ID2203 Distributed Systems, Advanced Course Project Final Report

Chen, Zidi

March 2021

1 Introduction

This project is to design, implement and test a distributed key-value store. The store should be replicated, partitioned, and it should support the PUT, GET and CAS operations.

The github for source code: https://github.com/Chen-Zidi/id2203-project.git

2 Design of System architecture

In this project, the model is assumed to be partially synchronous, so it is a fail-noisy model. In this section, I mainly introduce the partition, leader election and the replication algorithm that I have chosen for the project. How the failure and the network link are solved are also included in the next subsections.

2.1 Sequence Paxos with Leader

In this project, Sequence Paxos is applied with a single leader for reaching the consensus within the replication group. The network which was already provided by the project template is considered to be the perfect FIFO link. These are the basis for leader-based Sequence Paxos.

The motivation is that leader-based Sequence Paxos is a efficient and strong algorithm, and it have already been implemented it in the lab. Actually, I have considered to use Read-Impose Write-Consult Majority algorithm. But I think it is not strong enough for CAS operations. For example, the old value might be changed by other clients after I finish the reading phase in the CAS operation. In this situation, my CAS operation is actually overlapping with others' modification, so the CAS operation should not success. The Sequence Paxos can prevent this situation by making all the operations in sequence. Then, I made a choice between leaderless Sequence Paxos and leader-based Sequence Paxos. Considering the efficiency, I think the leader-based Sequence Paxos with single leader is a better choice. The leader-based Sequence Paxos provides validity, uniform agreement, integrity and termination.

2.2 Leader election

To select the leader, the gossip leader election which was implemented in the lab 6 is perfect. The protocol enables nodes to gossip with each other from time to time to check if the others are alive. This would be enough for this project as the failure detector, since the failure of the leader will cause the election of a new leader. Gossip leader election should works fine in the leader-based Paxos Sequence, with majority alive nodes.

2.3 Partition

The single partition should be enough for this project. As shown in the figure 1, there are 4 nodes in the single partition. There is one node as the leader, who acts as proposer in the Sequence Paxos. The leader is decided by gossip leader election. If the client requests an operation at a node who is not the leader, then the node should forward the operation request to the leader. Then, the leader can propose the operation in the Sequence Paxos component.

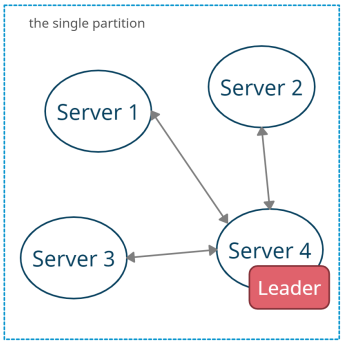

Figure 1: System partition

The predefined group node number is 4. In this case, if the leader dies, the rest of the 3 nodes can still reach the majority and decide another leader. I would say the replication degree in this case is 3, because the each operation will be replicated by the other three nodes. The resilience is 1 in this case. If more than 1 node dies, the group can not elect a new leader and continue with Sequence Paxos.

3 Implementation

In this section, the implementation of the project is explained.

3.1 Developing environment

The developing environment is IntelliJ IDEA on the ubuntu system. The ubuntu is installed on VM virtualBox on Windows.

3.2 Added components

Based on the template, I have added two components: Sequence Paxos and Ballot Leader Election. As figure 2 shows, the Sequence Paxos component is used by KV-Store and Overlay component, and it uses Ballot Leader Election component and network component. The ballot Leader election uses network and timer component for collecting heartbeats and passing messages.

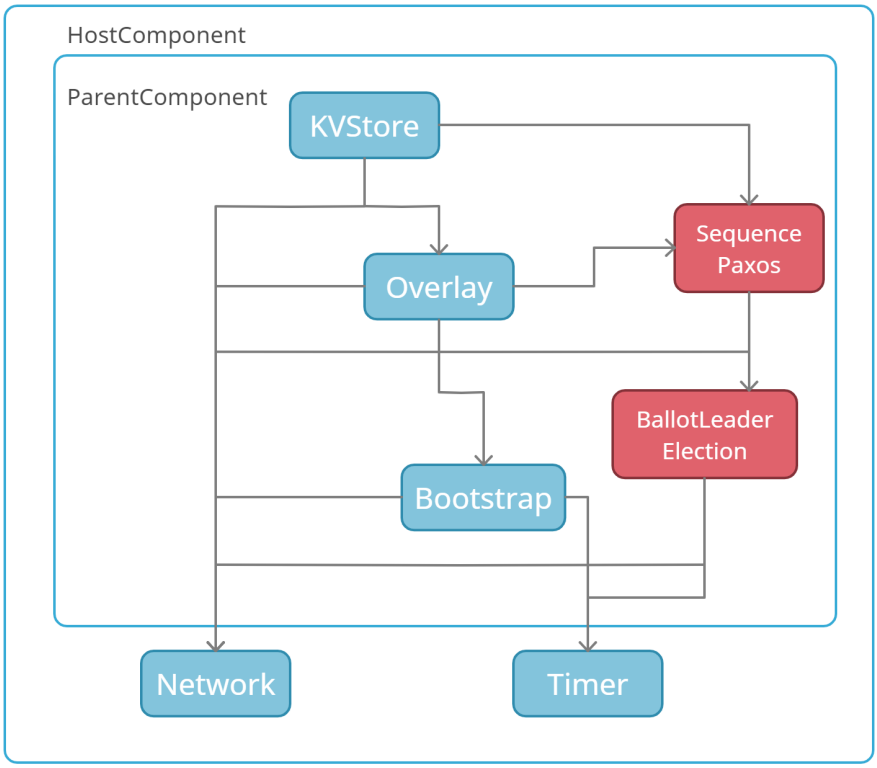

Figure 2: System structure

3.3 Operations

There are three classes which extend *Operation* class: *Put*, *Get* and *Cas*. The *OperationResponse* is extended to *GetResponse*, *PutResponse* and *CasResponse*. *ClientService* class sends the operations request, and the *KVStore* in the server

will receive the operation requests and process them. The *KVStore* has a *data* variable which is a Hash Map, storing all the key-value pairs in each node.

3.4 Overlay manager

When the overlay manager boots, it will initialize the Sequence Paxos with topology. This initialization is added in the Sequence Paxos component as a request event.

3.5 Leader-based Sequence Paxos

This component was basically adopted from the lab, with a few modifications. The network provided in the template was adopted. When booting the system, the Sequence Paxos gets the topology of all the nodes in the system. The Sequence Paxos would then call ballot leader election to start by passing the topology. When the server receives the operation request in the *KVStore*, the leader proposes this operation in the Sequence Paxos component. If the current node is not the leader, then the operation should be passed to the leader. The leader should be responsible to propose all the operations, including read and write. The *KVStore* will handle the operations in the local Hash Map *data* when *SequencePaxos* decides.

3.6 Ballot leader election

The Gossip leader election is also adopted from the lab with modifications. A request event *BLE_Start* is added, which should be called by *SequencePaxos* to initialize the topology. It is adopted to use the network and timer component in the template. When a leader dies, a new leader should be elected in this component.

4 Tests

In this section, I will explain what kind of tests I have designed, the reason, and the result of the test. In the server/src/test/scala folder, there is a *MainTest* class which is used to ensure the three tests works sequentially.

4.1 Simple operation test

This is to test the simple operations including put, get and cas. This test is adopted from the template, to ensure the operations are working fine. In the server/src/test/scala folder, *SimpleScenario*, *OperationTest* and *OperationClientScenario* are for the simple operation test. *SimpleScenario* is to simulate to start a client and servers. There is a modification that the network should not have a random latency when sending messages. *OperationClientScenario* simulates the client behavior to put some values, get some predefined values and cas some

predefined values in the data store. In the *OperationTest*, the simulation results are verified. According to the test result, the simple operations worked fine.

In addition, I have tested the following situations manually.

1. Get: Get an non-exist key. [Result: NotFound response]
2. Put: Put an exist key. (The key value is expected to be updated.) [Result: Ok response, the value is updated.]
3. Cas: Cas with a non-exist key. [Result: NotFound response]
4. Cas: Cas with a correct key, but false old value. [Result: NotFound response]

The results were as expected.

4.2 Linearizability test

Under the test folder, *LINTest* is to verify the test results, *LINScenario* is to start up a client and servers, and *LINClientScenario* is to simulate the client behavior. Linearlizability is compositional, so it is valid for the whole system if it is valid for one register. In the linearlizability test, there is a sequence of operations to request: put - cas - get - put - get - get, the result of each operation is verified according to the sequence. This can be seen as the Wing Gong algorithm without undo operation. [1] In the sequence of operations, we can match:

1. put - cas: write - write
2. cas - get / put - get: write - read
3. get - put: read - write
4. get - get: read - read

So this sequence includes ww, wr, rr, rw situations. The operations are related to each other by performing on same common key-value pairs. These operations are executed sequentially in a global time and the result of each step was as expected.

4.3 Node crash test

The tests of node crash are also considered. There are two possible situations: the leader crashes or a normal node crashes. In the group of four nodes, the system can tolerant maximum 1 node death, otherwise it is not possible to elect a new leader and reach consensus.

4.3.1 Normal node crash test

Under the test folder, *ServerCrashTest* is to verify the test result, and *ServerCrashScenario* is to simulate the server crash behavior. *ServerCrashScenario* is based on *SimpleScenario*, and it kills one node additionally in the scenario. In the *ServerCrashTest*, it verifies if the simple operation test still works after one node crashes. According to the result, the system works fine even if one node dies. In this test, I actually can not know the crashed node is leader or not, so I did the additional leader crash test manually.

4.3.2 Leader crash test

The leader crash test is done manually. I started 4 nodes, the ports are: 45678, 45679, 45680, 45681. The other three nodes were connected to 45678. Figure 3 shows that at time 21:21 the node at port 45680 is the initial leader through ballot leader election.

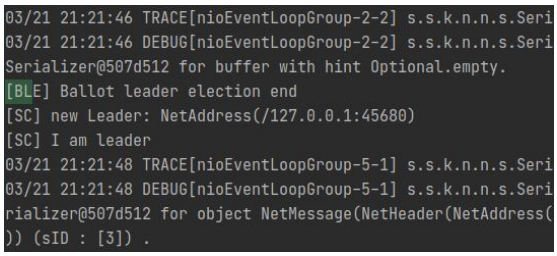

Figure 3: Initial leader

Then, I killed the process of the node at 45680. In figure 4, at time 21:23, the node at 45681 became the new leader through ballot leader election. So the system works fine even if the leader dies, the gossip leader election algorithm allows to elect a new leader with the rest three nodes.

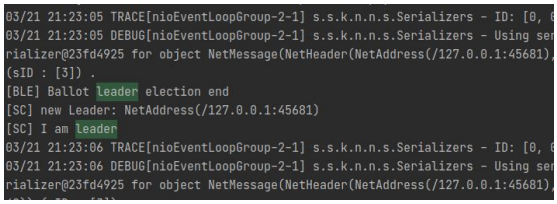

Figure 4: New leader

5 Conclusion

In conclusion, I have implemented the basic infrastructure, Key-Value store with put and get operation, and additional compare-and-swap operation. My solution

is to apply single partition with Leader-based Sequence Paxos. I think that this simple solution works fine without explicit failure detector and broadcast component.

The difficult part of this project is to think carefully about the design of the system. During the design phase, I think I earned a deeper understanding of what I am learning. In addition, the test of linearizability was time-consuming, I have modified a few different versions about how to do it to make it reasonable.

I think this project is really challenging and interesting. The possible improvement can be to consider more partitions and reconfiguration.

References

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