Design and Implementation of Distributed Application: Project Report

Wallace Garbim
Instituto Superior Tecnico
wallace.garbim@tecnico.ulisboa.pt

Miguel Crespo
Instituto Superior Tecnico
miguel.crespo@tecnico.ulisboa.pt

Florian Ehrenstorfer
Instituto Superior Tecnico
fehrenstorfer@gmail.com

Abstract

In this paper we present a simplified function-as-a-service cloud platform, which enables the distributed execution of a chain of operators on a shared storage. The system consists of one scheduler and multiple worker and storage nodes. Workload is balanced equally between workers through the scheduler. Data consistency and fault tolerance are guaranteed by the storage nodes. We present the problem and motivation first, then describe our solution in detail, and evaluate it at last in this paper.

1. Introduction

Distributed function-as-a-service cloud platforms are increasing in popularity with big cloud providers like AWS and Google Cloud offering custom solutions. They offer great flexibility with of the shelf plug and play options as well as customizable and self-developed options. For this project we developed our own distributed function-as-a-service cloud platform, a simplified version of other platforms. We focused on its design, implementation and evaluation in the project in this paper.

2. Problem

The function-as-a-service cloud platform allows to run applications in a distributed network. Applications consist of a chain of operators, which can access a shared storage system. Because the storage system is distributed as well, one of the main problems is data consistency and fault tolerance. In order to maintain data consistency a gossip protocol is used to propagate updates through the network. The storage system supports three operations: read, write, and updateIfValueIs. Records are stored with the value and a version number. For the identification of the correct storage server to contact about a value, consistent hashing is used. For data consistency, operators have to read consistent version of the data, which has to be ensured by the system. Because the system also has to provide fault-tolerance, data has to be distributed between storage nodes, and reads have to be redirected. The different operators are assigned to workers by a scheduler, and then send from one worker to another after the operator has finished.

3. Solution

In the following section, we will describe our proposed solution.

Our system consists of five main processes: The scheduler, worker and storage nodes, and the Puppetmaster and PCS processes. The Puppetmaster is the main entry point of our system and controls the startup and workflow of the whole system. The PCS enables the Puppetmaster to startup the three different types of nodes remotely. Because all three nodes receive their configuration information as command line arguments at startup, the whole system could also be run without the Puppetmaster and PCS. The Puppetmaster can be run with single commands in the command line in a loop or by providing a full script, which will be executed by the Puppetmaster.
To run the Puppetmaster the following command has to be executed inside the Puppetmaster directory: *dotnet run*.
To run the Puppetmaster with a full script the following command has to be executed inside the Puppetmaster directory: *dotnet run* $<$ *scriptname*. scriptname has to include the path to the script.
The PCS process can be started with *dotnet run* inside the PCS directory. It only has to be started once to start multiple other nodes. All printed output from the started nodes will be printed in this console. The Puppetmaster

sends start up requests to the PCS, which then starts worker, storage or scheduler nodes as a new process. All other types of requests are also sent from the Puppetmaster to PCS, where they are distributed to the relevant nodes. In order to achieve this the PCS stores the information of all its created nodes.
The scheduler is responsible for the assignment of operators to workers and the distribution of workload between them. Workers receive the chain of operators and execute the operator that was assigned to them, before storing the output in the chain and sending the chain to the next worker in line. Since they only receive the name of the operator, workers select the operator with reflection. For that the operator .dll file has to be placed inside the *Operator* folder inside the workers directory. The workers get the url of all storage nodes and the scheduler from the meta data of the chain.

3.1 Scheduler Implementation

The scheduler is responsible for the assignment of operators to workers and the distribution of workload between them.

We first opted for a simple round-robin solution to the scheduling problem, meaning that the operators were assigned one by one, in a round-robin fashion. This was a simple solution at first, but with some big problems. The scheduler was assigning operators equally without respecting if the worker has already finished or was still working on operators. Because of that, workers that were still executing operators could get new operators assigned even though another worker was free. Because of this flaw, we decide to implement a better scheduling algoritmn.

On our second implementation, we decided to take into account the amount of work that each worker had done and assigned the next operator to the worker that had done the least amount of work. Because of this, workers send completion confirmations to the scheduler when they have finish working on one operator, and the scheduler keeps track of the amount of operators each worker is working on, resulting in a better load balancing between the available workers. Therefore the scheduler always assigns the next operator to the worker who currently works on the least amount of operators.

3.2 Storage Implementation Overview

The storage system stores pairs of keys and values, aswell as up to 10 previous versions of the same key. All the keys are replicated in all of the nodes. The storage supports three types of operations: Read, Write and Update If. The Write operations are propagated through a gossip protocol, while the Update If operations are executed in all nodes simultaneously using the 2-Phase-Commit protocol.

When an update is issued, it is assigned a Version Number and Replica Identifier, aswell as some other timestamps relevant for the correct functioning of the gossip protocol. In our implementation, every key has a replica timestamp and a value timestamp associated with it. The gossip protocol used was the one described on the course's book.

3.2.1 Read Operations

Read operations are treated as simple queries to the server. If a read operation is received with a null DIDAVersion, the most recent version is returned, otherwise, the request version is returned. If the version is not found in the server, an error is returned.

3.2.2 Write Operations

Write operations need to be propagated to the other nodes through the gossip protocol, but an order also needs to be maintained between them, so that Update If operations can be executed on the same record in every node. For updates that are causally constrained (i.e. sequential updates issued to the same replica), the gossip protocol already takes care of this issue. For concurrent updates (i.e. updates that are executed in different replicas at the same time), we opted for simply ordering the updates based on their version number and replica id. Therefore, the most recent update on a replica is the one with the highest version number and lowest replica ID. For example, if we have three replicas, R1, R2, R3 and they all write concurrently to the same key K, they all generate and update with the same version number, but with different replica IDs. The update issued by R1 is going to be the most recent, as it has the lowest replica id. When the system stabilizes, all the replicas agree on what is the next version number to be given to a new version of that key, so updates with a higher version number will be causally dependent on updates with lower version number.

A replica can't process a write and an Update If at the same time.

3.2.3 Update If Operations

Update If operations are applied to the whole system at once, through a 2 Phase Commit protocol.

On the first phase, when an Update If reaches a replica R, for a key K, it blocks any further updates to that key until the Update If is applied to all replicas. It sends a message to every storage node with the last version it has for key K and the new desired value. When the other replicas receive this message, they check if the version they have for that key matches the one that was sent. If the versions match, it sends an OK to the coordinator and appends the update

to a pending Update If list, waiting for confirmation from the coordinator. If the versions don't match, it sends back a NOK.

On the second phase, if the coordinator receives any NOK message, it cancels the transaction, sending an abort message to all the storage nodes involved. If all participants respond with an OK message, the coordinator sends a doCommit message to all the participants and they all execute the update. The update might still fail if the provided old value does not match the value present in the record.

If two Update Ifs are concurrent (i.e. they are issued on different replicas at the same time) the system will abort both Update If operations, for simplicity reasons.

3.2.4 Update Propagation

The gossip protocol used is very close to the one described in the course's book, so we aren't going to go into too much detail about it.

Once a write request reaches the server, it is assigned a version number, replica Identifier and Lamport Clock corresponding to the last update seen by the client that sent the update. It is also checked if the update has already been seen (through the unique identifier of the update, generated by the client).

If the update is stable, meaning that it doesn't depend on an update that the replica hasn't seen, it is executed immediately. This update is then inserted into a log, that is propagated to every node in the system.

Once a node receives a gossip message, it merges the incoming log with its own, filtering any repeated entries, and checks to see if any update on its log has become stable. If so, the update is executed. A table of executed updates is kept, in order to avoid executing duplicate updates.

The clients also keep a Lamport Clock of the latest update they have seen, so that they don't run the risk of reading inconsistent values from a replica that is less up to date.

3.2.5 Data Consistency

Data consistency between worker nodes executing the same client script is ensured by storing the last written or read version of a certain record in the DIDAMetaRecord and passing this information along the chain of workers.

3.2.6 Failure Detection

The system detects failures by pinging the storage nodes periodically. When a storage node does not respond within the stipulated time, it is assumed dead and the information is sent to all the other storage nodes. These then remove that node from their list of other known storage servers. The PCS also knows what nodes are dead or alive, so it won't send information about dead storages to the workers.

3.2.7 Fault Tolerance

For N storage nodes, the system support N-1 failures, as there only needs to be one storage server running to accept requests from any worker. If a replica fails before it can send any new updates that it might have gotten after it sent the last gossip message, these updates are lost.

4. Evaluation

In this section we will evaluate our solution with a baseline scenario, as well as best and worst cases.

4.1. Baseline

For the Baseline evaluation of our system we use a modified version of the Puppetmaster and operator script provided for the checkpoint evaluation. The Puppetmaster script can be found as *baseline* in the pmscripts folder. We run four storage nodes and two worker nodes. The used operators are the UpdateAndChainOperator and the AddOperator. We compare the performance of our final system with replication, data consistency and fault tolerance, consisting of four storage nodes with a version without replication of data with only one storage node. For that we time how long the whole application runs, before returning to the Puppetmaster. Because the script includes waits to ensure the correct startup of all nodes, the values should only be used for comparison between the two systems. The full execution on the system without data replication took 3369 milliseconds. The full execution of the system with four storage node and data replication took 3407 milliseconds. From that we can see that the replicated system is slightly slower than a non replicated system, but there is only a small difference.

4.2. Worst Case Scenario

The worst case that can happen is a storage node crashing during the 2-phase commit. To evaluate we will run the same script for two cases: one case without a crash and one where the storage node that receives the update crashes during the 2-phase commit. In order to ensure that the node crashes during the 2-phase commit we introduce a sufficient gossip delay, which covers the time that it takes for the crash command to reach the node. The full execution of the system without a crash takes 1249 milliseconds. With a storage node crashing the system operates for 1519 milliseconds. That indicates a 17.8% increase in runtime in the event of a crash of a storage node.

4.3. Best Case Scenario

For our system the best case scenario consists of operators that all roughly operate for the same time and no failure

of storage nodes. Because the scheduler assigns operators to workers with the fewest active ones, if there is a big difference between the run time of operators, the scheduling can operate inefficient. One example of a best case is the execution of the checkpoint files. Here the full execution takes in total 2216 milliseconds. The execution time represents the time from when the scheduler executes the first client command to the completion and confirmation of the last operator.

5. Further Work

There are a lot of aspects that could have been bettered in our solution, but due to lack of time we didn't implement these solutions.

5.1 Update If operations

In our solution, it would would have been better if two replicas that disagreed with the latest version of a key could communicate with eachother and try to reach a state where the replica that was lagging behind could update itself and not cancel the entire transaction. The already existing replica Logs used on the gossip protocol could have been exchanged between the nodes, so that missing updates could be shared and even speedup the propagation speed of said updates.

Other improvement could consist in, instead of aborting two concurrent update if operations, to establish an order between them and execute them in that order.

5.2 Partial Replication

We did not implement partial replication on our solution, but it could easily be implemented by dividing the pool of available storage nodes into a set of groups and limiting the nodes known by any node to the ones on its group.

6. Conclusion

In this paper we presented a distributed function-as-a-service system that consists of scheduler, worker, and storage nodes. The scheduler dynamically distributes the workload between workers, and the storage nodes replicate data and provide data consistency between them. The evaluation showed that our system works, while also discussing its shortcomings.