Distributed Key-Value Store

Distributed Systems - Advanced, ID2203

Bernardo Gonzalez Riede & Vinit Kumar Verma

March 11, 2018

1 Introduction

1.1 Goals

The overall goal is to have a distributed, partitioned and in-memory key-value store. While not explicitly mentioned, the abstractions used throughout the course and the book are fundamental for planning and reasoning about the system development. Imbuing them and relying on underneath abstractions might reduce complexity considerably while working on a certain module/layer.

1.2 Requirements

The following points had to be covered:

- Linearizable operation semantics
- Development of test scenarios
- *GET*, *PUT* and *CAS* operations

2 Reasoning & Motivation

2.1 Distributed system model assumption

The model assumed is the fail-noisy one. The following sections will motivate the decision made.

2.1.1 Process/crash abstraction

Byzantine failures won't be covered. While a reality, it introduces a significant amount of complexity. The probability of a malicious failure during the development is low enough to be negligible. Arbitrary faults in the form of corrupted messages will be prevented by using TCP/IP. Moreover, being a memory key value store one could argue about

the usage of ECC buffered RAM which corrects corrupt memory, which itself is already an unlikely one-time failure. If a node crashes, it won't be restarted itself which makes omissions the only reason for using the crash-recovery model. Omissions may incur in the following way:

- A node is overloaded
- A message is lost in transit
- Network partition

A node overload can be prevented by vertical scaling while the perfect link abstraction assures exactly-once delivery. For this project it's feasible to assume that a high amount of crashes is unlikely, meaning a majority of correct nodes exists. Therefore the crash-stop model will be used.

2.1.2 Communication assumptions

Being the abstraction mostly used during the course and guaranteeing exactly-once delivery, perfect links will be used for communication.

2.1.3 Timing assumptions

Running the nodes on the same machine can allow for a synchronous system, although a growing key value store might pose a problem in terms of computation time. On the other hand, an asynchronous system is very weak and there exists a stronger model with real world value. The partially asynchronous system is the model of the internet and the one used in the project.

2.2 Design choices

2.2.1 Bootstrapping

Bootstrapping is the only weakness of the system. During bootstrapping the bootstrap servers poses a single point of failure. It reads the replication degree and the amount of partions wanted from a configuration file and waits until enough nodes have checked in to be able to form the distibuted & partitioned KV-store. It sends the generated lookup table to all nodes.

Each node then extracts the list of nodes of its own partitions to start the *SequenceePaxos*, which in turn starts the *BLE*, for the partition. Moreover, Ev.P is used to monitor all nodes in the system as to only provide a newly connected client only with a not suspected node as entry point

2.2.2 Partitioning

Partitioning is done automatically through the bootstrapping node which divides the total number space an *integer* can have to create even partitions. This decision is the result of the provided code base which includes a *String.toHash()* method returning an *integer*.

2.2.3 Replication

A shared memory abstraction is too weak to provide a compare and swap operation, therefore motivating the need of implementing a replicated state machine. This was done using the *SequencePaxos* algorithm from the course which is a leader-based sequence consensus.

2.2.4 Resilience

Google has mentioned that $f = 3$ is a good value, providing enough resilience for a real world scenarion. Being a small scale project for learning purposes the decision fell upon $f = 1$ for development and test. Nonetheless the value can be changed in the configuration file.

2.2.5 Client

The client only needs to know one correct process to be able to use the service. It connects to one server, which in turn will return a list of not suspected nodes at the time of connecting. The client may then go through the list or keep using the same node until an operation failes.

3 Implementation

3.1 Operation Invocation

As the saying goes: "A picture says more than a thousand words.", 1 shows the event flow from an operation invocation by a client until being included in the list of commands to agree upon.

3.2 Operation reply

Through the guarantees of the sequence consensus the KVstore safely executes an operation after a *SCDecide* event. For the client to know be informed about the result, i.e. the reply to the invocation, several approaches exists.

- Have the first receptor inside the responsible partition save the UUID + source IP of the operation. This allows to let only this node send a reply.
- Have the leader send a reply since there will always be a leader once established unless $f >$*resilience* .

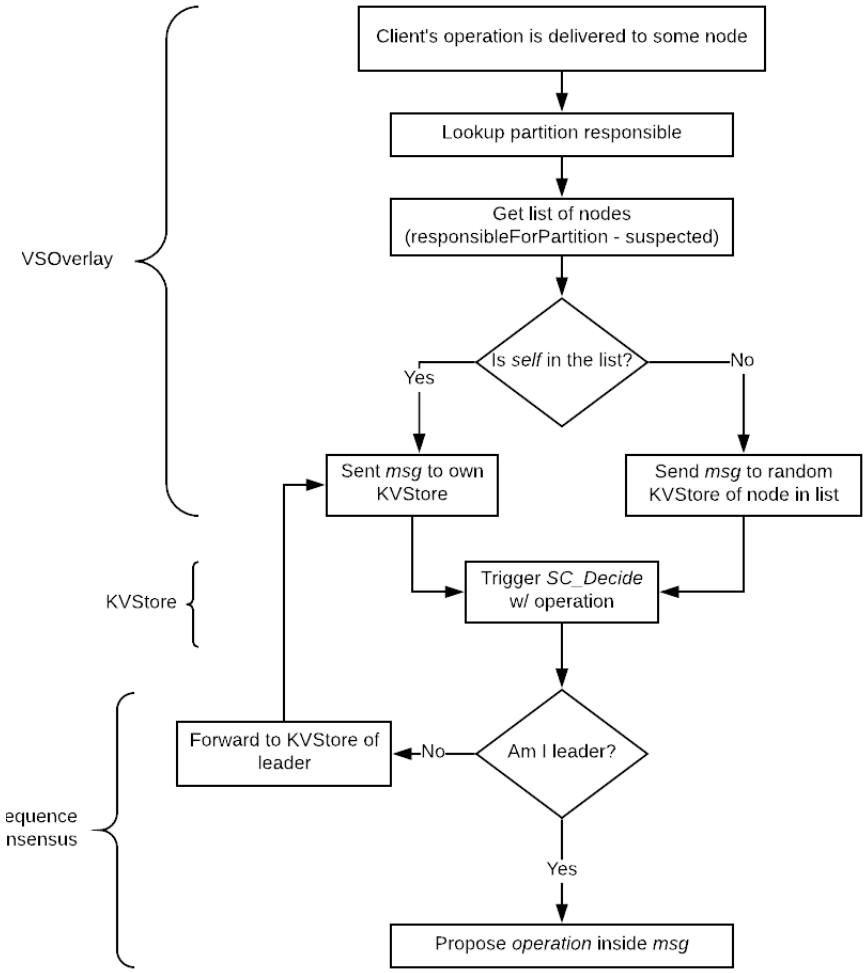

Figure 1: Flowchart of operation invocation

- Have all nodes in the partition send a reply

The first case allows for an omission of the reply should the receptor crash before sending the reply message although the operation might still be executed by the others.

While the second case has the least amount of messages, the last option was implemented. The client already has the functionality to ignore replies which it isn't waiting for. Moreover, the client removes the pending entry when receiving a reply, thus ignoring duplicate replies which can be identified thanks to the UUID. Furthermore, it allows for testing cases to count the amount of replies received and comparing the results. Inconsistencies in the RSM or slow nodes could be detected by this.

3.3 Codebase

The scala codebase was used as a head start for the implementation. Figure 2 shows the component diagram of the provided scala code base.

3.4 Added code

Figure 3 shows only the added components and affected components from the template except *Timer* and *Network* which should be clear. Furthermore the diagram shows the component using the network component although the code from the exercises rely on a *PerfectLink* in the code. Being only a wrapper for the network component the diagram shows no diferrence to the real world behaviour. This is thanks to the underlying usage of TCP which provides session based FIFO perfect links.

The components from the exercises where copied for the most part, but refined in some ways to fit the use case. Especially regarding the starting of *Ev.P* and *BLE* since the nodes don't know the other nodes in their replication group until bootstrapping is finished which required the introduction of a *Monitor* and *SetTopology* events.

4 Testing

5 Responsibilities

The project started out with both members wanting to contribute in the development but the decision was revisited later on to use the proposed division of labour to coding and testing.

6 Conclusion

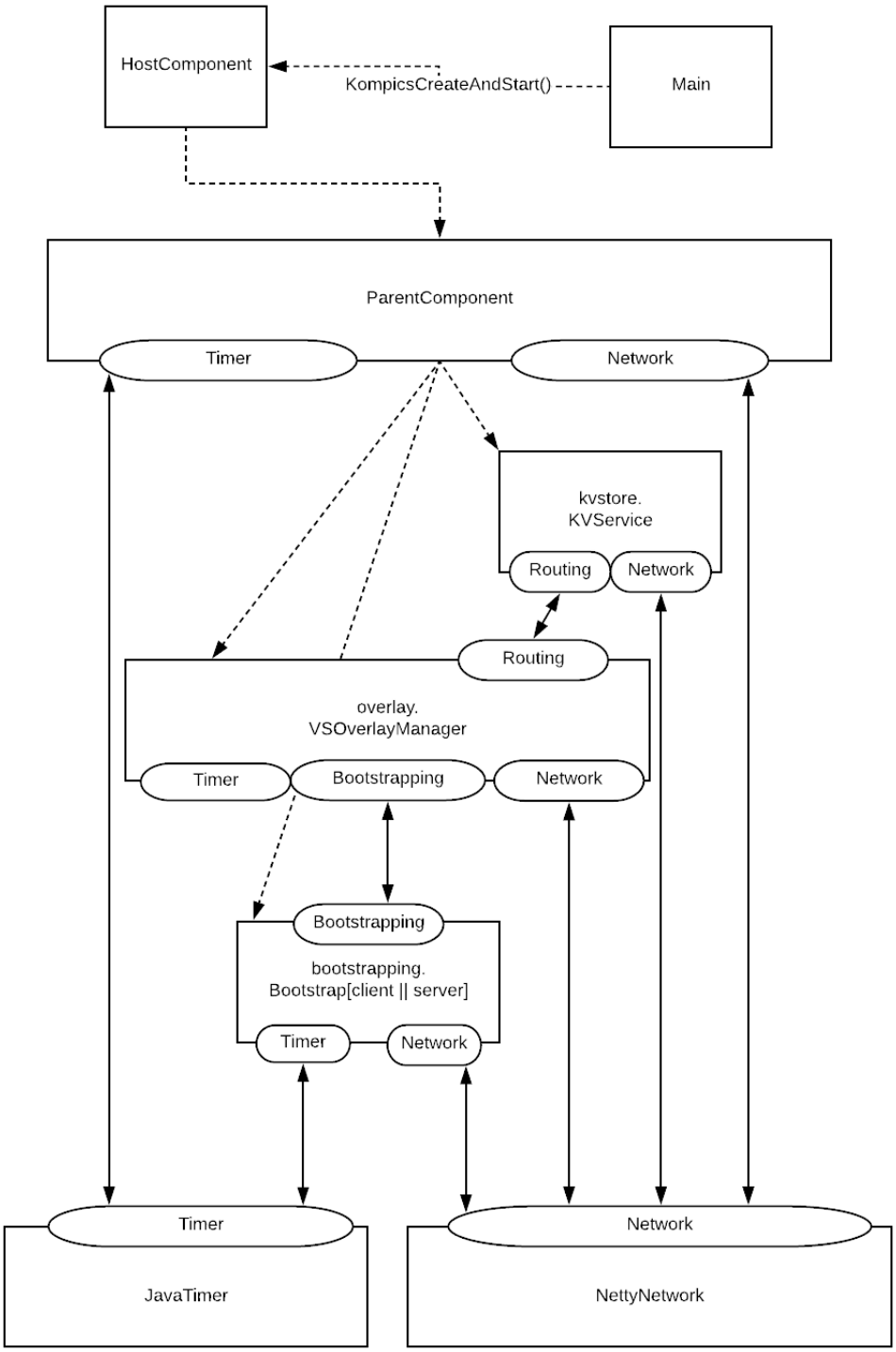

Figure 2: Component Hierarchy of the code template

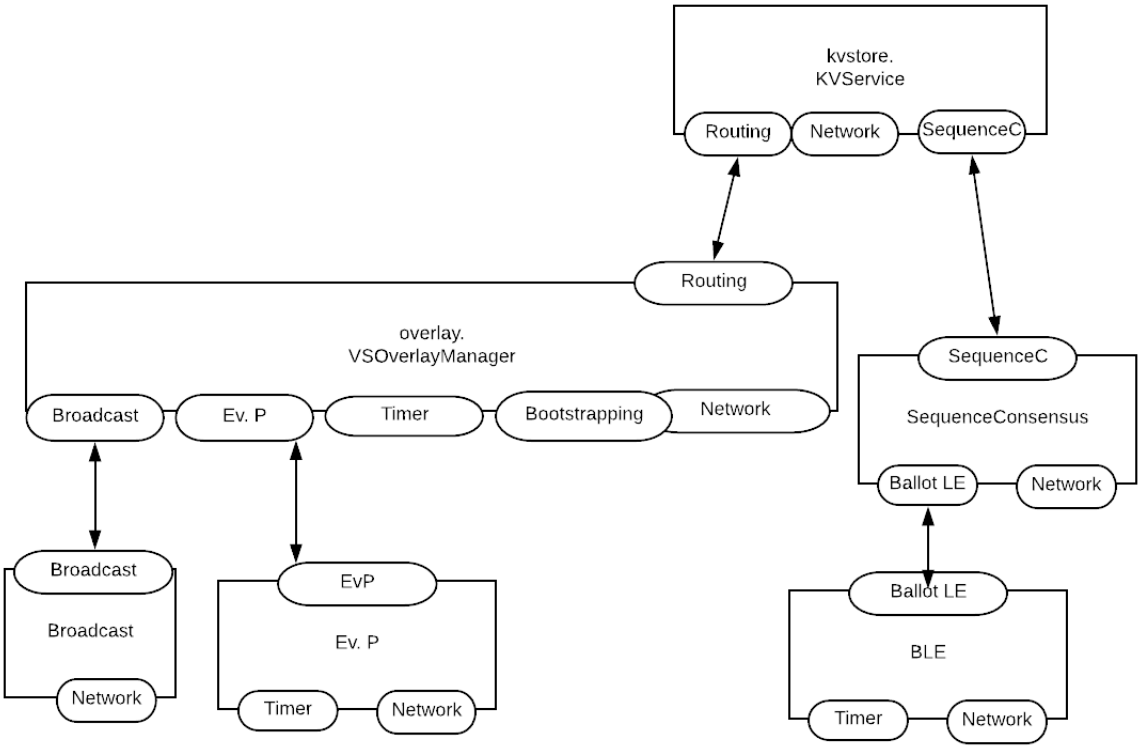

Figure 3: Purview of the added components