Sharded Distributed KV Store with Raft

Chengyang Huang
chehuang@kth.se

March 27, 2022

1 Introduction

Consensus is an important problem in distributed systems, applications like database, blockchain requires consensus to coordinate between servers and tolerate different kinds of failures. In this project, I solved consensus by implementing the Raft consensus protocol [6] and built a sharded key-value store based on it. To validate correctness, I imported some test cases from [7], which focused on checking safety and linearizability. In addition, I have also designed some extra tests focusing on liveness and performance. As a result, this implementation ensures safety and liveness within 512 executions of all test cases.

The system consists of a config center to maintain the sharding rules and a sharded raft cluster for data storage, both the config center and each shard is a Raft cluster with at least three members. The architecture is shown as follows.

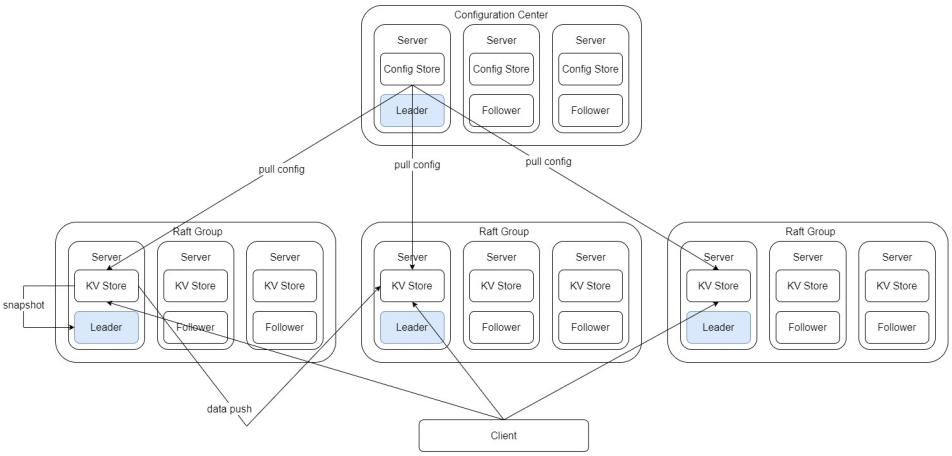

Figure 1: Architecture

2 Raft Consensus Algorithm

2.1 Leader Election

In Raft, each node is always in one of these three roles, leader, follower, and candidate. Node becomes follower when it receives message from node with a higher term, it becomes candidate when it doesn't receive message from leader for some time, it becomes leader when he received votes from a quorum (by proactively sending out requests and check the response). Candidates are not very different from followers, they all wait and respond to those who thought they are leaders, except for candidates send out requests for voting periodically.

By design, Raft requires at most one leader in any term, any proposal should be first handled by the leader, this avoids the synchronization phase (query phase in vanilla Paxos) before log replication, which reduces latency. On the other hand, single leader becomes the performance bottleneck of the system. I used sharding on application layer to solve this. To grant vote to a vote-requesting request, the peer must have a term larger of equal to current node, and has more up-to-date log than current node (by comparing the proposed term and log index of the last node), this ensures the leader has all commited logs.

2.2 Log Replication

Log replication is done from leader by broadcasting logs that have been proposed but not yet committed. It is possible that the receiver's find its log different from that sent from the leader, the receiver will then reply by asking the leader to adjust its hypothesis about the receiver's received log, and then resend logs with smaller index in log sequence. Also, the broadcast will carry the message of leader's committed index, so the receiver could decided on the committed index.

2.3 Snapshot

When cluster membership changes or connection to some server is slower, leaders may help them to catch up by sending a batch of log sequence in the log replication RPC, but the message body, which mostly consists of logs, could possibly be extremely large, especially in the case of newly added nodes. In this case, the application may periodically initiate a snapshot by sending Raft layer its application layer snapshot and the log index this snapshot corresponds to. The Raft layer would then create a consistent snapshot for both the application(RSM) and the Raft logs. This also could be seen as a garbage collection action to remove the logs that are no longer used. If leader wants to send logs to fall-behind nodes, and the log it requires is cleared by snaphotting, it will send the snapshot instead.

3 Sharded Key-Value Service

Based on Raft algorithm, I built a replicated state machine which serves as a sharded key-value service. Committed logs are asynchronously sent to the RSM to update its state. Supported operations include get, set, and append, it would be easy to expand to more operations like cas, lock, etc. The application periodically fetches configuration from a configuration service, which is also implemented with a replicated state machine to ensure better availability.

I implemented shard migration with a level-triggered approach, which periodically checks if current stored data does not conform with most recent configuration. The application pushes shard not belonging to it to the node it believes this shard belongs to. The receiver also checks if itself is waiting for this shard. Nodes will not apply new configuration until its data conforms with current configuration, by using this heuristic, it could be seen that no shard will be discarded. Though, I believe this part could be further optimized, especially when facing frequent configuration updates.

As a result, a sharded server consists of three threads, one gets and applies logs from underlying consensus algorithm, one pulls latest configuration periodically, and one tries to push and wait for shards to be aligned with current configuration.

4 Testing

Testing is critical in ensuring the correctness of the implementation. It should be done at the same time of development to minimize the range we search for bugs. In this section, I will introduce how my testing is performed.

To clarify, many of the test cases are imported from [7], it simulates the persistent storage and offers an atomic file writing interface to the program, so program does not need to consider partial write problem. Also, the testing framework simulates a network using channels in Golang, so the user may easily enable package reordering, delaying or dropping in test cases. The framework also provides some test cases, which I think are really comprehensive on checking safety and liveness, but not focusing on performance. I've added several test cases to show performance, and several test cases about liveness that is not covered by the imported test cases.

4.1 Debugging

Debugging a multi-threaded program is more complicated than single threaded program, using a debugger to execute step by step would no longer bu sufficient. Most of the time, I use logging to trace problems.

The following line of log comes from a real execution in Raft layer, from left to right, what the log means is as follows.

At time-point 52524ms since the beginning of testing, inside sub-cluster 102, which serves part of the shards, the Raft instance No.2 is currently a leader at term 2, the committed log index is now 35, the term and index of last log entry is 2 and 35, the snapshot is generated up to term and index of 2 and 32, log size in memory is 3, hypothesis of other member's latest log index in this sub-cluster is respectively 36, 35, and 7, the last log applied by the RSM is 32, the node is the second node in the sub-cluster. In the log body, it shows that the node is sending a append entry RPC with request id 62157.

```
[52524 102-2 Term: 2, Role: leader, cmt: 35, lastTerm/Idx:2-35,
    snapTerm/Idx:2-32 LogL: 3, Next:[36 35 7], lA:32 Id: 2]
   >>> sending req[62157] append[true] vote[false] snap[false]
```

Another line of log from real execution I would like to show is from the application layer, the meaning is as follows.

At time-point 52554ms, application instance with ID "Oatu" has applied log entries up to index of 121, its current configuration term is 3, current shard migration sequence number is 6. The application currently has four shards in memory and shard number 4 does not belong to the node according to its current configuration. The group number and sub-cluster node number is respectively 100 and 1. In the log body, it shows that the node is pushing shard number 4 with shard migration sequence id 6 to group 101, sub-cluster node number 0.

```
[52554 Oatu Idx:121 Cfg:3 Mv:6 state: map[0:serving 1:serving
   2:serving 4:leaving]] G-S:[100:1] >>>pushing shard[4] seq
   [6] to[101:server-101-0] pushId:100-1-push:6
```

4.2 Correctness

4.2.1 Consensus

Test cases on consensus is the core part of testing, I exposed several interfaces for the testing codes so it could monitor the internal state like role, term and logs, also, testing code can serve as the application layer to intercept the committed logs by Raft.

Test cases consists of several parts, I will introduce the most complicated one in each category:

1. **Leader Election**: Test is performed on a 7 nodes Raft cluster, in each iteration, a random minority of nodes will not partitioned from other nodes, then we will check if there is one and only one leader among nodes who are not partitioned, and there is no two leaders in the same term. This iteration repeats 10 times in each execution.

2. **Concurrent Proposal**: Test is performed on a 3 nodes Raft cluster, 5 clients propose values concurrently, we wait to see if all the proposed value are eventually committed.

3. **Failure Recovery**: Intentionally crash a quorum of nodes including current leader, then bring back the leader to see if later proposed value could be correctly committed.

4. **Log Overwrite**: Propose many values to some leaders that is not connected to a quorum, and later recover the network and propose values to another leader to see if logs in previous leaders that are not committed could be correctly overwritten.

5. **Snapshot**: Serially propose 300 log entries to the cluster, with possibility to shutdown random node(but keeps a quorum alive) after each proposed value is committed. The simulated network channel also delays, reorders or drops messages randomly. The test case will check if log sequence is too long(meaning snapshot is not performed in time) or any proposed value is not committed.

4.2.2 Key-Value Service

1. **Not Sharded**: Only puts one raft group to the configuration to simulate the not sharded situation. Keep proposing values to the leader while also enables node crashing, message delaying, reordering and dropping. This is to ensure the implementaion without sharding is correct.

2. **Serial Proposal**: This is one of the test cases I added. This test case is to test the situation when client need to see the effect of previous value before proposing the next value. If leader crashes before committing the old value and returns as leader in a new term, and no other logs are proposed, then the old value may never be committed, which is a liveness problem. The solution mentioned in [5] is to propose an empty log each time when new leader is elected.

3. **Concurrent Proposal**: Concurrently proposing values in application layer to see if all the values are eventually applied to the state machine.

4. **Shraded Group Join and Leave**: Modify the configuration of sharding to see if the shrads are correctly migrated.

5. **Partial Migration**: Modify the configuration of sharding while also proposing new values to the cluster to see if requests on shards that are not affected by the configuration change can still be handled when other shards are migrating.

6. **Linearizability**: For Linearizability checking, I used [1], which implemented [3], an optimized algorithm compared to [4]. To use this library, we need to offer it with a structure array that consists of operation input, invoke time, operation output, and response time. Also

we need to implement some interface to tell the library how the state of the state machine is transferred on each operation, and how to partition the operations on different registers. I did not delve into the detail of the algorithm, but I could see from its source code that it uses the partition function we provided to separate the operations, since linearizability is a compositional property, this helps to reduce the complexity of the problem.

4.3 Performance

Following the performance report from etcd [2], I designed some performance tests to show performance of the cluster. The test was performed on my personal computer with CPU of 8 cores 16 threads and 64GB memory, Go version 1.17.5, GOMAXPROCS is set to 8. All Raft sub-cluster has 3 nodes. Each client proposes 1000 values, the testing ends until all proposed values are committed.

Setting	Client Number	QPS	Latency(ms)
Serial Read Write	1	33	30.1
Concurrent Proposal	100	1890	52.9
Sharded Concurrent Proposal	100	1980	50.5

Table 1: Testing of performance under different scenarios

It can be seen that the QPS of concurrent proposal is 57 times of the serial case, while the latency is 1.76 times of the serial case. My hypothesis on the reason why performance in sharded scenario is not significantly better than the non-sharded case is that the test is mostly computational intensive and I have observed a CPU usage of 100%, which indicates that the CPU becomes the bottleneck.

5 Conclusions

In this project, I have implemented a sharded key-value store using Raft consensus algorithm, tested the system from varies layer and scenarios on safety, liveness and performance. The most challenging part of this project would be debugging in a high concurrency scenario. I have learned to design the log in a clear format and to find the problem from millions lines of logs. Solving the deadlock and livelock problem is also very interesting.

There are still plenty of features and tricks that I don't have time to implement.

1. Implementing membership change algorithm to allow nodes join or leave.

2. Implementing PreVote and CheckQuorum [5] to ensure liveness under partial connected network environment.

3. Introduce Learner role to first catch up with the leader before been promoted follower to reduce the unavailable time during membership reconfiguration.

4. Introduce a storage engine to really persistent the data, rather than calling the persistent API provided by the testing framework to simulate this.

5. Testing on multiple machines to see if there are problems that is not manifested on a single machine.

The source code of this project could be found here.

References

[1] Anish Athalye. Porcupine: A fast linearizability checker in Go. https://github.com/anishathalye/porcupine, 2017.

[2] CoreOS. etcd. https://etcd.io/, 2013.

[3] Alex Horn and Daniel Kroening. Faster Linearizability Checking via P-Compositionality. In Susanne Graf and Mahesh Viswanathan, editors, *35th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, volume LNCS-9039 of *Formal Techniques for Distributed Objects, Components, and Systems*, pages 50–65, Grenoble, France, June 2015. Springer International Publishing. Part 1: Ensuring Properties of Distributed Systems.

[4] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29, 12 2016.

[5] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford, CA, USA, 2014. AAI28121474.

[6] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[7] Robert Morris. 6.824: Distributed systems. http://nil.csail.mit.edu/6.824/2018/, 2018.