ID2203 Final Project

Sihan Chen

March 08 2021

1 Introduction

The goal of this project is to create a distributed key-value store. Kompics Scala is used as basic library[1] to provides a component based model to deal with message-passing, event handling and scheduling in distributed systems. What should be focused on in this project is to ensure the consensus between one primary (leader) and several backups with a replicated state machine abstraction. Firstly, three kind of operations (GET/PUT/CAS) are implemented. Moreover, each replica is ensured to have the same sequence of operations, which yields the same state transition on each replica by using sequence paxos. In addition, fault tolerance is promised by a similar idea like the replication set in MongoDB, that uses heartbeat to keep following the leader and elect a new leader when lose the contact. Tests are written to test the correctness of the operations. The address of the project is https://github.com/Spycsh/distributedKVstore.

2 Infrastructure

In this section, the basic infrastructure of the system will be explained.

Due to time restriction, no partitioning over key space is implemented. Only one partition is supported and in the configuration file it is specified that there are three nodes in the partition, one primary and two replicas. Of course, more nodes can be connected as specified in the configuration file. Fault tolerance is ensured that whenever one leader node crash, a new leader will be elected out. Sequence based Paxos is used to ensure the consensus on operation sequence with a replicated state machine. Ballot leader election is used to reduce the redundant messages sent during propose phase. Reads and Writes should be sent to each replica to make sure that they hold the same operation sequences and yield same value sequences in the KV store.

The main workflow is shown in Figure 1. As shown in the graph, user execute a GET or PUT or CAS operation in the client console, and the operation will be parsed (now only one-character key is allowed, e.g. PUT A aValue) and wrapped as a case class to the net. The net will broadcast the message and the KVStore component will receive the operation and trigger the SC_propose,

namely send it to the SequencePaxos component. The SequencePaxos component will use leaderElection component. The overlay manager manages the routing of messages, booting and the topology initialization.

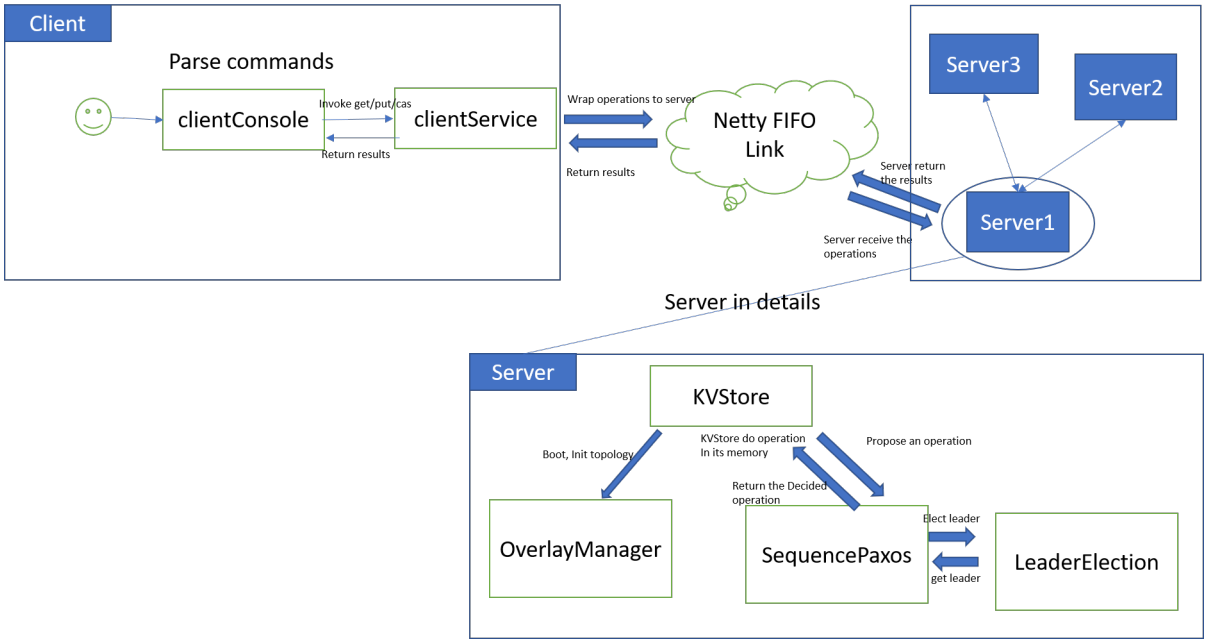

Figure 1: Workflow

3 Implementation

In this section, the implementation of the system will be explained in detail.

3.1 Consensus

The consensus algorithms should be chosen to ensure the value that under the GET/PUT/CAS operations to be the same on all the servers. The initial idea of algorithm to support the read/write operations is the (N,N) Atomic Register algorithm (Read-Impose Write-Consult-Majority). However, it is not applicable for the CAS operation because CAS operation assumes "firstly compare with the old value and if different then swap it with the new one" is an atomic operation. It requires replicated state machine and Read-Impose Write-Consult-Majority algorithm is not strong enough to support the atomic CAS operation.

Therefore, seqeuce paxos [2] is used in the project as the algorithm to ensure the consensus regarding to different operations. A list of operations is needed to represent the replicated state machine, which means that every round one operation is executed atomically and each replica holds the same operation list.

Besides, the ballot leader election [3] is chosen to reduce the messages sent during propose phase. Firstly, the topology is initialized for the ballot leader election and after that, for each period of time, each server will send a heartbeat request to the others and if the number of heartbeat responses is larger than majority, it should check the leader and select the one with highest ballot (round number-pid pair) as the leader. In that case, only the leader receiving the GET/PUT/CAS operations can propose and not everyone proposes, which cost fewer messages in transmit.

The eventually perfect failure detector[4] is also implemented in this project. It will output log information that someone is suspected to crash when one server crashes or the delivery of the heartbeat of that server exceeds the delay in configuration file set initially. When the server just some undergoes some time delay, the failure detector component should also output log that the server is no longer suspected and increase the time period it need to receive the heartbeats from serves.

In this project partitioning (split-brain) problem (use time-lease to solve) is solved by using the quorum mechanism. The split-brain problem happens when a partition is divided into two smaller partitions when each small partition elects its own leader. Think about we have 3 nodes and are divided into 1 + 2 nodes. Quorum mechanism ensures that one leader will be elected only when it receives acknowledgement from more than half of all nodes, namely two acknowledgement messages. That ensures that only the small partition with 2 nodes can have a leader in future. That means 3 nodes have a resilience of 1 (resilience means the maximum number of failure nodes). When it comes to even number of nodes 4, quorum mechanism remains feasible with a resilience of 1. Therefore, it is recommended to set the number of the replication set to be an odd number.

Another problem is that if the leader falsely dies and recovers (usually because of delay of time) and it thinks itself is a leader but other two nodes may have elected a new leader. In the project, the ballot leader election ensures that will not happen because it will check the leader and find its ballot number to be smaller than the others. The ballot number is the similar idea as the controller_epoch in Kafka.

3.2 Key-Value GET/PUT/CAS

As the system is an in-memory store. We can create a HashMap for the key-value pairs in the code to store the data. GET operation is to read the corresponding value of the key. PUT operation is to update the corresponding value of a key with a new value. CAS (Compare and Swap) operation is to compare the previously read value with the current value and if there is a match, then swap the current value with the new one. Think about one scenario when a user firstly executed GET and then PUT a new value which is 1 larger than the GET value to the same key individually. Another user executes PUT between the two operations performed by the previous user. Then there should be some dirty data because the previous user does not know that the value he reads

initially is changed before he execute PUT. In such case, CAS is provided to ensure that the previous user indeed changed the value he would like to change atomically and prevent the write from other processes between read and write.

4 Test Result

The operation test with 5 single tests is coded to test the correctness of the GET/PUT/CAS operations. The first test is to test the PUT operation, when we put ten new key value pairs into the KV store and get the response of the new values and the new values should be the values that we insert in. The second test is to test the GET operation by initializing KV using the PUT operation tested before. First some KV pairs are initialized in KV store and GET operation should return the new values. Other three tests are to test the CAS operations with regarding to the three scenarios: no key found, key found but old value not match, key found and old value match. These three scenarios should return a None, old value and a new value correspondingly. Some tricky problems may still happen although we guarantee consensus of replicas and atomic CAS. I think it may be due to Sequence Paxos assumes a FIFO link but the net we use may not turn out to be that case. For example, in the third test we firstly use the PUT to initialize and then CAS to edit the values. If FIFO is not permitted, it may happen that CAS is performed before the PUT so our CAS will not ever perform! In that case, an operation list, in which operation is executed after the response of previous one is received, is used in the test to ensure that it performs in the FIFO order.

5 Outlook

Improvements can be implemented such that the key-space should be arranged to different partitions in the future. Routing mechanism should also be implemented to support lookup for different partitions.

Actually, the failure detector does not play an important role in this project and it only output some log information if some nodes is crashed or is suspected to crash. Maybe it can be customized to more tricky problems.

References

[1] "Kompics," https://kompics.github.io/kompics-scala/tutorial/basics.html, accessed March 22, 2021.

[2] "Sequencepaxos," https://courses.edx.org/asset-v1:KTHx+ID2203.2x+2016T4+type@asset+block@sequence-paxos.pdf, accessed March 25, 2021.

[3] "Ballotleaderelection," https://courses.edx.org/asset-v1:KTHx+ID2203.2x+2016T4+type@asset+block@ble.pdf, accessed March 25, 2021.

[4] "Eventuallyperfectfailuredetector," https://courses.edx.org/asset-v1:KTHx+ID2203.1x+2016T3+type@asset+block@epfd.pdf, accessed March 25, 2021.