DIDAProject 2021/22 - Group 7

João Pedro Lopes
joaopedrolopes00@tecnico.ulisboa.pt
ist193588

Francisco Rosa
francisco.rosa@tecnico.ulisboa.pt
ist193578

Pedro Morais
pedro.morais@tecnico.ulisboa.pt
ist193607

Instituto Superior Tecnico - Taguspark
Av. Prof. Dr. Cavaco Silva
2744-016 Porto Salvo

Abstract

The main goal of this project was to design and implement a simpler version of a distributed FaaS cloud platform. This platform runs applications composed of a chain of custom operators that share data via a storage system. A scheduler receives a script sent by applications that specifies the order of execution and assigns a worker to initiate the execution of the aforementioned chain.
The application operators function calls work in a daisy chain sequence, whereas the application data is, as mentioned, saved in a storage system maintained by a set of nodes.
There are several nuances and challenges to take in consideration when designing and implementing a system like this. Mainly: how to access and replicate data records, where to execute each operator, how to maintain consistency between the storage nodes and how to guarantee the correct functioning of the replication due to the innate challenges related to the implementation of a distributed system.

1 Proposed Implementation Solutions

1.1 Gossip Protocol

1.1.1 Introduction to Gossip

Gossip is a communication protocol that allows replication in distributed systems. Contrary to primary-backup, it does not have a centralized node or cluster to disseminate information. However, every node maintains a "keep-alive" system with other nodes. A keep-alive is a periodic message sent by a server to other servers to show that it's functioning and hasn't crashed. Whenever a server crashes or goes down, it stops sending these periodic messages and all the other stakeholders in the system find out. Since a message is sent per server to every other server at every period, which is a rather expensive operation for any node cluster of a decent size, it is considered that gossip has poor scalability.

1.1.2 Usage of partial replication

Our implementation of the replication protocol is a variant of Gossip which uses partial replication. In partial replication data objects are associated to a primary replica and every write and update operation for that data object is done in the said replica. Later the operation are propagated to a subset of the storage servers according to a decided replication factor (An integer that represents how many servers will store copies of that specific record, including the primary replica server).

1.1.3 Replica operation logs

As writes and updates are completed in the replica associated with a specific record, the structures of the grpc requests and their respective DIDAVersions are stored in two operation logs, respectively the write and update operations log. This log is later propagated on a timer that is defined in the script that initializes the storage nodes. This timer does not necessarily have to be the same for every replica as it can vary independently. As the logs are received by a replica, they are executed according to the total ordering

property, which states that ”If a correct RM (replica manager) handles request r before request r’, then any correct RM that handles r’ handles r before it”. Whenever the system faces an update operation with a specific version number, it needs to execute every write operation for that record that has a lower version number than the update’s number. Also, these writes are sorted and executed by the lowest version number.

1.2 Worker daisy chain

1.2.1 Daisy chain execution

A daisy chain is a layout where nodes are linked together either in a sequence or in a ring. In this case, workers are linked together in a ring.

1.2.2 Selecting the first worker

The scheduler’s first task before building the DIDARequest is assigning the order that the workers will execute the request. The first worker is the one whose port number has the smallest value. The order of which the others are assigned to the operators is irrelevant. Since we are executing in a daisy chain, if there are more operators than workers, the sequence will loop back to the beginning of the ring to continue the execution of the request, similarly to the Round-Robin algorithm.

1.2.3 Reflection load

Whenever a worker receives a DIDARequest, he also receives the name of the operator and, using reflection load, it will start searching for the code that needs to be executed. This code is executed by the storage proxy that has direct contact with the storage nodes which are associated to the replicas due to the consistent hashing mechanism. After executing the operator, the worker increments the value of the next attribute of it’s associated DIDARequest and passes on said request to the next worker in the daisy chain sequence, with a certain delay, defined in the script.

1.3 PCS - Process Creation Service

In the PuppetMaster, it’s verified if there is an existent connection to the server of process creation services. If there isn’t one, a grpc connection is created and stored in a client map data structure. Finally, a process is created according to the parameters and prerequisites necessary.

1.4 Crash Handling

Every storage node has an associated hash ID. If any node crashes, the connected nodes will have to compute again which nodes are closer to themselves. This computation works by searching the server ID whose ID is immediately above the storage node’s ID. After the computation is completed, the workers and storage nodes are notified.

2 Algorithms used

For this project we implemented the consistent hashing algorithm. Every server has associated with itself an hash ID and every executed object that needs to be stored in the aforementioned servers have an associated hash ID.
The hash IDs of the objects are compared with those of the servers and every object is associated with the server whose hash’s ID is the first one above the object hash’s upper bound. If there isn’t one, it will be associated with the lowest server hash ID.

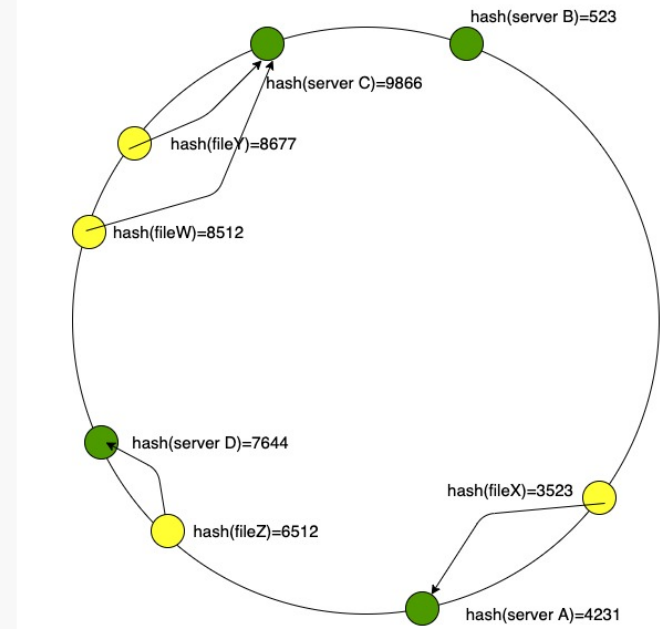

Figure 1. Consistent hashing diagram

3 Design Decisions

The design diagram is present in the appendix.

4 Final Notes

The .dll file needs to be in the main directory of the project and every file, be it scripts or necessary files to run them, must be in a folder named scripts also present in the main directory. The debug feature is not implemented due to conflicts with a service unavailable exception on PuppetMaster’s server at port 10001. Worker load balance and

concurrent data structures were not implemented due to time constraints.

5 Appendix

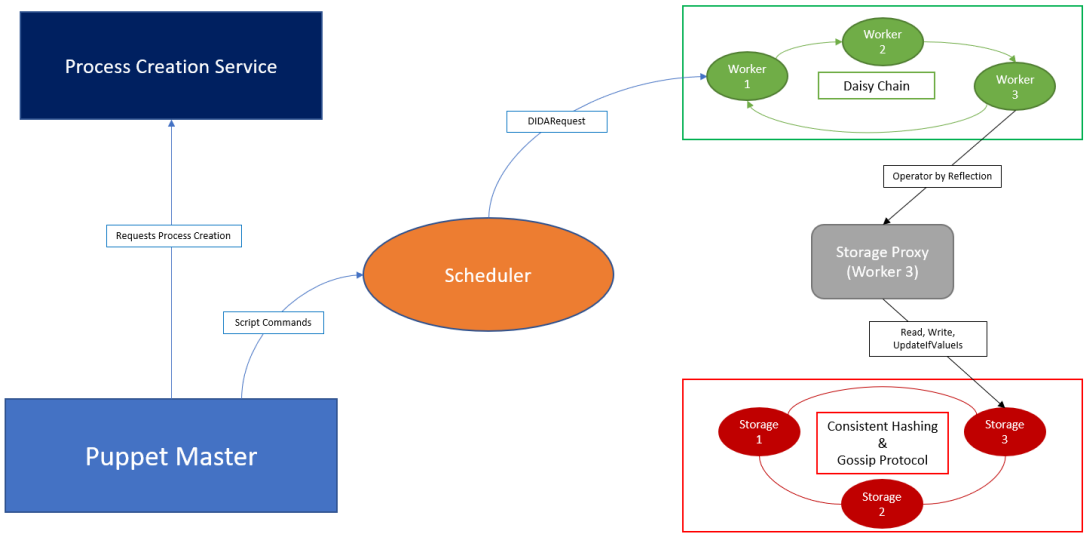

Figure 2. Design implementation diagram