U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Distributed and Partitioned Key-Value Store

LEIC - Computação Paralela e Distribuída 3º Ano - 2º Semestre

Turma 5
up201906834 - Óscar Gonçalo Martins Esteves
up201905396 - Pedro Magalhães Moreira Nunes
up201904979 - Luís Carlos Barros Viegas

Distributed and Partitioned Key-Value Store

In this project, we developed a distributed key-value persistent store for a large cluster. The objective is to create a reliable means of storage at a large scale. It stores arbitrary data objects that are accessed by means of a key, similar to a hashing table. These values are stored in persistent storage and they are partitioned among various cluster nodes, and the design is somewhat based on Amazon's Dynamo, utilizing consistent hashing to partition the key-value pairs among the different nodes.

Membership Service

Join works with any amount of nodes, and when a node enters, it sends a message via IP multicast in the following format:

Leave

Periodic membership -> envia info de membership periodicamente para evitar stale info?

Log merging como da merger de logs de outros nós c o dele?

With avoidance of stale info: Must include message format and description (how and why) of missing details (including fault-tolerance).

Storage Service

Message Format

The first line tells us which command we're running (put, delete, get, putreplica)

The second line tells us the key of the value we're dealing with. Those two consist of our header

The body is only used for put and putreplica. It holds the content we're saving

Put

Put was successfully implemented according to the guide. A store creates its file system, creates a txt with the hashed value of the file content (see hashing and binary search)

Delete

Delete was successfully implemented according to the guide

Get

Get was successfully implemented according to the guide

Putreplica

A put which will be only called on valid nodes, making the replication process more easier to handle. All the preprocessing is done before calling it.

Hashing and Binary Search

A TreeMap was chosen since it holds a similar behavior to the one described in the guide.

There was also a function that allowed us to choose the node according to the guide - ceilingEntry(). (*The ceilingEntry(K key) method is used to return a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.)*

Features not implemented

Pair transfer and fault tolerance were not implemented. Thus if a node leaves or crashes all its content is lost, besides if it's saved with the replication implementation.

Replication

Our replication works in the following way: the node where we are storing our file has a log with the known nodes. We read that file and create a tree map. While creating it we read the number of known nodes. If the number of known nodes is under three, that value is taken as the replication factor. If not, three is taken as the replication factor.

During put we find all the nodes that are valid for a given key. With that we mean returned by the ceilingEntry() function. If no node is found, replication will not be taken regardless of the node factor. That happens because our get function will only be able to find nodes that are valid. Then putreplica is called, and since we already know its a valid node it automatically inserts it in the node store.

If a node crashes, then we should be able to find the contents of said node in the rest of the cluster if replication was done. Since the replication is done only in values that are returned by ceilingEntry() our get function should be able to find all the correct nodes.

We weren't able to implement the delete function perfectly at the time of writing this report. It deletes the file, but we didn't add the replication factor to the function, so it never really knows how many nodes have said file stored.

Concurrency

Thread-Pools

Thread-pools were implemented and are used in the Store class. This allows nodes to be able to process more than one request at a time.

Conclusions

Seeing as an outage may have, in certain types of platforms,

massive consequences, reliability is a very important requirement when building storage systems. Utilizing consistent-hashing as well as replication and thread-pooling, the result is a highly available means of data storage.