KTH Royal Institute of Technology

Distributed Systems
Kompics Report

Félix Fonteneau

March 24, 2021

Contents

1 Introduction

In this document, I will approach my reasoning and my implementation about the conception of a distributed KV-Store with linearizable operation semantics. This project is left with a certain freedom of action, so I will present some of my choices and arguments supporting them.

1.1 Generality

The idea behind this project is to create a Key-Value Store distributed on several servers (nodes). The general operation of this architecture is that a user will send some basic requests to the system, like `write(key="aa", value="Test")` or `read(key="aa")`. Then inside the system, the nodes will communicate to find the information or to append it to the storage. The system will store value and key as String to be simple and clear for tests, but it could be any type of serializable data.

This system is distributed and therefore is more complex than a centralized one, we will have to prevent different types of problems like crashes or connection of new node, but also concurrency and linearizability between the nodes.

GitHub link for the code.

2 Infrastructure

2.1 General Infrastructure

First, in a bigger picture, the system will be able to answer a client from any node. In other words, every server is capable to be an entry point for clients (see Figure 1). This property is interesting because it reduces the bottleneck of having only one entry point if a lot of requests happened.

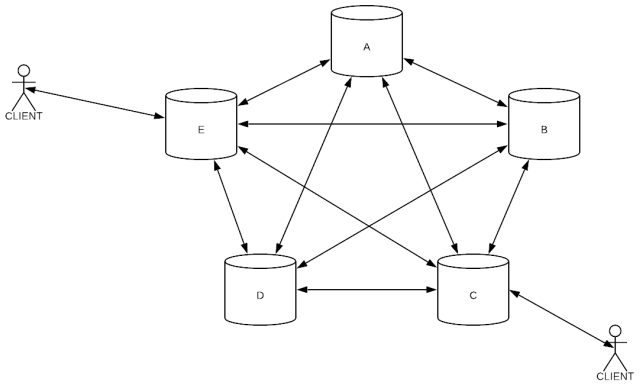

Figure 1: System Abstraction with 5 nodes

Then the memory is shared among nodes. In this storage there is only one unique partition, all the nodes replicate the same state of memory. Each server will save the tuples key-value registered with a hash map in memory but can easily be persistable. Note that the tuples stored are equals among the nodes, but the real state of each hash map can vary regarding the implementation of the hashing algorithms.

2.2 Sequence Flow

The system works in the following manner.

- First, we start the first node in bootstrap server mode. This first server will help to link the node to create the system.
- Then, the servers start and connect to each other via the bootstrap server. After completing this part, the system is fully operational and ready to accept clients.
- The client connects to a node of the system to access the storage.
- The client sends a request to the system.
- The node receive the operation to execute and send several messages to all nodes to agree on a consensus of execution.

- At the end of the consensus, every node executes the operation in the same order.
- The node reply to the client of the result.

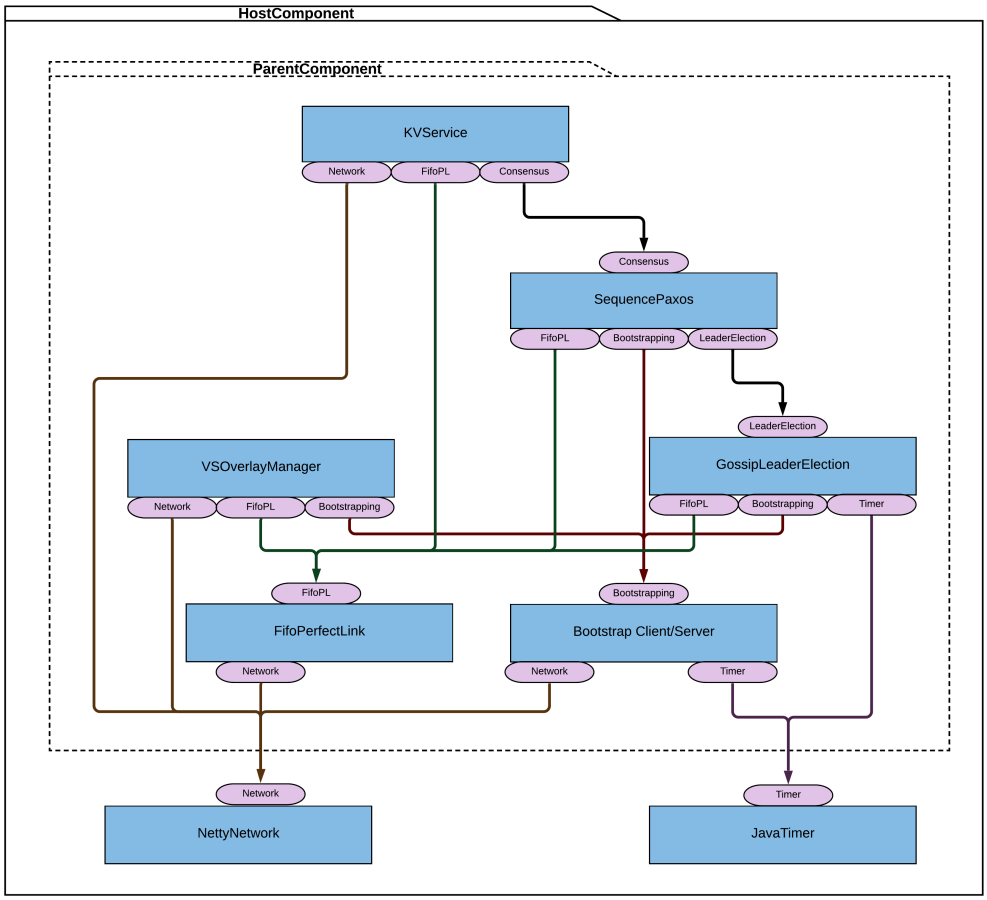

Figure 2: Layer Architecture of a Node

2.3 Networking and Communication Protocols

This system demands different protocols or channels of communication.

- In the first place, a protocol is used to connect the nodes together, with the help of the Bootstrapping component.
- Then, we have the communication between the clients and the nodes done by either the OverlayManager to receive messages/commands of the client, then the KVService will answer the request to the client.
- A channel is used between the nodes to elect one leader via a gossip ballot leader election.
- Finally, a protocol is used to create a consensus among the nodes regarding the order of execution of each request in the data structure. This is done via the *SequencePaxos* component.

Note that some networking protocols use a FIFO perfect peer to peer link, it is the case of the consensus and the ballot leader election.

3 Replication Protocol

In order to respect the first specification, I choose to use a leader based sequence Paxos, for the replication design. This algorithm has properties that are interesting for our case, it allows us to replicate the very same state on every servers despite the proposing requests distributed on different nodes in a finite time.

3.1 Leader ballot election

With this intention, we need a leader election algorithm to run a correct version of the Leader Based Sequence Paxos. The leader ballot election will guarantee a failure detection and the switching of the leader. Without a failing node, the system will maintain itself but will not handle a complete recovery scenario if the node comes back later.

3.2 Leader based sequence Paxos

In the first place, a sequence Paxos allows us to guarantee the same execution of several requests in the same order. The sequencing shared by the nodes is the key to linearizability. However, the simple sequence Paxos algorithm is not suitable for correct distributed storage. In fact, the sequence Paxos does not promise any progress, and the choosing of a sequence can fail with two nodes proposing and discarding the sequence of the other in parallel. The use of the leader based sequence Paxos fixes this problem by locking the proposition of sequences to a single leader. There is no starvation since the leader is not static and changes with the ballot election.

Furthermore, the sequencing of the operations to execute on every node allows us to keep the linearizability property with the Compare and Swap operation.

4 Testing

The testing of the system was an important task in this project. Several types of test scenarios were developed.

4.1 Basic operation tests

One first type of test consists of checking the result validity of the basic operations. A scenario just pops one client that executes basic requests and sees the result.

4.2 Simple Linearizability tests

Then a second scenario, more complex tests the linearizability of several clients sending random requests to a single node of a 5 servers system. In order to achieve that, a component is deployed in parallel to the execution to record every event, the sending of a request, and the answer of a request, the *ScenarioHistory*.

Then, after the end of the scenario, the component sends the data structure of the history of every event dated to be analyzed. The analysis of the linearizability of the history is done with the *Wing Gong linearizability algorithm*[1].

4.3 Advanced Linearizability tests

After testing the behavior and linearizability of several clients requesting on a single node of a system, a more advanced test plan came to test the reliability of the linearizability with more risky scenarios.

The scenarios consist of having a five servers system with several clients requesting different nodes. Then we kill several nodes of the system and start other clients. The client request random operations (GET, PUT, CAS). The complete history is still recorded with the history component, and we do the analysis the same way as before.

[1] https://www.cs.ox.ac.uk/people/gavin.lowe/LinearizabiltyTesting/paper.pdf, page 4