:

On account of the unreliable nature of the *User Datagram Protocol*, *nodes may miss membership change messages*. Therefore, in an attempt to cover these failures, every 1 second, a *coordinator node* of the cluster sends a multicast membership message with the most recent 32 membership events in its log. Upon receiving such a message (from now called *election ping*) a node updates its membership log as well as its membership cluster, using the methodology explained in *Log Merging Algorithm*.

The election mechanism used is based in a ring network. A new election may start by various sources: Membership Join (1), Membership leave (2), Election Ping (3) or due to a timeout without receiving *election pings* (4). The node starting the election sends a TCP election request to the next node in the ring, which in turn, will propagate to the following node if it considers the membership view of the sender to be more up-to-date. The algorithm that evaluates whether the received membership information is more updated will be explained further on. Consequently, a node will be elected if and only if it receives its own election request, meaning that the request has done a full trip around the circle. Since the election requests are made through TCP, we can assume these messages are reliable. However, if a cluster's node crashes it will block all election requests from succeeding. Thus, every election request has a timeout, after which the sender considers the node to be unavailable and sends the same message to the next node in the ring.

Upon a membership join, the joining node will start an election after updating its view of the cluster with the membership information from other nodes in the network. On the other hand, when the *coordinator node* leaves the cluster it sends a special *Election Leave* message that transfers the full view of the membership to the node with the lowest id. That node is chosen due to the algorithm that is used to check if the new membership information is more updated. Additionally, when nodes receive *election pings*, they verify if they are more updated than the current leader. If that is the case, they initialize a membership request as well. Finally, to handle special cases such as the *coordinator* crashing, the clusters' nodes have a timeout after which they will start a new membership request if they do not receive an election ping in the meantime.

Log Recency Comparison

This algorithm is responsible for analyzing the received membership log and comparing it to the current node's log. It starts by transforming both membership information into a data type that is more efficient to analyze, *HashMaps* mapping the node id to its membership counter. Afterwards, the keys of both maps are joined onto a *Set* to avoid duplicates, which will then be iterated and analyzed one by one. For each node id, the method evaluates the membership counter on both the new and current set, setting it to -1 if not existent. It then proceeds to compare the counters and if the new counter is higher, it increments the global counter that starts at 0 by 1. If, on the other hand, the new counter is

lower, the score decrements by 1. As a tiebreaker, if the global counter ends being 0, the new node is considered to be more updated if its id's hash value is lower. The global counter is then returned by this method and used to decide whether a node should propagate an election request or if a node should begin an election request upon receiving an *election ping*.

Storage Service

Message Format:

Identical to the membership service, the storage service uses a format with a header composed of the type of message (e.g. request and reply) and the action (e.g. put and get), accompanied by an optional body containing data (a byte sequence). The messages between nodes use the TCP protocol, which are sent by the Sender class and processed by the TCPListener.

Consistent Hashing and Map of Nodes:

In order to implement and properly make use of consistent hashing, Java's data structure *TreeMap* was used. This seemed like the best choice for this use case, after taking into account its Red-Black tree based implementation, meaning it can naturally guarantee *log(N)* time complexity in the system's most needed operations (i.e. get, put and delete). The most important uses of this structure are in the calculation of the responsible node given some key (binary search), and when getting the previous and next node in the map.

Redirecting a request to the responsible node:

The client should be able to make a key-store request to any available node in the cluster, even if it is not responsible for the file associated with that request. Therefore, the system must be capable of redirecting the operation to the right node. The general flow for this type of situation is the following:

- The client sends a key-value operation request to a node.
- That node receives it and calculates the node responsible for that key.
- If that node is the responsible one, then it processes the request.
- Otherwise, it sends a redirect message back to the client.
- Depending on the reply, the client re-sends the request to the right node.

Replication:

In order to increase the system's availability, key-value pairs should be replicated between a number of nodes. Without replication, if a node that stores a given key-value pair goes down, then that pair becomes unavailable until it comes back up.

In this project, a replication factor of 3 was used. It's possible to have less than 3 copies of the same pair if a node is down, but the system can quickly recover from that when the node reconnects to the cluster. Regarding the position of the replicas, it works in a similar way as in Amazon's Dynamo, meaning that the key-value pair will be stored in the responsible node and in the 2 others following it (in other words, replication factor - 1). If the system has less nodes than the replication factor, then every bucket contains all the pairs.

The use of replication has strong implications in the storage and transfer services, which is explained in the sections below.

Put Operation with Replication:

With the use of replicas, the *put* operation needs to be readjusted. Aside from verifying the responsible node, it needs to ensure the new key-value pair is being replicated. The way this is done is straightforward: take the following *replication - 1* nodes and send them a request to save the new pair, as can be seen here. Then, the receiving node processes the request by simply saving the pair sent, even if it's not the one responsible for the respective key, which is done in the saveFile function.

Delete Operation and Tombstones:

Unlike the *put* operation, here, the implications caused by replication aren't so simple to solve. Since the pairs are being replicated in multiple nodes, the execution of operations might occur in different order in different replicas. For example, a node might process a *delete* operation before the respective *put*, or it might miss a *delete* operation and later on spread that key-value pair to new nodes joining the cluster (they're usually known as *zombies*).

The solution to this kind of problems is the use of **tombstones.** In this project, a tombstone is simply a file informing the node that a given pair has been deleted which stores a timestamp of when it was deleted. Note, however, that the respective pair isn't actually deleted upon the creation of the tombstone. That is only done later on, after the system has had time to update all of its replicas.

That being said, the *delete* operation calls a *safeDelete* function that creates a tombstone for the given key-value pair. Afterwards, it also sends a *safeDelete* request to the nodes containing that pair's replicas, as can be seen here.

In order to avoid overloading the servers' storage with deleted files, each node has a thread running a *TombstoneManager*, which periodically checks the tombstones, deleting it and the respective key-value pair if the defined expiration time has passed (in this case, 10 seconds were used).

Updating the Cluster in a Join Event:

Given the nature of the cluster, it's necessary to readjust the storage whenever there's a membership event (either a *join* or a *leave*), which is all managed in the TransferService class. As for the *join* event, we need to consider two situations:

The cluster has less nodes than the replication factor- In this case, the new node must copy all the key-value pairs in the cluster. For this, it asks each node for a list of their pairs' keys and then asks for the respective files, after filtering out unnecessary replicas.

The cluster was already fulfilling the replication factor- In this case, the new node should only copy the pairs for each he's now responsible, and the replicas of its previous nodes. It should also tell the nodes that previously contained the replicas to delete them, since they're not necessary anymore. These operations are done here.

Updating the Cluster in a Leave Event:

Similarly to the previous section, the *leave* event also has to consider two situations:

The cluster will have less nodes than the replication factor- The node doesn't need to do anything because every bucket already contains all the pairs.

The cluster is still fulfilling the replication factor- In this case, the node must send its files to the nodes which will store its replicas. It's not necessary to send the files to the new responsible node, since it already contains the replicas. These operations are done here.

Recovering from a Crash:

After a node is crashed for some time and comes back to the cluster, it's not enough to just recover its view of the cluster, it must update its storage as well, since its replicas could have been deleted or and new files could have been added to the cluster.

For this reason, the recovery process of the storage system executes the following steps:

- Get the pairs the node's responsible for and save their keys, from the next node in the cluster, and copy the ones it doesn't yet have.
- Get the replicas it should store from the previous nodes, similarly to a join request. Save their keys as well.
- Delete all the pairs whose keys were not saved in the previous sections, since they're invalid in the current state of the cluster.

Fault-tolerance

In the protocols specified above, a lot of common failure scenarios were covered. However, it's virtually impossible to cover all the scenarios. In this section, we specify a few situations that were identified and how they were solved:

When a node is down for a long time and misses many membership events: In this situation, the periodic membership messages may not be enough for the node to learn the current view of the cluster. For this reason, the system's recovery process is capable of requesting an updated version of the nodes' map, preventing this problem.

When the membership view of the node to which the client sent a request is not up-to-date: In this situation, the request sent by the client will probably not be successful. This is mitigated by the election/leadership mechanism. By having a leader constantly updating the other members of the cluster, these types of errors are much less likely to occur.

Concurrency

Thread Pools:

To allow for the concurrent execution of multiple requests in the server nodes, Java's threads were used. Since the system has a considerable amount of quick tasks, it's a very good situation to use a thread pool, because it reuses previously created threads to execute new tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.

More specifically, the system uses a cached thread pool, given by the Executors class. The reason for this choice, instead of using a fixed number of threads, is that the system can have an unpredictable frequency of requests and their quantity shouldn't be large enough to significantly slow down the server by using too many resources. If we used a fixed number of threads, it's likely that many of them would just stay idle for a long time in some situations and, in others, the chosen number of threads might not be enough to handle the incoming threads, making it a difficult choice. For example, in a real life storing

system, there'd be a lot more requests during the afternoon than during the night, when most people are sleeping. Therefore, it's useful to shutdown threads not being used for a long time and reopen them when needed.

In order to avoid collisions when accessing the file system in concurrent threads, the *synchronization* mechanism was used. To maximize the availability of the file system and therefore increase the application's performance, the objects used for the synchronization locks were the keys of the files being accessed (the same *hash* being used to save key-value pairs). Since Java uses a constant pool to access and save its strings, it's possible to use the *intern()* method to use those objects, thus guaranteeing that equal strings will correspond to the same object.

This way, the performance of a storage system is significantly improved. Without the use of concurrency, each client would have to wait for the completion of the tasks requested by all the other clients before him, which could result in an unusable application.

It'd also become impossible to have *working threads* in the system, like the one used to handle the nodes' tombstones.

Asynchronous I/O:

Using the standard synchronous I/O, the threads requesting the read/write operations to the file system need to wait until the operation is finished, which can take a considerable amount of time, depending on the management done by the operating system. In the time they're just waiting for those operations to finish, the threads are still active and cannot be used for other tasks, therefore, they're just wasting the computer's resources.

On the other hand, if asynchronous I/O is used, the threads can stop running the moment they ask for an I/O operation, and come back to the task when the operation is completed. This allows for other tasks to be executed in that same thread, instead of creating a new one for that purpose. Therefore, asynchronous I/O can significantly reduce the overall number of threads active in the system.

Conclusions

With this project, we understood how complex it is to handle a multithreaded and distributed system of this sort, especially a storing application like the one implemented here. There's an enormous amount of variables when taking into account efficiency, availability, consistency and fault-tolerance. However, a hard task like this one also lets us increase our knowledge on the topic and understand how a system like this actually works in real life.