Design and Implementation of Distributed Applications 2021-2022

Ricardo Caetano
Instituto Superior Técnico
Student Number: 87699
ricardo.caetano.aleixo@tecnico.ulisboa.pt

Samuel Vicente
Instituto Superior Técnico
Student Number: 87704
samuel.vicente@tecnico.ulisboa.pt

Vasco Faria
Instituto Superior Técnico
Student Number: 89559
vasco.faria@tecnico.ulisboa.pt

Abstract

The goal of this project is to design, implement and evaluate a simplified version of a distributed function-as-a-service cloud platform. This platform will run application composed of a chain of custom operators that share data via a storage system. The platform is composed of a PuppetMaster, that will be our client, a Scheduler, Process Creation Service (PCS), Worker Nodes and Storage Nodes.

The main goals of this project are to design a system that is partially replicated, where replicas are updated via gossip and the selection of the Storages Nodes for which the key is stored is computed using a Consistent Hashing algorithm, and fault tolerant, where we can tolerate N/2 faults where N is the number of Storage Nodes.

1. Introduction

Function-as-a-service (FaaS) offers a platform that enables clients to construct and run application functions without the difficulty of establishing and maintaining the infrastructure that is traditionally involved with designing and launching an app.

On our FaaS system the operator is the unit of work in our system. The application that runs on our platform is an ordered collection of operators that are chained together to achieve the desired output. When a user wishes to start an application, he must supply the platform with an application file, where each line represents the operator that should be executed, in the order they should execute. The code for these operators will be supplied by the user through a DLL that will be loaded on the worker node as needed.

The storage infrastructure that allows data to be passed between operators and applications is distributed and partially replicated. The DIDARecord is the data unit that we store. The DIDARecord is made up of one key that identifies the record and a set of up to five versions of that record. To accomplish partial replication, we must split the data consistently across the Storage Nodes and sync them to achieve fault tolerance. We chose an upgraded version of Consistent Hashing [2] since we could presume that the Storage nodes would only fail and that new or failed ones would not join. We used the Gossip protocol [1] to prevent data loss in the event of a crash.

```
1 DIDARecord Read(string id, DIDAVersion
      version);
2 DIDAVersion Write(string id, string val);
3 DIDAVersion UpdateIfValueIs(string id, string
       oldvalue, string newvalue);
```

Listing 1: Storage API

This system exposes an API, Listing 1, to the operators. The available operations are the read, write and conditional update (UpdateIfValueIs).

The operators are distributed throughout the worker nodes by the scheduler. The Scheduler distributes the operators through the workers in a Round Robin fashion.

Some assumptions are made in order for the applications to perform properly. If an operator reads or writes a key, all future operators in the program will read the version that was read or written previously. If the required record has been garbage collected (due to the restricted number of versions saved), the application should be terminated.

2. Design

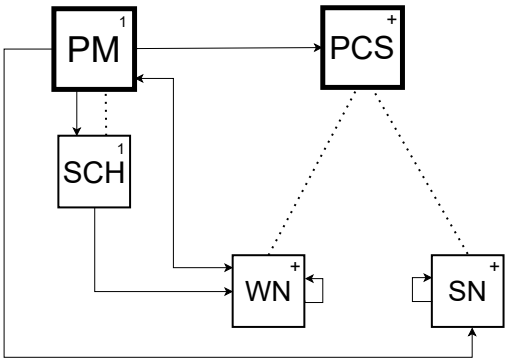

Figure 1: Architecture of DIDA2020/21

A bold box denotes the components that must be manually initiated. Each component box has a sign denoting the number of copies, with 1 indicating that they are one-of-a-kind and + indicating that there are at least one. The arrows indicate communications, with the arrow pointing to the gRPC Server. The dotted lines indicate the link between the parent and child processes.

2.1. Puppet Master

The PuppetMaster will act as a client, allowing us to control the system through him. PuppetMaster will contact the Process Creation Services (PCS), the Scheduler (SCH), Storage Nodes (SN), and Worker Nodes (WN) via gRPC.

The first thing that the PuppetMaster must do is to create the Scheduler, so PuppetMaster will create a new Process with two arguments the Scheduler Id and the URL, we use the Id to identify all of our processes and the URL is stored in the PuppetMaster so it can contact later when we want to run applications.

The second thing that the PuppetMaster must do is to create all Storage Nodes, the PuppetMaster will contact the PCS of the pretended host of the Storage Node, and send three arguments, the Storage Node server Id, the Storage Node's URL and his gossip delay in milliseconds.

Then we can proceed to create all the Worker Nodes, this process is equal to the creation of the Storage Nodes, the Puppet Master will contact the PCS of the pretended Worker Node and then the PCS will be in charge of the creation of the Worker.

After the creation of our infrastructure (Scheduler, Storage Nodes and Worker Nodes), the PuppetMaster can now give commands for the system.

2.2. Process Creation Service

The Process Creation Service (PCS) is in charge of the creation of the Worker Nodes and Storage Nodes for one Host. So we will have N PCSs, one for each host machine. When receiving a creation rpc call from the PuppterMaster it will proceed to create a child process with the same arguments as it received in the rpc call. PCS is also responsible for killing the process that created, so we decided to create a Dictionary where the Key is the Server Id and the Value is the Process created, this makes it easier to kill the process. In our system the only servers that can crash are the storage nodes, but we keep all the processes information in order to terminate everything at the same time with the exit command.

2.3. Scheduler

Unlike the PCS where we have N PCSs, one for each host, we only have one Scheduler. The Scheduler is responsible to create a sequential chain of workers, one for each operator that the user want to run in a application.

The way the Scheduler decides which Worker will run which Operator is Round Robin fashion. So, if we have two workers and four Operators, at first the Scheduler starts by shuffling the list of the workers, and then it will assign the first Operator to the first Worker of the shuffled list, the second Operator to the second Worker, the third Operator to the first Worker and the fourth Operator to the second Worker.

For instance if we have two Workers and four Operators our Scheduler will assign the first Operator to the fist Worker, the second Operator do the second Worker, the third Operator to the first Worker and the fourth Operator to the second Worker.

2.4. Storage Node

```
1  struct StorageNodeStruct
2  {
3   string serverId
4   string url
5   int replicaId
6   bool isDown
7   GrpcChannel channel
8   GossipServiceClient gossipClient
9   UpdateIfValueIsServiceClient uiviClient
10 }
```

Listing 2: StorageNodeStruct

The Storage Node will be created by the host PCS and will receive four arguments upon creation:

- Storage Server Id;
- URL;
- Gossip Delay;
- Debug Flag;
- Replica Id;

Storage Nodes have a Dictionary called storageNodes, where we store all other Storage Nodes in the system, the key is the server id, and the value is the structure of Listing 2.

This Dictionary allow us to know relevant information of other Storage Nodes that we will try to connect when we try to do operations like updateIfValueIs or when we run the Gossip Algorithm.

2.5. Worker Node

The Worker Node will be created by the host PCS and will receive five arguments:

- Worker Server Id;
- URL;
- Delay (when passing to other nodes);
- Debug Flag;
- Log Server URL;

Like the Storage Nodes, Worker Nodes have the information of all Storage Nodes, we also have have a Storage Proxy that is used to connect the Operator to the Worker Node so it can make rpc calls to the various Storages Nodes.

2.6. Operator

This is the code that runs in the Worker Node. The user must supply the platform with the DLL containing the Operators. To simplify, in this project, this DLL is loaded from memory rather than being provided from the client in the UIs. These Operators are then loaded in the Worker Nodes through reflection. When the Operator is loaded, a Storage Proxy is supplied to it in order to have access to the Storage System.

3. Partial Data Replication

In this Section we will describe more in detail the two algorithms that we used in our system, Consistent Hashing and Gossip in regards of Partial Data Replication. The system has a replication factor of $(N/2) + 1$ to tolerate $N/2$ storage crashes.

3.1. Consistent Hashing

Consistent Hashing is a technique that uses hash functions to distribute the requests over the system nodes.

The original algorithm allows the system to append or drop nodes in run time, but, because new nodes are not allowed, in this project, and the system will have a relatively small number of storage nodes we made a variation of the algorithm.

First, since we know, in the beginning, all of the storage nodes, we divide the ring in a way that all the nodes have the same amount of space within the ring, we opted for a ring size of 256 positions. So, instead of using an hash function to calculate the nodes positions we instead multiply the replicaId by the size of a ring slice ($RingSize/StorageNodesNumber$). To make the nodes' sticking to the same position we always need to use the number of storage nodes in the beginning of the execution.

As the number of nodes for this system will be relatively small in relation to some uses of the Consistent Hashing algorithm, our variation is a one hop solution, so all the nodes can reach each other just by computing each other positions.

To calculate which replicas will store each DIDARecord, we start by computing the hash of the respective record, here we used the SHA256 hash function and then we extract only the last byte. By extracting this last byte we end up calculating the modulo of 256, which is our ring size. Then, we search the $(N/2) + 1$ (Replication Factor) storage nodes with the position in the ring next to the key hash value and put them in a set.

To prevent a heat replica in the ring, we distribute the load of the requests by shuffling the possible set of replicas, so when a client tries to contact the replicas from the first to the last, the set order is always different.

When a replica fails, we don't re-balance the ring, so the set of possible replicas includes a replica down, and so we can support $N/2$ fails, one less then the set size.

3.2. Gossip

When there is an operation in a Storage server that changes his state, we start a timer that will run gossip every gossipDelay milliseconds. Every Storage Node has a Replica Manager, this class is used to store a replicaTimestamp and a timeStampTable. The replicaTimestamp is a dictionary where the key is the id of the DIDARecords and the value is a list of DIDAVersions. The timeStampTable is a dictionary where the key is the serverId of all other Storage Nodes and the value is a dictionary where the key is the id of the DIDARecords and the value is a list of DIDAVersion, this last dictionary corresponds to the replicaTimeStamp of each server. We use the DIDAVersion as a logical clock and the stored DIDARecords as the updateLog messages. The replicaTimeStamp is used for each Storage Node to track the versions of each key, and the timeStampTable to track the versions that all other Storage Nodes have.

When the timer triggers a gossip, the replica will compare every version in its replicaTimestamp to every timestamp of all the nodes in the timeStampTable in order to check if the other server is missing any DIDAVersion.

If a replica has missing versions for a specific id, in comparison with the ones that the Storage Node have, it's possible that that replica is not been updated and so it will search for the DIDARecords that corresponds to that DIDAVersions and send them to that replica along with the Storage Node timestamp.

The Storage Node that receives a GossipRequest will check if it already has the DIDARecords that the other Storage sent, if the Storage Node does not have that DIDARecord it will add to the storage, and will update his replicaTimeStamp, and send his replicaTimeStamp as a GossipReply.

When a Storage Node receives a GossipReply with a timestamp of a replica, it will replace that replica's timestamp.

4. Operations

The Storage API, Listing 1, offers three operations: Read, Write and UpdateIfValueIs. In this section we will explain in detail our implementation of these operations.

4.1. Read

The Read storage operation receives the id of the DIDARecord we want to read and, optionally, we can also specify a version, depending if we want to read a consistent version or the most recent version (in the replica it accesses).

In order to make this operation request, the operator has to contact the Storage Proxy, which starts by calculating the set of replicas that contain the given id, then checks if there is a specified version in MetaRecord to ensure that the read is consistent with the previous operations in the application chain, and then tries to read from the first replica of the given set of replicas.

If this replica is down or doesn't have the requested record (or version), the proxy will automatically try the next one until one of the replicas can handle the request.

If the requested version is not available in any of the replicas, then the application is aborted.

4.2. Write

The write storage operation receives as arguments the id of the DIDARecord we want to create/update and the respective new value. This operation starts by checking if the requested id is not being used by any updateIfValueIs request.

If it is being used, then it is returned a version with both versionNumber and replicaId with a value of -2 meaning that the requested id is locked. If not, the storage starts by searching the greater version for that id in order generate the next version for that record, if it doesn't have any version for that id, it creates a new storage entry and the respective timestamp.

When the maximum number of versions for that id is reached, the storage removes the oldest version in order to save the new one.

To call this operation, the operator also has to contact the Storage Proxy, that starts by calculating the set of replicas that contain the given id and tries to connect to the first replica of the calculated set.

If the replica replies with already locked, the proxy sleeps for a random time, to try again later until the replica return a different output. If that output is null then the proxy tries to connect with the next replica of the set, otherwise the write was successful and the proxy updates the MetaRecord in the ProcessOperator Request.

4.3. UpdateIfValueIs

The updateIfValueIs storage operation receives the id of the DIDARecord, a string with the old value and string with the new value. Our protocol for the UpdateIfValueIs uses a similar approach compare to a two phase commit. We can divide the protocol in two phases the first being LockAndPull Phase and the final one being the CommitPhase.

- LockAndPull: In this first phase of the UpdateIfValueIs the storage node that received the operation contacts all the Storage Nodes that can store that key, which is computed via the consistent hashing algorithm, asking for the last version of the record stored of that key. The replicas that receive the rpc call of the LockAndPull Phase will first check if the key is locked via the leaseKeyLock dictionary, if the key is locked the replica will send a LockAndPullReply with the field AlreadyLocked set to true, the Storage Node that was responsible of making the UpdateIfValueIs send a Version with values replicaId = -2 and versionNumber = -2 to the Storage Proxy that made the rpc call so the Storage Proxy can wait a random amount of seconds, between 0 and 5, to try again, otherwise, if the key is not locked the storage node promises to lock that key (lease) for 5 seconds. If all of the Storage Nodes replied with their last Version of the DIDARecord, we can proceed to the CommitPhase with a commit message. If any of the requested replicas crashes, takes too long to reply, or replies with already locked we also proceed to the CommitPhase with an abort message.

- CommitPhase: From all the DIDARecords received

we first calculate the most recent record, meaning the DIDARecord with the most recent DIDAVersion, we calculate the most recent DIDAVersion with the field versionNumber. If there are two DIDARecords with the same versionNumber, the most recent will be the one which has the higher replicaId. If the oldValue is the same as the value of the most recent version, then the Storage Node will create a new DIDARecord with the desired value and the newest version and broadcast it along with the commit message, otherwise, the Storage will broadcast an abort message.

- Lease Timeout: After the lease timeout, if a replica has not yet received the commit/abort request it unlocks the key in order to avoid deadlocks, in the case the other replica has crashed.

When dealing with the dictionary where we store all keys and the list of DIDARecords we lock the key of the pretended DIDARecord in all of the possible Storages so the UpdateIfValueIs can proceed with no concurrent writes or other UpdateIfValueIs. If an Operator want to do an UpdateIfValueIs but that key is locked, the Storage Node will tell the Worker sending a Version with the versionNumber = -2 and a replicaId = -2 so the worker can try again.

5. Evaluation

In order to evaluate the system, we prepared some scenarios to test the main properties of our solution, to which our project produced the desired outputs and tolerated the faults when crashing servers.

5.1. Consistency Use Case

In this scenario, the goal is to evaluate if the system supports data consistency, where a client always has to read consistent versions. To test this, we use two clients, one of them does two increments separated by a 5 seconds sleep while the other makes a number of increments lesser than the Max Versions (5). So, when the sleep operator is over, the first client has to read a consistent version related to the version accessed in the first operator.

5.2. Max Versions Use Case

This experiment is very close to the previous one, except for the number of increments the second client does, which is bigger than the Max Versions. By doing this, when the sleep time of the first client is over, the version he wants to access is no longer accessible and so the application is aborted.

5.3. Failure Tolerance Use Case

To test that the system tolerates $(N/2) + 1$ failures, we instantiate 10 storage nodes, and while two clients are using the system we gradually start crashing 5 Storage Nodes, half of the nodes that were initialised.

5.4. UpdateIfValueIs Tiebreak Use Case

This scenario aims to test when two, or more, updateIfValueIs operations happen at the same time, trying to update the same key. So, we made a login usecase, where each user can only login once, by making a updateIfValueIs request to the key ¡username¿, from the old value "out" to the new value "in". So, in the experiment, we try to login twice with "Bob", and what happen is that when two clients try to login as "Bob" at the same time, both requests are suspended for a random tiebreak timeout until one manage to complete the operation, making only possible for one of the users to login, while the other application is aborted.

5.5. Load Test Use Case

The goal of this test is to see how the system reacts with a lot of updateIfValueIs requests, so we created 5 clients, each one making 10 updatesIfValueIs operations in a row, leading to a total of 50 operations and the system finish as supposed with the last 5 version of the 50 updateIfValueIs operations.

6. Further Improvements

To improve even further our system we could change would be to when a replica node crashes the gossip would select a new storage node for those keys maintaining always a replication factor of N/2 + 1 where N would be the (Storage Nodes - Storages Nodes Crashed), this would make our system resilient to N - 1 faults.

7. Conclusion

To conclude, our project accomplishes the stated goal of being fault tolerant up to N/2 storage failures and is partially replicated utilising the two algorithms, consistent hashing and gossip. Where the consistent hashing provides a set of replicas to where a specific key can be stored and the gossip algorithm provides the missing updates to all the keys for the right replicas providing data consistency between systems' Storage Nodes.

References

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. International computer science series. Addison-Wesley, 2012.

[2] S. Neelakantam. How we efficiently implemented consistent hashing, Dec 2020. https://ably.com/blog/implementing-efficient-consistent-hashing.