1 Introduction

This project extend the Raft-based key/value service implemented in NYU Distributed Systems Class (FALL 2018)[6] Lab2. These extensions are essential when building a system for production use. Specifically, the project consist of four extensions:

1. Add snapshot support, which provide log compaction mechanism. The basic algorithm is described in Raft tech report [2], Section 7.
2. Shard our key/value service to provide a better throughput/performance. Each shard is a subset of key/value pairs; Each replica group handles requests for just a few of the shards, and all the groups operate in parallel. In this project we named it ShardKv service.
3. Support shard reconfiguration. This is done by implementing a configuration Master service, which named ShardMaster in my implementation. The ShardMaster use the same idea of BigTable [1], and the clients/ShardKvs would query the master node for configurations. We guarantee the consistency on concurrent shard migration and key/value operations, and also we don't pause the key/value operations on unaffected keys due to the consideration on the performance requirement on a real world system.
4. Supports cluster membership change. This project also provide the simplest cluster membership change operation - each time add a single node or remove a single node from the current view. This simplifies the scenario mentioned in Raft tech report[2], Secion 6, thus we use a slightly different method to commit a view change.

The project's basic idea is brought from NYU Distributed Systems Class (Fall2017): Lab4 [4], Lab5 and Challenges [5]. Though in this project we made a brand new implementation based on our new framework (i.e. Lab2 starter code 2018), as well as the clients and the tests. Meanwhile we also added cluster membership change support to make it more applicable to real world applications.

2 Motivation

As mentioned in the class, it is hard and error-prone to implement every services directly with the consensus protocols. A more practical way would be implement a distributed key/value service and build other services on the top of that. In order to satisfy various of needs of efficiency and scalability from different applications, such problems are essential to be solved in our distributed key/value service:

- Raft logs would grow without bounds if there's no such log compaction mechanism implemented. In real production, it's not practical for a long-running server to remember the complete Raft log forever [4], because this would occupy too much storage on the servers, and it would take a long time for a rebooting server to replay the whole log history. The snapshot described in Raft paper[2] Section 7 can solve this problem - the restarted server can install a snapshot and then replays log entries from after the snapshot point, but not the whole log history.
- Shard the keys in to groups can provide us a better throughput/performance. Each replica group handles requests for just a few of the shards, and all the groups operate in parallel. Thus the total system throughput increases roughly in proportion to the number of groups (if the workloads are balanced).
- Shard reconfiguration is also important for a production system: The workload on each shard could change at times do due to varies of reasons. Some shards may become more loaded than others, so that shards need to be moved across the ShardKv groups to balance the load.

- Finally, view change is another brick to make the system applicable to the real world problems. The peers may join and leave the system - new peer may be added to increase capacity, or existing peer may be taken offline for repair or retirement.

By implementing these extensions into our system, we step towards building a key/value service that could support production use cases.

3 Design and Implementation

3.1 Snapshot

The snapshot mechanism has already been described well in Raft report [2]. In this project we simplify it by using a single chunk of snapshot instead of sending series of chunks. Figure 1 generally shows the snapshot

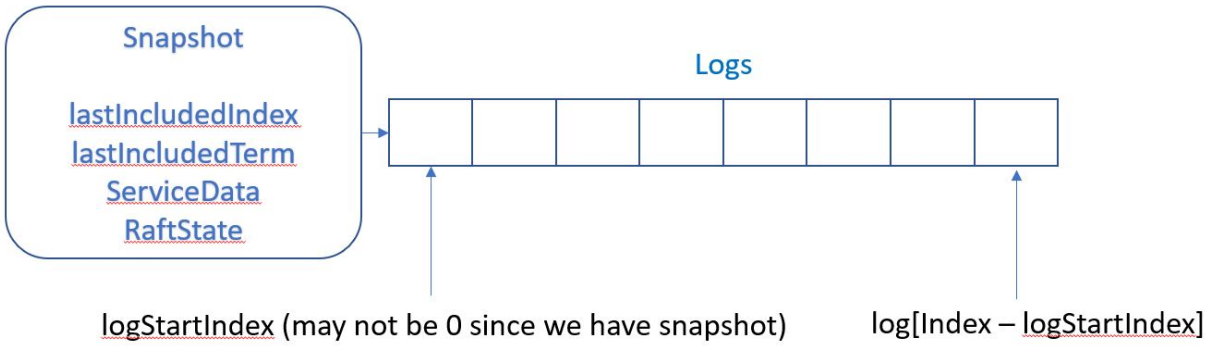

Figure 1: Snapshot Illustration

implemented in this project. The $ServiceData$ and $RaftState$ are encoded using package $encoder/gob$ to make the implementation more portable. All the servers will create a snapshot if the current number of applied logs grows larger than a certain threshold. An additional pointer $logStartIndex$ is maintained as an offset to calculate the "physical" array index from the theoretical log index, because the new log may not start from log 0 as we created the snapshot and discarded the packed ones.

3.1.1 Leader behavior on InstallShapshot RPC

The Leader may use $InstallShapShot$ RPC instead of $AppendEntries$ RPC when $nextIndex[peer]$ is smaller than $logStartIndex$. Obviously that's because we don't have this log entry anymore so we cannot send it. Then the leader will send the whole snapshot instead.

3.1.2 Follower behavior on InstallShapshot RPC

The receiver's reaction on $InstallShapshot$ is clearly listed in Raft paper, Section 7, though we can skip step2,3,4,5 because our tiny system maintains everything in memory and don't have the real disk files.

3.2 Shard KeyValue Service

Generally we implement our sharded key/value service with two components: a ShardMaster service and several ShardKv services. Each of them are built on the top of our Raft implementation in Lab2, thus our model will result in a single ShardMaster cluster and several shard group clusters. This structure provide both fault tolerance on each of the services.

Meanwhile currently in this project we simply the problem by using a single key as a shard. However this can be extended with the improvements as follows:

- Add a key-shard mapping function
- Transfer the whole shard instead of a single key between shard groups

3.2.1 ShardMaster

The ShardMaster service is responsible for:

1. Maintain a global view of current shard-group mapping. The client will retrieve this info using $GetKeyGroup$ RPC to know which group is currently serving the target key.
2. Create and maintain an array of $Reconfig$, where each item is a 3-tuple: $(Key, SrcGroup, DstGroup)$. The meaning of a tuple is "move the Key from $SrcGroup$ to $DstGroup$". The array index is also served as a unique $ReconfigId$.
3. Guarantee that all the $Reconfig$ items are valid operation. i.e.:
   - Key_k must belongs to $SrcGroup_k$ right after the $Reconfig_{k-1}$
   - $SrcGroup_k \neq DstGroup_k$

   Since the client command don't need to give out the $SrcGroup$ (RPC details below). The first condition is inferred by the ShardMaster itself. However the second rule need to be checked by the ShardMaster. In the case client send a command to assign a key to the group that it already belongs to, the ShardMaster will reply with $Success$ without creating any new $Reconfig$ items.

Correspondingly, the ShardMaster service provide 3 RPCs:

- $GetKeyGroup(Key)$: The clients will call this function to query which Group is now serving this key. Returns the $GroupId$.
- $Reconfig(Key, DstGroup)$: The clients will call this function to re-assign Key to the $DstGroup$. Note that at the time this function returns, it is only guaranteed that this is committed on the ShardMaster. The real time point that key migration happens depends on the shard groups (i.e. ShardKv services).
- $GetReconfig(ReconfigId)$: Differ from previous two, this RPC is called by each shard group to get a $Reconfig$ entry from the ShardMaster. The ShardMaster will simply return the corresponding $Reconfig$ item if the parameter is a valid index. A special situation is that the shard group's configuration is already up-to-date and request a $ReconfigId$ that out of bound. In this case the ShardMaster will simply return a $Success$ message.

3.2.2 ShardKv

The ShardKv Service is mostly built based on the original KVStore service. In addition to the storage of the key/value pairs, now it maintains an extra $currConfig$ and a $valid[]$ marker map. The $currConfig$ stores an $ReconfigId$ which indicate the latest re-config that it committed and applied. The $valid[]$ is a simply marker which stores whether the current group is serving a certain key. With these extra fields, we have updated/added RPCs as follows:

- $Get/Set/CAS$: The service now will check if the current operation is on a key k where $valid[k] = true$. If this doesn't holds, the return value will be a $NotResponsible$ message.
- $KeyMigration(ReconfigId, Key)$: This RPC is used internally between the shard groups to really finish a $Reconfig$ operation. The callee will first check if the $currConfig$ is not less than $ReconfigId$. If it holds the callee will return the desired key/value pair to the caller.
- $CheckCommit(ReconfigId)$: This RPC is used internally between the shard groups for garbage collection. In the implemented below the sender need to query the receiver to check if it can safely remove the disabled key in its' store. This RPC simply returns whether $ReconfigId \leq currConfig$.

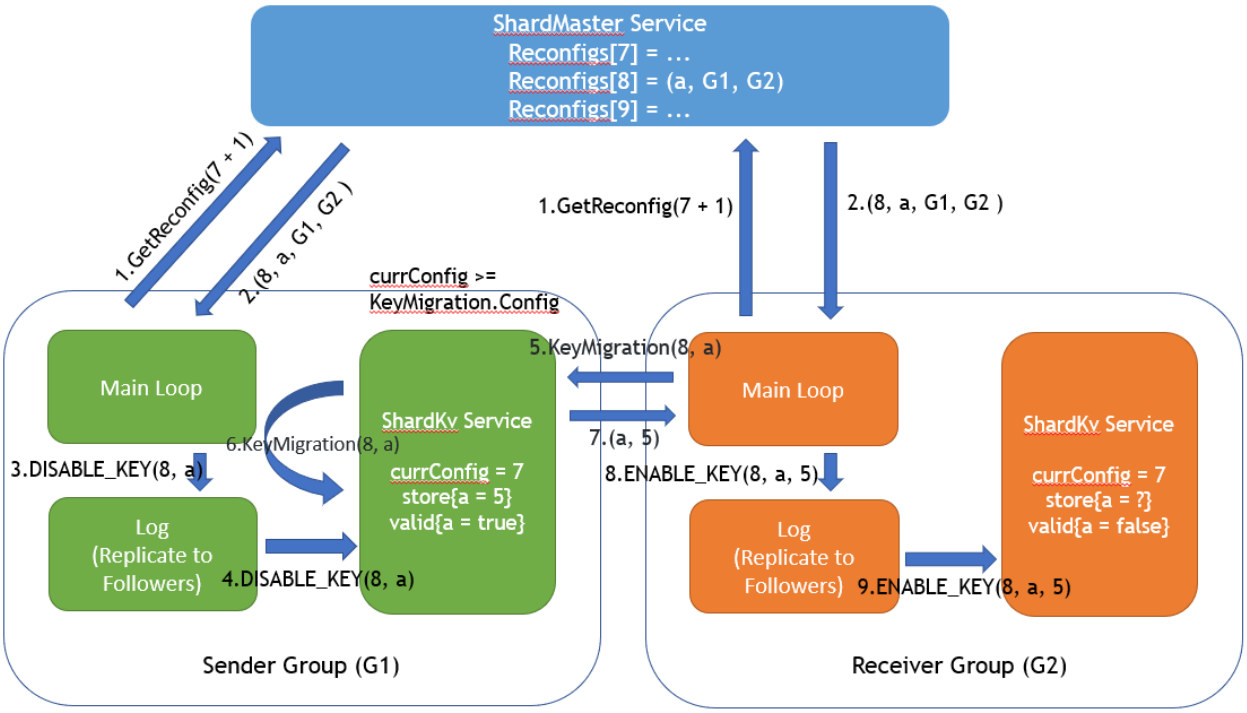

Figure 2: Shard Migration Illustration

3.3 Shard Migration Mechanism

Figure 2 illustrates an example of the work flow of how a re-config is applied in the system. Here explain all the steps in this figure:

1. Heartbeat timer for re-config time out. Both sender group(G1) and receiver group(G2) **leaders** query the ShardMaster Service for the $Reconfig$ entry at the index $currConfig + 1$.
2. The ShardMaster Service reply with $(8, a, G1, G2)$, which means: move key a from $Group1$ to $Group2$.
3. The sender group got this entry, and realized that it is the sender. The leader then append a DISABLE_KEY entry into the log.
4. After the DISABLE_KEY entry was replicated and committed, the ShardKv service execute it and set $valid[a]$ to $false$. Meanwhile it also increase $currConfig$.
5. On the receiver side, the receiver leader may query the sender group with $KeyMigration(8, a)$ to get the corresponding value of key a.
6. The sender receive the request and replicate/commit the log.
7. The sender check if the $currConfig$ is not less than $KeyMigration.ConfigId$. If it holds then return the key/value pair. Returns an empty message other wise.
8. The receiver get the value of a and append a ENABLE_KEY entry in the log
9. The ShardKv service on the receiver side execute the command, set $valid[a]$ to $true$ and increase the $currConfig$

Note that a group could be neither the sender group nor the receiver group(i.e. the $GroupId$ is not appeared in the $Reconfig$ entry). In this case the leader just append a UPDATE_CONFIG command to the log, which simply increase the $currConfig$ when executed. At last, the sender need to use $CheckCommit$ RPC to check whether it's safe to remove the migrated key from it's store.

3.3.1 Safety

The core safety requirement is that our protocol should guarantee there's no data lost during the shard migration. It is equivalent to the following condition: The sender should have stopped serving the key at the time receiver got the corresponding value of that key. Otherwise the receiver will get an out-dated value which break the consistency.

In our implementation this condition is guaranteed by the checker on the $KeyMigration$ RPC: When $currConfig \geq KeyMigration.ConfigId$ holds, it implies that the sender group has already committed the DISABLE_KEY command for this configuration change, which means the sender has already stopped serving this key. Note that this implies there could be a period the key is not served by any groups (i.e. after sender commit but before the receiver commit).

3.3.2 Trade-offs

In my implementation I went with the receiver pull method (i.e. the receiver will call the $KeyMigration$ on the sender to retrieve the desired key. However the sender push method may also works. To do this the sender may need to periodically push the key/value pair to the receiver until the receiver confirm that the reconfiguration has been committed. This implies an additional heartbeat cycle on trying to send out the key/value pairs. However this method is easier to implement garbage collection - the sender can clean up the store right after the receiver replies that the entry has already been committed. In my opinion there's no clear judgment that one way is better than the other, because they both need to finish a whole round of message exchange.

Another trade-off was made on the shard migration mechanism is consistency vs availability. In our implementation we guarantee the consistency with the safety condition above, result in a time period that no groups is serving for the migrating key. However if consistency is not strictly need on key migrations then we could have the sender continue to serve for a while until the receiver told the sender that the reconfiguration has been committed. In this case the receiver will get the out-dated values - i.e. the operations happened after shard migration begins has been lost. In my implementation I choose the consistency rather than availability because it meets the same lineraizable requirement on Lab2.

3.4 Cluster Membership Change

In this project I use a rather simpler model than the one mentioned in the Raft paper: This implementation only support one node change each time. i.e. A command could be either $Join(peer)$ or $Leave(peer)$. The main reason is that by restricting our model it would be easier to implement the "joint consensus", though these two commands are already powerful enough to constitute more complex membership changes.

To support membership change, the raft need to maintain an extra state $currView$, which will also need to be encoded in the snapshot if it is enabled. Any messages from a peer that not in $currView$ will be ignored. After the leader receive a $Join$ or a $Leave$ command, it will treat it just like a normal command - append it into the log and replicate it, until commit time. At commit time, the server will check if the currently committing command is a $Join$ or $Leave$. If so, it would do the following updates on the raft states:

1. Update $currView$
2. Update quorum size. **Every logs following the view change log** should use the new quorum size.
3. The leader should step off if itself is the one to be removed from the cluster.

3.4.1 Safety

In this section we argues that the quorum size needed to commit a view change command is the same as the old view, by proving the following: Any of the majority in the old view have a intersection with any of the majority in the new view - then we can follow the same logic of "joint consensus".

Assuming that the command is a $Join$:

- If the number of peers in the old view is a odd number N, then the quorum size in the old view is $(N+1)/2$, the quorum size in the new view is $(N+1)/2+1$. So a majority comes from the old view plus a majority comes from the new view would have $N+2$ nodes. Note that we would have only $N+1$ nodes after view change, thus there must be a intersection.
- If the number of peers in the old view is a even number N, then the quorum size in the old view is $N/2+1$, the quorum size in the new view is also $N/2+1$. So a majority comes from the old view plus a majority comes from the new view would still have $N+2$ nodes.

A similar proof can be made on $Leave$ command - we can prove that there's at least N nodes by combining the majority, after we've already subtracted 1 from it because one vote from the old view could be made by the leaving node. And we will have only $N-1$ nodes in the new view.

4 Validation Methods

In this project I majorly use one or multiple long-run client to check whether there are inconsistency or data lost. Specifically, the client may:

1. Loop forever
2. Constantly Set/Get/CAS on the same key using different values.
3. Automatically query the ShardMaster for the group assignment, if got a $NotResponsible$ response.
4. Automatically redirect to the new server, if got a $Redirect$ response
5. Automatically retry on the next server in the server list, if the RPC returns an error(probably means that the server failed in this case)
6. Re-try the last command when 3,4 or 5 happens.
7. If the RPC successfully returns, check the correctness of the response.

Meanwhile, by manually or automatically, we try to make the cluster unstable by randomly apply the following operations (Of course the operations are legal and didn't make a majority fail):

1. Kill a server
2. Launch a server
3. Move a shard from one group to the other, by making a RPC to the ShardMaster
4. Change the cluster view, by making a $Join$ or $Leave$ RPC to a replica group or the ShardMaster.

Beside the long-run client test above, I also have several unit tests that was used in Lab2. Especially note that the clients above will each work on a different key (otherwise we cannot check the result), thus additional test cases to check the contention situation are needed. To validate this I wrote a test case which simply fires bunch of CAS operation on the same key, and check: 1. If the program encounter any crashes/deadlocks 2. If the result is correct - only one CAS should be a success CAS operation.

5 Related Works

The implementation of Snapshot mechanism is based on Raft paper[2], Section 7. The general sharding design pattern is similar with BigTable [1], Apache HBase [3] and many other distributed system design - it is comprised of a configuration service and a set of replica groups. The basic project idea is brought from Lab4 [4], Lab5 and Challenges [5] from NYU Distributed Systems Class (Fall2017). The code skeleton is based on the Lab2 starter code [7] from NYU Distributed Systems Class(Fall2018) [6]

6 Challenges

This section briefly list the challenges/difficulties I met when working on this project...

- This is indeed a "implementation heavly" project...
  - More than 2K lines of not duplicate code in a new learned language (Go)
  - Need to make it almost bug free in a distributed environment. And later on we even have multiple clusters.
- Got lost on shard moving mechanism for a while
  - Initially didn't figure out what is "correctness/safety" for a shard system
  - Later on realized that a trade off has to be made between consistency and the availability during a shard migration
- Need to think much about the details and safety on our own.

7 Acknowledgement

The author would like to thank Professor Aurojit Panda for the kind helps given on this project.

References

[1] Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. *Bigtable: A distributed storage system for structured data*. ACM Transactions on Computer Systems (TOCS) 26, no. 2 (2008): 4

[2] Ongaro, Diego, and John K. Ousterhout. *In search of an understandable consensus algorithm*. USENIX Annual Technical Conference. 2014.

[3] Apache HBase: https://hbase.apache.org/

[4] http://www.news.cs.nyu.edu/ jinyang/fa17-ds/labs/lab-kvraft.html

[5] http://www.news.cs.nyu.edu/ jinyang/fa17-ds/labs/lab-shard.html

[6] https://cs.nyu.edu/courses/fall18/CSCI-GA.3033-002/

[7] https://github.com/nyu-distributed-systems-fa18