ID2203 Final Report

Àlex Costa Sánchez
alexcs@kth.se

1 Introduction

The goal of the project is to implement and test a distributed in-memory key-value store with linearisable operation semantics. The operations to be suppported are `Put(key)`, `Get(key, value)` and `CAS(ley, referenceValue, newValue)` (which updates a value is the `key` and the current value match).

This project has been implemented in Scala**??**, using the Kompics programming model. Kompics is a programming model for distributed systems that implements protocols as event-driven components connected by channels.

For this implementation we assume a partially synchronous system, considered useful for the Internet, where consensus is solvable with up to $N/2$ crashes. Adding an Eventually Failure Detector ($\diamond P$) or its equivalent Eventual Leader Election (Ω), and assuming from now on perfect links, we have a Fail-noisy model.

2 Infrastructure

2.1 Overview

As mentioned in the introduction, this project is implemented using the Kompics programming model. According to its documentation[1], "Kompics provides a form of type system for events, where every component declares its required and provided ports, which in turn define which event-types may travel along their channels and in which direction. The channels themselves provide first-in-first-out (FIFO) order exactly-once (per receiver) delivery and events are queued up at the receiving ports until the component is scheduled to execute them."

The skeleton code provided includes most of the structure. The client has already implemented the `Get(value)` operation, and has the structure for `Put(key, value)` and `CAS(key, oldValue, newValue)`. The implementation of these two operations from the client side is be fairly trivial; this report focuses on the server side of the project.

In the server, the component structure is depicted in figure 1. When the server is started, a `Host` component is created, which creates a `Parent` component. Then, the `Parent` component creates the rest and connects them

accordingly. Note that the `Routing` port connection is dashed as it is not used in this implementation of the project. The `Bootstrap` component is triggered and boots the system. When the replication threshold is reached, the `VSOverlayManager` is triggered and creates a lookup table with all the nodes (would not be all the nodes if the system was partitioned). This information is sent to teh `BallotLeaderElection` component, which starts selecting a leader, and to the `SequencePaxos`. The last component is the `KVService`, that effectively implements the Key-value storage. All the components are explained in more depth in the following section.

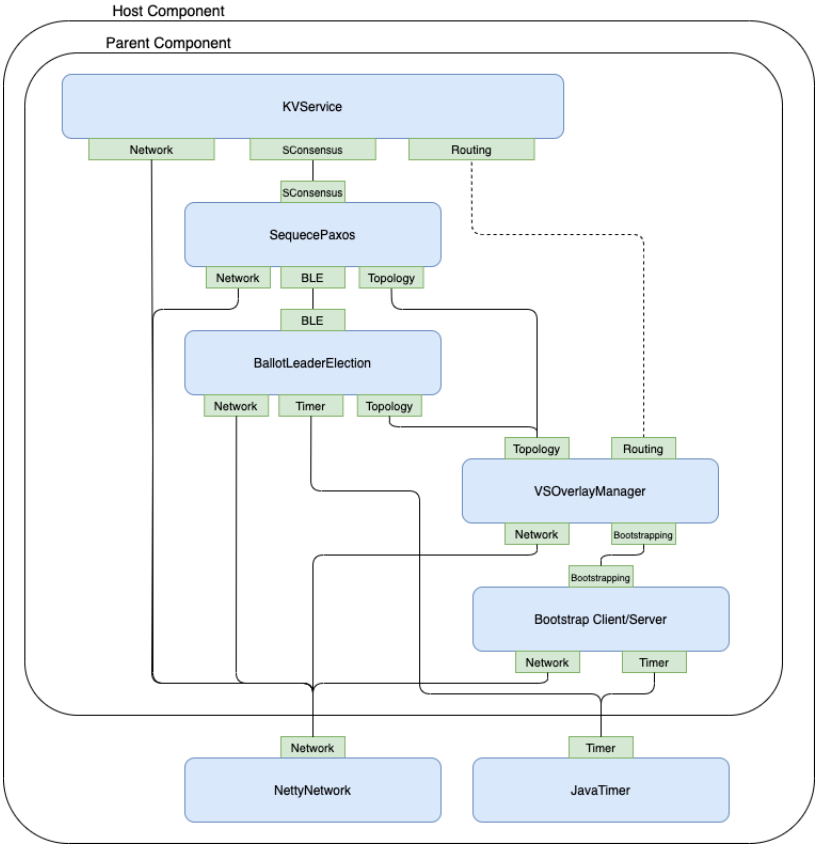

Figure 1: Component structure in the server side.

Once the system is up and running, a client can connect to it and issue an operation. Since the client does not know where to send the command, it is sent to the bootstrap address. The bootstrap server broadcasts it to every node, and then the command is handled by the `KVService` component. The `KVService` component will propose the command to the `SequencePaxos`, execute decided commands and sends a response back to the client.

2.2 Components

KVService

The KVservice component is the main component in the structure, which effectively implements the in-memory key-value store, in the form of a dictionary. It handles operations (`get(value)`, `put(key, value)` and `cas(key, oldValue, newValue)` from clients and uses sequence consensus to perform this operations in a reliable manner. This is why is requires a `SequenceConsensus` port. It also requires a `Network` port to send responses to the clients, and a `Routing` one - which would only be needed if we were to partition the key space (more on this in section 6.2)

VSOverlayManager

Since in this project we will only have one partition, this component is a bit of an overkill. As provided, it will basically broadcast messages from clients to all replicas of the Replicated State Machine (RSM). It requires a `Bootstrapping` port to boot. It provides a `Routing` port (not needed for this project) and a `TopologyProvider` port. Once the system is up and running (the `VSOverlayManager` has received a `Booted` event from the `Bootstrap` component), it sends the topology through this port to the `SequencePaxos` and the `BallotLeaderElection` components.

Bootstrap

This component initializes the replicas in the RSM. Provides the `Bootstrapping` port for the `VSOverlayManager`. It requires a `Timer` and the `Network`.

SequencePaxos

To implement the Replicated State Machine, we need consensus; to be more precise, sequence consensus. This will he achieved by this component, implementing a Leader-Based Sequence Consensus (more on this in section 3.2). It requires a `BallotLeaderElection` port to elect a leader, `TopologyProvider` to learn about the topology, and `Network`. It provides a `SequenceConsensus` port.

BallotLeaderElection

Leader-based consensus relies on a Ballot Leader Election, implemented by this component; concretely it implements the Gossip Leader Election algorithm. It provides a `BallotLeaderElection` port and requires `Timer`, `Network` and `TopologyProvider`.

2.3 Replication and partitioning

As previously mentioned, this implementation of the distributed key-value store is replicated. The replication algorithm and other considerations are further

explained in section 3. The replication degree (δ) can be specified in the configuration files of the project.

On the other hand, the key space is not partitioned. It is one of the proposed extensions to the project in section 6.2.

3 Replication Protocol

3.1 Consensus

To achieved the desired replicatoin, we need consensus. In consensus, processes propose values and they all have to agree on one of these values. Paxos is arguably the most important algorithm in distributed computing as it solves the single value consensus problem efficiently in a fail-noisy model like ours. This project implements a slightly modification of this algorithm, the Leader-based Sequence Paxos. The following explanations are based on the lecture notes[2].

3.2 Replicated State Machine (RSM) and Leader-based Sequence Consensus

What we want to achieve are Replicated State Machines (RSM). RSMs need to agree on the sequence of commands to execute, execute them to transform its state and possibly produce some output. Since commands are deterministic, we ensure to have the same state and output in a majority of machines.

In order to agree on a growing sequence of commands, we need Sequence Consensus. Leader-based Sequence Paxos is an efficient modification of the Paxos algorithm to agree on growing sequences of commands, given that the processes rarely fail (long-lasting leaders).

3.3 Assumptions

We need to make to following assumptions for the algorithm to function properly:

- We have Replicated State Machines - each process acts in all roles: proposer, acceptor and learner.
- The links in the system are FIFO Perfect Links for commands to be acceptes incrementally.
- The leader will run for a long period of time. Leader-based Sequence Paxos ensures safety if not, but it is optimized for long-lasting leaders.

3.4 Linearizability

Linearizability or atomic consistency is a property that makes a system appear as if there was only one copy of data and all operations on it were atomic

(and happen at some point between its invocation and response). Sequence Consensus provides this property form a theoretical point of view (we will also test it in practice, see section 4).

4 Testing

4.1 Operations testing

Basic tests for the `Put(key)`, `Get(key, value)` and `CAS(ley, referenceValue, newValue)` operations have been implemented and passed.

4.2 Linearizability testing

Arguably the most challenging part of the project has been to test linearizability of the key-value storage system. To do so, we have used an algorithm proposed in the paper *Testing for Linearizability* (Lowe, G. (2017))[3].

The algorithm has been successfully implemented and linearizability tested for a single writer. Section 6.2 discusses how testing can be further improved.

5 Code

5.1 Repository

The code of the project can be found in the following link:

`https://gits-15.sys.kth.se/alexcs/id2203project21`

5.2 Running the code

In order to run the project, first the bootstrap server needs to be started as follows:

```
java -jar server/target/scala-2.13/server.jar -p 45678
```

After the bootstrap server is started on `<bsip>:<bsport>`, start the rest of the servers by:

```
java -jar server/target/scala-2.13/server.jar -p 45679 -s <bsip>:<bsport>
```

By default you need δ nodes (including the bootstrap server), before the system will actually generate a lookup table and allow you to interact with it. This parameter can be configured in the configuration file.

To start a client (after the cluster is properly running) execute:

```
java -jar client/target/scala-2.13/client.jar -p 56787 -s <bsip>:<bsport>
```

This process can be simplified by executing the following scripts:

```
./cluster_setup.sh $delta
./client.sh
```

6 Conclusions

6.1 Summary

All the required sections of the project have been fully implemented and tested, including the bonus part (`CAS(key, refValue, value)` operation). This report discusses the main theoretical concepts that are relevant for the implementation of the project (such as consensus, linearizability or replication, among others). It also discusses the main challenges of the implementation itself (mainly, components specification and linearizability testing).

6.2 Future work

The section of the project with room for improvement are the testing scenarios. linearizabililty has been tested with only one client. It would obviously be interesting to improve the testing scenarios to multiple clients to ensure this property in more complex situation.

There are two main features that can be added in future versions of this project. The first one is partitioning the key space. In real-world databases, where the amount of information can be enormous is it useful to have it partitioned - divided in different servers according to their key. A second feature to be added is reconfiguration - an extension of the Sequence Paxos algorithm that allows replacing faulty processes.

References

[1] Welcome to Kompics's documentation! — Kompics 1.2.1 documentation. `https://kompics.github.io/docs/current/index.html`.

[2] Seif Haridi, Lars Kroll, and Paris Carbone. Lecture notes on leader-based sequence paxos – an understandable sequence consensus algorithm, 2020.

[3] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4):e3928, 2017. e3928 cpe.3928.