ID2203 - Distributed Systems, Advanced Course - Report

Jacopo Castello, Fernando González, Marta Bertran, Svenja Räther

March 2020

1 Introduction

In the project we implemented a partitioned and distributed key-value store. The system can handle the basic operations of a key-value storage system: GET, PUT and Compare-And-Swap.

The project is build up based on the Kompics programming model for distributed systems that implements protocols as event-driven components connected by channels. Those provide first-in-first-out (FIFO) order, exactly-once (per receiver) delivery and the queuing of events at the receiver until they are scheduled to be executed. Kompics event-handling allows that events are broadcasted across all connected channels, thus same events can be received by many components. Matching and handling of events is done by the components event-handlers. Given this, we assume to have underlying perfect FIFO Links (Reliable Links). [1]

We further assume a partially synchronous system. Those abstractions can be combined with the eventually perfect failure detector to form a Fail-noisy model.

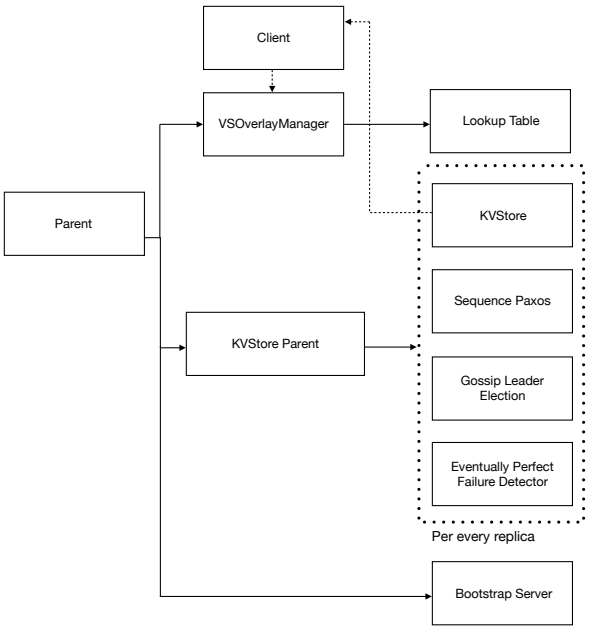

Figure 1: Structure of the components used to set up the system.

The structure of the different blocks that compose the final system is shown in figure 1. The initial component is called Parent component. It creates the Overlay Manager component, responsible of forwarding the operations to the nodes, as it is the one in charge of keeping the Look up table updated with the nodes of each of the Paxos groups. Each server that is started by the system creates a Parent component and all corresponding components once their initiation is triggered. To start the system, the bootstrapping procedure is started and the specified amount of nodes need to be booted in order to initialize the Lookup table and start the system.

Moreover, the Parent component also creates the KVStore Parent component, which is the responsible of creating and starting new replicas. At the start of each configuration (also for configuration 0), it creates the corresponding replicas, each of which consists on several modules: the KVStore for handling the execution of the operations, the Sequence Paxos and Gossip Leader Election for implementing the Leader-based Sequence-Paxos algorithm and the Eventually Perfect failure detector for detecting process failures.

Once the system is started, a client can issues an operation, that is therefore forwarded to the Overlay Manager, who ensures that it gets handled to the consensus module. Once the command is successfully decided by the consensus and executed by the KV-store of the assigned group, the reply is send back to

the client.

2 Infrastructure

2.1 Partitioning key-Space

The system supports partitioning of the key-space over the replicated groups of available nodes. Each group is assigned a corresponding partition index. All nodes in the group are responsible to store the key-value entries that get assigned to this index. Each of the nodes in a group will hold a copy of the key-value entries belonging to the index that has been assigned to them. Once the initial setup is done, the number of groups is static for the remaining time that the system is running.

The system implements a Lookup table which allows us to look up the partition index for each key stored in the system. It ensures, that any key given to that function will be placed in any of the available groups with an equal probability. It does so by calculating the positive modulo of the hashed key. Any key in this scenario will thus be assigned an integer index value in the range between 0 and the number of groups -1.

The partitioning procedure is visualized together with the replication procedure in figure 2.

2.2 Replication

The size of a group can be specified in the configuration of the system. The setup ensures that each group will have at least the configured group size and at most one less than double of the group size. The Lookup generation partitions the available nodes (limited by the configured bootThreshold) into groups.

Further testing of the system was done with a group size of 3 nodes (6 nodes in two groups having 3 nodes each). The minimum number of three nodes per group has been elected for ensuring that the processes inside the group will be capable of achieving a quorum from a majority of processes. This way, replication of the key-value entries among the nodes of each group can be achieved. Since we operate in a partially synchronous system, we assume a majority of the nodes in a group to be correct.

An example of the partitioning and replication procedure is shown in figure 2. After the initial boot of the specified number of nodes (in this example 6), those get partitioned to groups with at least the number of nodes specified for one group (in this case 3). In the KV-Store scenario, clients can send operations including a key-value pair. Each key of a key-value pair gets first hashed to an integer value using a hash function and then reduced to one partition index

of the existing groups. In this example, the key "Example_key" will be stored in the group with index 1. There the consensus component take care of the command and make sure that all the nodes agree to execute the operation with the key-value pair on their KV-store.

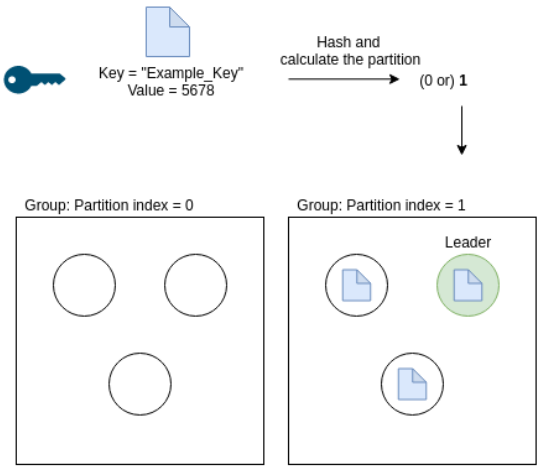

Figure 2: Partitioning and replication

2.3 Failure Detector

The failure detector that has been used is an eventually perfect failure detector, as we are implementing a partially synchronous system. Each node in our system has its own failure detector, with a complete view of all the nodes of its Paxos group. For a node to be considered alive by the rest of the processes, heartbeat messages (requests and replies) have to be exchanged between them at regular timestamps. Using the eventually perfect failure detector our system is capable of achieving:

- Strong completeness: Eventually every process that crashes is permanently detected by every correct process, which ensures liveness. If a process crashes, it will not send heartbeats to the other processes, and that will be detected.
- Eventual Strong Accuracy: Eventually, no correct process is suspected by

any correct process. A process which is inaccurately suspected by an other correct process is restored when the heartbeat is received.

The implementation of the perfect failure detector that has been used has been taken from the course materials. Once a process is suspected, a message is sent to the overlay manager, which removes the node from the Paxos group and issues the start of a new configuration (section 4). Once a node is suspected, the policy is set to create a new configuration, not waiting for a possible restore of the crashed process. This way, safety is ensured.

2.4 Lookup Test

To ensure the functionality of the partitioning, the replication, and further functionality needed later in the project, a test for the lookup table was constructed. This tests all existing functions in the lookup table and verifies their functionality by applying five small tests on a sample input scenario:

- Lookup Table generation
- Lookup the partition index for a key
- Lookup group for a node
- Adding a node to a group
- Removing a node from a group

3 KV-Store

The KV-Store is the module where the key-value pairs are stored. In our case, for being able to access the register data from the different configurations (section 4), we have implemented a persistent storage. We use a directory structure: each node has its own folder, with a file created per key containing the corresponding value.

3.1 PUT/GET/CAS Operations

After having the basic infrastructure in place, the `GET` (read), `PUT` (write) and `CAS` (compare and swap) operations were added.
After their codification, firstly, a manual try out was carried out from the terminal to make sure that the corresponding operations were working with a correct behaviour. Once this was assured, a series of tests were developed (along with a scenario that maintained the logic of these tests) in which the functionalities of the three basic operations were checked to behave correctly.

3.2 Paxos

Our system implements a leader-based sequence Paxos [2] for achieving consensus in the sequence of commands to execute. For ensuring termination of the Paxos algorithm, we have also implemented a Ballot Leader Election, Ω. The leader is elected together with a ballot number that is globally unique and locally monotonically increasing. The elected leader is the responsible of starting a new round of sequence Paxos with its corresponding prepare phase. The Ballot Leader Election implementation fulfills following properties:

- Completeness: Eventually, every correct process elects some correct process, if a majority of processes is correct.
- Eventual Agreement: Eventually, no two correct processes elect different correct processes.
- Monotonic Unique Ballots: If a process L with ballot n is elected as leader by a process p, then all previously elected leaders by p have ballot numbers m with m ¡ n, and the pair (L, n) is globally unique.

While implementing consensus with the ballot leader election, we ensure our system provides:

- Validity - If process p decides v then v is a sequence of proposed commands.
- Uniform Agreement - If process p decides u and process q decides v then one is a prefix of the other.
- Integrity - If process p decides u and later decides v then u is a strict prefix of v.
- Termination - If command C is proposed by a correct process then eventually every correct process decides a sequence containing C.

The following assumptions need to be taken:

- The Leader-based Sequence Paxos is optimized for the case when a single proposer runs for a longer period of time as a leader. Although the leader will not be aborted for a while, it ensures safety if aborted.
- Each process acts in all roles as a proposer, acceptor and learner (replicated state machines).
- The links of the systems are FIFO Perfect Links, which ensure the messages from source to destination arrive in the same sent order. This implies commands can be accepted incrementally.

Our solution implements three different operations: Read, Write and Compare-and-swap. The Paxos component is set on a layer below the KVstore component. Once a command is sent to the system, the process with the role of the leader proposes it and, once uniform consensus is achieved between all the processes

on the sequence of the proposed commands, it is executed and the result is sent back to the client. We ensure linearisability by being the leader the only node proposing commands. The Overlay Manager component is responsible for updates on the topology of the different partitions.

3.3 Tests for Paxos

The testing for the Paxos algorithm is performed through the issuing of several Read, Write and Compare-and-swap commands. This test is called *OpsTest* and it uses the scenario *ScenarioClient*. We want to ensure that the commands are eventually performed in the order they were issued. We have tested three possible cases:

- Simple Operation test: This test performs Read operations of several keys. As we did not perform any Write operation beforehand, we want to ensure that the system does not return any value.
- Write then Read test: This test alternates Write operations with its corresponding Read. What is wanted to be ensured is that the Read returns the value previously written by the Write operation.
- Compare and swap test: Initially, this test writes values to all the keys we want to check. Half of the keys have the same value, but half of them do not. Then, we perform CAS operations in every key, setting as the expected value the one kept constant in half of the keys. At the end of the test, we make sure that all the keys contain the same value. The keys which were assigned initially different values will already have them changed by the CAS operation.

3.4 Linearizability

Throughout the course of the entire project it had to be ensured, at every step, that a property was satisfied. This property is the Linearizability.

We ensured linearizability by using the Leader-based sequence Paxos algorithm, in which the new commands are always proposed by the leader to all the other processes. It is also the leader the one in charge of issuing the Hand-over command to all other replicas in the start of a new configuration.

For this purpose, a scenario has been described which, in an immovable way, was securing this property in each phase. From the beginning, when the operations did not suppose more complexity than the realization of its function together with the application of the consensus and paxos until the reconfiguration of the system itself. In all these scenarios, linearization was guaranteed.

3.5 Tests for Linearizability

Speaking more deeply about the mentioned property, first of all it must be understood that a linearizable system is one in which each of the operations that are being executed seem to do it instantly at a point between the invocation and the response of this operation.

So, a scenario was developed in which a series of operations were executed a significant number of times (such as 10,000 times). These operations would be entered into a special data structure such as a FIFO queue, so that the order in which the operations were called would be automatically saved. In addition, for each response received for each operation executed, this response would be stored in a queue with the same characteristics as those of the calls in order to guarantee the order of reception of the responses generated by the operations being executed.

In the end, after all the proposed operations were executed, both queues were "enqueued" (remember that the FIFO queues obey the "First-in-First-Out" logic) and it was checked that the order of the responses corresponded to the order of the operation calls. Only if all the operations had succeeded each other correctly we could confirm that the property was guaranteed, since with a simple error, this property would be violated.

But not only the order is important, that's why we also consider that the values returned by the operations correspond to the previous sequence executed. For this reason, we also check that the current value of the response obtained from the operation call is consistent with the previous ones executed. In this way, the current values are compared with the previous ones with the aim of guaranteeing coherence in the execution.

This previous test is valid for testing the linearizability during the gradual development of the consensus techniques and other tasks regarding the operation (at the algorithm level) of the system functions, but the approach should be changed to test the linearizability together with the reconfiguration of the system.

To do this, the test would remain the same but the scenario would be changed. In this scenario, we intentionally establish the situation in which a node stops working and we check that the reconfiguration mechanisms (already implemented) restores this node and that the linearizability remains intact.

4 Reconfiguration

Our system is implementing a Replicated State Machine (RSM) using a set of processes, tolerating the failures of up to half of the nodes. Therefore, a way

is needed to replace faulty processes: process reconfiguration. Our system is capable of detecting crashes of processes through the eventually perfect failure detector suspicions. Once a process is suspected, all the other processes of the Paxos group are replaced by the creation of new replicas in the new configuration. Configurations are instances of Leader-based Sequence-Paxos, each of them containing several replicas that run the consensus algorithm. The used algorithm for implementing Leader-Based Sequence-Consensus, together with reconfiguration, has been taken from [3].

At the moment the first process detects a crash, it issues a Stop-sign command, that will be the last command of the decided final sequence of the configuration running at the time. Then, after issuing the command to all the nodes of the Paxos-group, the alive nodes boot a new replica on the new configuration. Then, the Overlay Manager updates the topology that will be used in the new configuration. Processes can have several replicas at the same time, but only one of them will be running. Each of the replicas has one of the states:

- *"RUNNING"*: The replica is executing the sequence-Paxos algorithm.
- *"HELPING"*: The replica is helping the last configuration issued to reach consensus in the final sequence.
- *"WAITING"*: A new configuration replica has already been booted, but the hand-over to the new replica has not been issued yet.

When the Stop-sign command is decided, the replica in the new configuration can start. The activation process of the new replica is done through the sending of a Hand-over message to the Overlay Manager of the leader, which will then send the final decided sequence to all the replicas of the new configuration through a special event called *SC_Handover*. The sent sequence is the initial sequence chosen in the new configuration. Old replicas will be killed once they are not needed any more.

To summarize, the implemented Reconfiguration supports successfully moving to a new configuration once a crashed process is suspected.

4.1 Tests for Reconfiguration

This test is similar to the test created for testing the Leader-Based Sequence-Paxos (section 3.3). The test is called *ReconfigurationTest* and it uses the same scenario as before. As in this case what we want to test is correct functioning of the system after the crashing of a node, once we have issued the sequence of commands of the test a node of the group is killed. When this happens, the same initial commands are issued again. The goal of this test is ensuring that the whole system keeps its correct performance taking out the dead node at the creation of the new configuration. We check that the last set of commands do also have the desired outputs.

5 Advanced Tasks: Leader Lease on Sequence Paxos

As most of the operations performed are usually read operations, the efficiency and performance of a store can be improved by implementing a secure leader based mechanism which unlock the possibility to respond to read operations without involving a majority of replicas, the so called "fast read". This lays a main challenge which is the possibility of having multiple nodes acting as leaders due to network partition, hence violating linearizability. This is solved by implementing a timed lease based mechanism in which the Leader holds the lease for a determined amount of time after which the lease expires. During the time it holds the lease the Leader can respond to read operations directly with its local state. When the lease expires the node loses the role of Leader and a new one might be elected. This achieve the goal of having time disjoint leaders and preserve linearizability. In order to provide safety we have to take in consideration not only the possible delay in the network but also the clocks drift. For this reason the start of the lease period is always set to the moment when the proposer sends the Prepare(n). As for the clock drift we simply use p, which represents the clock drift rate, and we use it in our timing comparison both on the proposer and on the acceptors in order to ensure safety, with the two time comparison being respectively:

$$C(T) - tL < 10 * (1 - p)$$

and

$$C(T) - tProm > 10 * (1 + p)$$

We implemented the Timed Leader Lease on top of Sequence Paxos. As before the acceptors reply with a Promise to the Prepare(n) of the proposer, promising to not accept proposal in lower rounds. On top of that in Timed Leader Lease they also promise to not join higher round until time t + leaseDuration. When the proposer gets promises from a majority then it holds the lease for the determined duration of the lease itself. The two main elements, the lease duration and the clock drift rate, in our code has been implemented as parameters (leaseDuration and clockError) which are manually set in the configuration files of the project.

5.1 Tests for Leader Lease on Non-reconfiguration

In order to test the better performance introduced by using the Timed Leader Lease we set up a simple test where different types (write and read) and numbers of operations (10, 100, 1000 and 10000) were executed first with the parameter leaseDuration set to 0 ms, hence no Leader Lease, and then with leaseDuration set to 10000 ms. For each number of operations the same test was executed a fixed number of times, defined in the test through the parameter rounds, and the final result returned was the average over the results of the single round. This was done inside the OpsTest, hence still assuring the correctness of the

operations, changing the number of operations by simply iterating over a predefined sequence, nMessagesL in the code, containing the different numbers of operations to run. The results obtained are showed in the following tables.

Results for leaseDuration = 0 ms

Number of OPs	**Time for completion (ms)**
10	7913
100	8801
1000	13869
10000	84657

Results for leaseDuration = 10000 ms

Number of OPs	**Time for completion (ms)**
10	6263
100	6468
1000	7788
10000	24349

Comparing the results for the same number of operations we can clearly see that, as expected, the introduction of the Leader Lease introduces better performance.

6 Advanced Tasks: Leader Lease on Reconfiguration

We then proceeded to implement the Timed Leader Leased together with our system now supporting dynamic reconfiguration. The overall functioning of the Timed Leader Leased remains the same but it is not anymore implemented over a static store but rather over a dynamic one capable of handling failures, nodes leaving and joining.

6.1 Tests for Leader Lease on Reconfiguration

For testing the correct implementation of the Timed Leader Leased in our now dynamic environment we used a procedure similar to the one described above. The main difference lays in the changing of scenario being used. In this case we used the scenario designed for the testing of the reconfiguration functionalities where after correctly starting the system a node is killed in order to trigger the reconfiguration countermeasures now in place. In this dynamic scenario we ran the same tests mentioned for the testing of the Timed Leader Leased without the reconfiguration: different types (write and read) and numbers of operations (10, 100, 1000 and 10000) were executed first with the parameter leaseDuration set to 0 ms, hence no Leader Lease, and then with leaseDuration set to 10000 ms. For each number of operations the same test was executed a fixed number

of times, defined in the test through the parameter rounds, and the final result returned was the average over the results of the single round. This was done inside the ReconfigurationTest using the test strucutres from OpsTest to assure the correctness of the operations, changing the number of operations by simply iterating over a predefined sequence, nMessagesL in the code, containing the different numbers of operations to run. The results obtained are showed in the following tables.

Results for leaseDuration = 0 ms

Number of OPs	**Time for completion (ms)**
10	9221
100	9283
1000	14750
10000	92139

Results for leaseDuration = 100000 ms

Number of OPs	**Time for completion (ms)**
10	8088
100	8748
1000	13820
10000	59719

7 Teamwork

First of all, we have used a shared repository in Github which has let us work in a collaborative and productive way in the different tasks performed since this tool facilitates the distinction of the same ones regarding the code.
We have followed a work methodology in which, at least, the team met once a week. In the first meeting all those functionalities described in the project's statement were arranged. From that point on, a separation of tasks was carried out with respect to the research of the different areas that were dealt with.
After adopting a general knowledge of the project requirements, the technique known as Pair Programming based on iterations was applied. That is, we were distributed in pairs and, with the assigned tasks, we were developing the agreed parts. During this process we met in those situations that were considered necessary, maintaining a constant pace of work from the beginning of the project until the end of it.
Below is the distribution and arrangement of the different tasks required in the project statement (including advanced tasks):

Task	Contributors
Infrastructure	Marta and Svenja
KV-Store (Client Side)	Fernando and Jacopo
KV-Store (Server Side)	Marta
Reconfiguration	Marta and Svenja
Leader Leases	Fernando and Jacopo
Reconfigurable Leaser Leases	Fernando and Jacopo and Svenja

8 Summary

8.1 Completed Tasks

Our implementation of the simple partitioned, distributed in-memory key-value store with linearisable operation semantics builds upon the project template. The following tasks were implemented on top of the template:

- Infrastructure
- KV-Store
- Reconfiguration
- Advanced Tasks: Leader Leases
- Reconfigurable Leader Leases

Each of those was tested by different scenarios to ensure the functionality and linearizability of the system.

8.2 Optional future extensions

The system fulfills the requirements given by the course. However, there are optional extensions that could be implemented in the future to extend the current functionality of our key-value store.
Examples of those include:

- Extension of the static nature of the number groups: This would enable the operator to add groups even after the initial set up of the system.
- Extension of Reconfiguration: Another case in which reconfiguration is often used in production systems is the change of configurations from an external agent. This would enable the operator to manually exchange processes if needed. (Optionally, this could go hand in hand with optimisations for the handover)

References

[1] "Kompics documentation,"

[2] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001.

[3] L. K. Seif Haridi and P. Carbone, "Lecture notes on leader-based sequence paxos - an understandable sequence consensus algorithm,"