Design and Implementation of Distributed Applications 2021-2022

David Miranda
93697
Instituto Superior Técnico
Computer Science
david.domingues.miranda@tecnico.ulisboa.pt

Bernardo Várzea
102150
Instituto Superior Técnico
Computer Science
bernardo.c.varzea@tecnico.ulisboa.pt

Abstract

The goal of this project was to design, implement, and evaluate a simplified version of a distributed function-as-a-service cloud platform. Using Gossip and consistent hashing combined with the idea of Lamport Clocks, helped us deliver a work that combines Consistency and Availability.

1. Introduction

We are going to talk firstly about the system interface that we used to implement this system, secondly about the ideas behind our implementation of the algorithm that permitted consistency between data and lastly main problems that we faced/thought and how we resolved that.

2. System Description

In this system we have 5 main components: PuppetMaster, Process Creation Service, Scheduler, Worker and Storage. The architecture is to use the PuppetMaster as a master to the system connecting to Process Creation Service(PCS) to create the other components. The Scheduler then contacts with the Workers that contact with the Storages.

2.1. PuppetMaster

The PuppetMaster is a simple console with a GUI, making possible to the users to insert a script with predetermined commands and also to see debugging data or to insert commands on application runtime.

2.2. Process Creation Service

The PCS is used only to start the Scheduler, Worker and Storage processes, with specific URL's and possibly in different machines.

2.3. Scheduler

The Scheduler is responsible to receive commands from the user to run an application and create a task to run that application and give it to a Worker so he can start a chain of operations.

2.4. Worker

The Worker is basically a computing node. This means that the worker computes a operator from an application request sent by the scheduler and from worker to worker it iterates operator by operator, executing tasks to the storages and passing the outputs and metadata, given by that operator tasks, through workers.

2.5. Storage Proxy

The Storage Proxy is in charge to contact the storages, and this is because the worker sends a request to this storage proxy and it is responsible to do all of the messages exchanges and metadata updates.

2.6. Storage

The Storage is what the name means, a storage node. Which is a node where the computed data by the worker is stored at. It does this by being multi-versioned, maintaining data consistency and fault tolerance.

3. Motivation

The motivation of this work was to develop a system that could tolerate fails in storages, has consistent data between all workers and can preserve causality between a chain of an operation runned.

4. Functions

The user can call one of this three function in the operators that he wants to run in our service.

4.1. Write

In this function the user gives a value to the system and the id of the variable where he would like to save the value. The system stores the variable and returns the version to the user.

4.2. Read

In this function the user gives a specific version or not to the system. The system returns the value corresponding to the specific version if found, or the most recent value (obeying the principle of causality) and returns the value and the version to the user.

4.3. UpdateIfValueIs

In this function the user gives a old value and a new value to the system, and also the id of the variable where he would like to save the value. The system waits for all writes and updateIfValueIs that can be waiting to be done and when all the previous writes/updates are completed (obeying total ordering) the system then updates the value ,if the old value corresponds to the most recent value existing in the system(do not forget that if two apps are trying to updateIfValueIs in the same variable with the same parameters only one will succeed and the other will receive a message regarding the failure of the operation).

5. Algorithm - Conceptual View

The algorithm we implemented used Gossip as the main solution. For that we needed to use an array Timestamp reflecting the version of the list data values accesses by the Storage Proxy. This Timestamp contains an entry for every storage that was created in the setup of the system. This view is based in the existence of 3 operations.

5.1. Important Data Structures

5.1.1 Worker

The Worker only has a MetaRecord, a class that contains information to replicate a chain of commands. What is important here is that the Worker in the MetaRecord also has a Timestamp and a Dictionary with all the identifiers and the last versions seen by all the previous and current worker involved in the current chain of operations.

5.1.2 Storage Proxy

The Storage Proxy (SP) has a timestamp that will be changed by all operations done and will be sent to the worker so he can merge the timestamps.

5.1.3 Storage

The storage has the most data. Including a **Value Timestamp** that is a timestamp reflecting the updates that result in the most recent version of the value stored. The **Update Log** is an array of **Records** that represent updates that cannot yet be performed by the storage (because the value is not yet stable(there are updates that didn't yet were preformed to that value)). The **Replica Timestamp** that is a timestamp reflecting the updates that are yet to be done but were already received and are stored in the Update Log.

5.2. Read

When the SP performs a read request q, to the Storage, it sends his timestamp. And the read can only be preformed if

$$q.TS \leq ValueTS$$

And everytime a new write/gossip is executed in the storage, regarding that key, the previous equality will be checked again.

5.3. Write

We can separate a write in two parts: Request and Validation.

5.3.1 Request

In this part the SP sends to the Storage a write request w. In this request there is the SP current timestamp regarding that variable **r.prev**, the Id of the variable **r.id** and the new value **r.val**. The Storage after receiving this request increments the i'th element of its replicaTS, where i is the storage Id, given by the PuppetMaster in the setup. After that a new timestamp ts is created, that time stamp is the copy of r.prev but replacing the i'th element with replicaTS' i'th element. After the creation of this timestamp it's created a Record to store in the Update Log.

Record : (i, ts, r.prev, r.id, r.value)

After this Record r is stored the Storage sends ts to the SP.

5.3.2 Validation

When the SP receives the ts it will compares it with its timestamp. With this compare operation are 2 ways. If the ts is smaller than SPTs, then the SP will send a message not

acknowledging that ts. If it is bigger then SP sends a success message. When the Storage receives those messages there are two options. If he received the success message he writes the value that its save in the record and update its valueTS with r.ts (record saved). After that deletes the record from the Update Log and sends the version of the written data to the SP with

5.4. Update If Value Is

In this case the SP sends a request r to the storage with the Id of the var, the old and new values and the SPTimeStamp r.prev. The Storage receiving this request will send a ping to all storages, and after receiving a reply of that ping we can now be sure that, there is no more messages on the network. Now the storage only needs to wait for the update Log to be empty of records regarding the target Id. The storage now is in the most updated state. After that if the value matches there the update is done and the ith value if the valueTS is incremented by one. The r.prev is merged with the valueTS updating the SP with important data (case after the crash of the previous storage that was responsible for the variable with this ID). If the updateIf is not successfull then its sent a null version to the SP.

6. Algorithm - Implementation

We implemented a Consistent Hashing model making us able to, in a time span with no crashes, only one storage is responsible for each variable used.

6.1. Overview

The concept view showed in a superficial way how the gossip and the data replication is done, but in this context where the same worker can be accessed at the same time by multiple operations, that can be the same operation run multiple times. As you can see, a gossip algorithm is not enough to prevent conflicts when accessing the storage or when trying to update the worker timestamp. To better illustrate our implementation, we are going to present you some issues that this system might have and how we dealt with them. Presenting this issues will also answer, we hope, most of the questions regarding the implementation.

6.2. Issues

Given that the GRPC calls are asynchronous there can be reads and writes simultaneous to a target storage regarding the same variable. To prevent this we use locks in all accesses to the valueTS, replicaTS and the data(volatile memory where the data is stored), Update Log and operationId(used to give the records an unique ID).

6.2.1 Mutliple Operator in the same Worker

But this is not the only problem, we know that a worker can be accessed by multiple operators and the worker timeStamp is a global variable, so multiple operators can access and modify that value. To resolve this we create a class MetaRecord and in the MetaRecord we have a field where we copy the workerTS to there. The MetaRecordTS is now the only one that is changed by the StorageProxy (a new one is created by the worker every time he receives an operator). So there is a SP to each operator runned in a worker. When the operator completes, the MetaRecordTS is merged with the workerTS(locked operation) making the worker always be the more updated possible.

6.2.2 Mutliple OperatorChains accessing the same variable

We know that the problem requests the preservation of causality between operators of the same chain. This could be easily done only using the timestamps, but the problem is that multiple chains of operators can be accessing the same variable stored. How to solve this? Besides the usage of locks to prevent conflicts on reads and writes, our storage memory also saves the 5 more current versions of a variable, making possible for at max 5 chains of operators run consecutively without reading each others values. But as you know, the user probably doesn't know/care if there are multiple chains of operators running, so they don't specify a version to read and only read the most recent version. To solve this the MetaRecord has a variable called **lastChanges**, those lastChanges are a dictionary specifying the last version read, of a specific variable, by that chain of operators. So, when reading the last version, the real version that its being read is the version that is given by the lastChanges.

6.2.3 UpdateIfValueIs total order and causality

As proposed the updateIfValueIs needed total order, in order to do so we wait for all gossip messages that are still in the network, and wait for all records regarding the target variable to be computed (emptying the date Log). But that brings a problem. What about multiple operator chains writing in the same variable, what happens in that case? Let's approach this respecting total order. In this case we would wait for all writes of all operator chains locking every write and updateIf that appeared after. But in this case the updateIfValueIs,if made by the operator A, has a chance to dont be applied over the variable that was written by the OperatorA but by OperatorB that wrote after OperatorA wrote. So in this case the UpdateIfValueIs can fail, and this is done on purpose. This doesnt mean that when the OperatorA will

try to read again he will read a random version, he will read the version that he wrote before the failed UpdateIfValueIs.

7. Conclusions

The system solved the problems proposed with success, with the usage of locks and gossip architecture we could prevent problems that couldn't let us solve the issue given. With that said we could say that our writes and reads are relatively fast, but the number of storages available increases the speed of the system, given that if there is only two storages and with the consistent hashing only one is target of all reads and writes, the system will not improve the speed.