Building a Sharded, Replicated Key/Value Store Service with Strong Consistency Guarantee

Xiang Gu
EID: xg2847
Click Here For Code Repo

May 18, 2021

Abstract

I followed labs from an online open course on distributed systems [1] and built a sharded, replicated key/value store service with strong consistency guarantee. This report details the problems, ideas, architectures, test results, and lessons learned from this building-from-scratch process.

1 Introduction

Distributed systems have become increasingly popular in practice rather than an academic curiosity. Most medium or larger size companies, organizations, institutions built their service as distributed systems to provide high availability. But learning and writing distributed system code is hard, even for seasoned engineers. This report is a summary of my first serious endeavor into this field – building a distributed key/value store service from scratch.

The report is organized as follows: Section 2 gives the client interface and system properties of our service. Section 3 dives into the nitty-gritty of the implementation by breaking down the problem into three sub-problems and discusses how to tackle them one by one. Section 4 gives screenshot of running provided tests. Section 5 offers some of the useful lessons we learned which we deemed transferable onto building other distributed systems as well. Section 6 briefly mentions the necessary additional optimizations we need to do towards an industrial strength key/value store service (such as ETCD). Finally, section 7 summarizes the report.

2 Client Interface and System Properties

Our service is a string-to-string key/value service supporting the following three client requests:

1. $Get(k\ string)$: Get the value of the input key k from the store; if key is not present, return an empty string;
2. $Put(k, v\ string)$: Update the value of the input key k to v. If key is not present, this will create an entry for k with value v.
3. $Append(k, v\ string)$: Append v to the value of key k. If key is not present, this will create an entry for k with value v.

Those three requests are guaranteed to have taken effect upon return (so clients don't need to worry about, say, retries). Moreover, our service provides strong consistency (aka linearizability) to clients. That is, a total ordering of (possibly concurrent) client requests exists such that 1). ordering of non-concurrent requests is honored in this total ordering, and 2). every $Get(k)$ sees result of the most recent $Put(k, v)/Append(k, v)$ in this total ordering.

Our system is fault-tolerant against crash faults. It continues to operate as long as a majority of servers are up and working. When crashed servers restarts, they can rejoin the cluster and continue to serve client requests without manual intervention.

Our system is sharded. Sharding is a logical partition of the workload by their request keys (e.g. all requests with a key starting with letter ”a” form the first shard, ”b” for the second shard, etc, partitioning all client requests into 26 groups); The idea is that since sharding partitions client requests into disjoint groups (also referred to as shards), we can process requests in each shard in parallel to increase throughput. A physical server can be responsible for processing workloads from one or more shards, in which case we say those shards are assigned to this server [1]. The assignment from shards to servers is also known as a $configuration$.
On that front, we created another service, ShardMaster, with another set of client interface that allows clients (usually system administrators) to dynamically scale up (resp. down) the key/value store service throughput by adding (resp. removing) servers to the cluster. They are

1. $Join(servers\ map[int][]string)$: It takes as input a mapping from group IDs to server names in that group, add it to the cluster, remove shard(s) from some existing servers and assign them to those new servers in a evenly loaded way. It also automatically transfers the key/value pairs within those shards to those new servers because they are now serving requests in those shards.

2. $Leave(GIDs\ []int)$: Removes servers with those group IDs from the cluster. Shards previously processed by those groups will be again evenly distributed to remaining groups and the shards data will automatically be transferred as well.

3. $Move(shard\, int,\ GID\, int)$: Moves shard *shard* to group *GID*.

4. $Query(num\ int)$: Returns information of the *num*-th configuration[2].

[1]Instead of one physical server, it’s actually a group of servers, serving as replica of each other, for fault-tolerance purpose.

[2]Configurations are numbered from 1 and increase monotonically.

Our service works with a fair-loss network where packets might drop, duplicate, and reorder.

3 Method: A Three Step Approach

Implementing such a sharded, replicated service might sound daunting; We break it down to a three step approach where we build one thing atop the previous one toward the final goal. The plan is to first build a replicated state machine (RSM) using Raft [2], then add client/server code for a key/value service which uses the RSM to agree on the sequence of client requests to apply to the state machine, and finally create the ShardMaster service (which itself is also a replicated service so its architecture resembles the one we built in the previous step) as well as adding logic to our key/value service to support dynamic configuration changes.

The following three sub-sections detail the three steps one by one. We also provide a screenshot of running the course-provided testing suite against our implementation.

3.1 Raft RSM and Log Compaction

This part is actually project 1 of this course so I will briefly go over it. Several servers forms a cluster and users can call the $Start(op\ Op)$ interface on the leader and the leader will append this operation to its log and start to broadcast it to other follower peers. Upon receiving positive reply saying ”Okay, leader, I got it and also put that op at the same spot in my log” from a majority of servers, the leader (as well as other followers who have that operation in the same index in their log) pushes this operation into a channel $applyChan$, at which point we say that operation is *applied*. This makes a RSM possible because we can now apply operations coming out of this $applyChan$ one by one to our state machine, and rest assured that all servers in the group will eventually be in the same state because they process the same operations in the same order, hence serving as replicas of each other and providing fault tolerance. In the case of crash and restart, the server will read its persistent state from disk, re-push all the operations previously pushed to *applyChan* before the crash, and at the same time communicate with the leader to fill in its log all the operations it missed.

3.2 Replicated Key/Value Service with Log Compaction

We are ready to implement a key/value store service based on the Raft RSM. We add both client side and server side code for this part.

We begin with the client side logic, which is simpler and follows a pattern that will be repeatedly used in later parts. The main object in this part is a Clerk struct which is used by customers to invoke the abovementioned three requests:$Get(k)$, $Put(k, v)$, and $Append(k, v)$. We assume each clerk has at most one outstanding request at any point in time (meaning the current request must return before starting the next). There might be multiple clerks issuing requests at the same time though. Each clerk is uniquely identified by a *clerkId* and within each clerk, requests are also uniquely identified by *requestId* that increases monotonically. Each clerk also knows the endpoints for each servers in the cluster so as to send RPC requests to them.

The logic for those three requests follow the same pattern – **keep retrying until success**. The clerk will send the RPC request to the leader server (because in Raft RSM all client requests are processed by the leader). If it has not heard back from the leader after a certain period of time, or the server replied saying "I'm not the leader", then we will re-send RPC to the next server in the cluster, hoping this time this new server is the leader and reply with "okay, we've committed and applied this request to the state machine. Result is included in it too!", at which point the client-side implementation of the three requests can return.

On the server side, two main pieces are needed – the request RPC handler and a long-running thread $Run()$ that repeatedly reads from *applyChan* for committed requests and apply them to the state machine.

First and foremost, let's define what exactly is the state of our state machine in our key/value service. It consists of two parts: a simple map *kvStore* that stores all the key/value pairs and a map *maxAppliedOpIds* that keeps track of the maximum applied *requestId* for each *clerkId* (for deduplication reasons detailed later).

The $Run()$ thread has a very mechanized logic – **listen, apply, and notify**. Concretely, it listens to *applyChan* and reads a request once available, skip it if this request has been applied before using *maxAppliedOpIds* [3], apply the request to the state machine (i.e. update both *kvStore* and *maxAppliedOpIds*), and finally notify the waiting request handler if a leader.

The request handlers, for all three of them, also follows the same pattern: **replicate if leader and wait for success**. Namely, upon receiving a client request, it rejects with "I'm not the leader" if not the leader. Otherwise, it invokes the interface $Start(op)$ on the underlying Raft peer with *op* prepared for this request. It then waits for the notification from the $Run()$ thread telling it "this request has been committed and applied", at which time point the handler can safely return by replying to the clerk "okay, we've committed and applied this request to the state machine. Result is included in it too!"[4]. If the handler is not notified by $Run()$ after a certain period of time, it replies with "I'm the wrong leader" [5].

We mentioned above that we use *maxAppliedOpIds* in $Run()$ for de-duplication. You may wonder is it even possible to have the same request committed and hence pushed twice (or more times) to *applyChan* by the underlying Raft RSM. Yes, it is possible. Consider the case where one client request was committed and applied to the state machine but the leader crashes before replying to clerk; the client side hence times out and re-send the same request to another (new) leader who has this request committed to its log but not applied to

[3] Recall that requestId increases monotonically for each client so using *maxAppliedOpIds* allows us to filter out all requests with a $\leq$ requestId from the same client.

[4] Of course, only $Get(k)$'s need a result in their reply.

[5] This message is purely syntactical so the client can resend the request to other servers. Consider the case where the server is actually a leader in a minority partition and hence it can never commit this request. Such a timeout mechanism allows the client to retry to the leader in the majority partition

state machine yet (i.e. it's still awaiting process in *applyChan*); our implementation is that this new leader will blindly try to replicate this request to other follower peers, just like the previous leader did; This means the same request will be pushed to *applyChan* again.

We also implemented an optimization: log compaction. For any practical, long-running service based on Raft RSM, client requests in its servers' logs are ever accumulating. It is necessary to trim log entries that are no longer used to keep the log size manageable. But what if the server crashes after such a trim? It won't be able to restore to exactly the same pre-crash state by replaying all requests in log because some log entries are now missing.
The idea is to have application (in our case the key/value service) work with the Raft RSM. Namely, the application took a snapshot of its state (*kvStore* and *maxAppliedOpIds* in our case) after applying the i requests in its log, every once in a while. It hands over this snapshot to its underlying Raft peer so that the Raft peer can now trim all entries with index $\leq i$ and persist the new log as well as the snapshot to its disk. If the server crashes after this, upon restart, it will read from disk the trimmed log and the snapshot. The Raft peer first hands over to the application the snapshot via *applyChan* [6], which is used to restore the application state in *Run*(). Then the Raft peer starts to push all the operations in its log, starting from the $(i + 1)$ operation, to *applyChan*.
One additional complexity this optimization brings is the question : What if the leader trimmed all log entries with index $\leq i$ but a slow server (or a new server that just joins the cluster) is asking for a log entry that is trimmed [7]? Our solution, based on a vague description in section 7 in [2], is to have the leader send its snapshot of the application state to that follower so that that follower can "fast-forward" its application state to the snapshot and start to acquire log entries with index $\geq i + 1$.
Two things that require careful treatment are 1. if existing log entry has same index and term as snapshot's last included entry, retain log entries following it, and 2. successfully installing a snapshot from leader is equivalent to committing the snapshot's last included entry so both the follower and leader need to update their raft state to reflect that fact [8].

[6]This means two types of operations will now be pushed to *applyChan*: committed client requests and persisted snapshots.

[7]This pertains to how Raft leader append entries to followers. Refer to section 5.3 in [2].

3.3 Sharded, Replicated Key/Value Service

The final and last piece is to make our service sharded. We further break down the problem into two sub-problems: 1. build a replicated service, ShardMaster, that is responsible for managing sharding configuration, and 2. modify our key/value service to work with ShardMaster for a sharded key/value service.

Sub-problem one is straightforward to build: it employs the same architecture as our key/value store service built from the last two sections (i.e. a Raft RSM based service). The state for ShardMaster becomes a history of configurations where each client request (*Join*, *Leave*, *Move*, or *Query*) creates a new configuration and append it to that history. This service will try to distribute shards as evenly as possible to all replication groups while minimizing the number of shards moved across replication groups [9]. This is done by repeatedly assigning one shard from the most loaded group to the least loaded group until the most loaded group is one or zero shard higher than the least loaded group. This service does not actually transfer shard(s) between groups; it just record and store the configuration history information in an array of $Config$'s [10]. Sub-problem two is a bit tricky. Again, we start from the same replicated key/value service we built from the previous two sections. The client-side logic is minimally modified except that the clerk will now query ShardMaster for the latest con-

[8]This means the follower updates *commitIndex* and *lastApplied*; The leader updates *nextIndex* and *matchIndex* for the follower.

[9]Each replication group is a cluster of key/value servers responsible for zero or more shards. Union of all *kvStore* from all replication groups form all the key/value pairs in the data store.

[10]A $Config$ maintains the configuration number, group IDs, server names in each group, and shard assignment

figuration and send requests to leader of group responsible for the shard of the key. In the server-side logic, we will now have another long-running thread $PollConfigAndTransferShard()$ where the leader of each replication group will periodically query ShardMaster about the next configuration cfg_2 from its current configuration cfg_1 (i.e. $cfg_2 = Query(cfg_1.Num + 1)$). When a new configuration is detected, the leader checks whether this group has been involved in a shard assignment change. In particular, our implementation enforces that if this group g_1 is assigned a shard s that it was not responsible for previously, then the leader of this group sends a "PullShard" RPC request to the leader of the previously responsible group g_2 to ask for state of shard s [11] because it now needs to serve requests in this shard. Upon receiving state of this shard, g_1 invoke $Start(op)$ on its underlying Raft peer with op prepared for this configuration update. In the usual $Run()$ thread, we now need to watch out for the third type of operations popped from $applyChan$ – a $ConfigUpdate$ operation that is used to make sure all servers apply this configuration update (i.e. from cfg_1 to cfg_2) to their state machine at the same time. Of course, we also need to add a handler for the $PullShard$ RPC that replies with the state of the shard that was requested.

In the handler of $PullShard$ RPC, we can only proceed to prepare the state of the shard requested when our own configuration number is larger that of request. Consider the configuration change from cfg_1 to cfg_2 where shard s's assignment is moved from group g_1 to g_2. In our implementation, g_2 will send $PullShard$ to g_1 when it detects cfg_2. It is possible that when g_1's $PullShard$ handler is invoked, g_1 does not know ShardMaster updated configuration to cfg_2, nor does the clerks, so clerks are still sending requests of shard s to g_1 and g_1 is still processing and responding to clerks. Hence, g_1 should reject this $PullShard$ at this moment until it detects, replicates, and applies the configuration change to cfg_2, at which time g_1 can now safely respond to g_2 about the state of shard s. During this shard transfer time, clerks are sending their request in shard s to g_2 if it detects cfg_2 (or to g_1 if it doesn't). Either way, the clerks will be rejected because both g_1 and g_2 claim they are not responsible for shard s. This is okay because eventually the shard transfer will complete and group g_2 will respond to such requests. A another more subtle concern is that what if the new configuration requires two groups g_1 and g_2 to $PullShard$ from each other for some shard s_1 and s_2. Wouldn't our implementation cause a deadlock because s_1 and s_2 are waiting for each other to finish the configuration update so the other group can have a higher configuration number and hence start responding to their $PullShard$ RPC? We haven't experience that in tests yet but our shard assignment balancing algorithm made sure to transfer the minimum number of shards so we doubt such a case would ever happen because it seems that such an assignment does not transfer the minimum number of shards.

[11] Just like before, the state of a shard is 1. $kvStore$ portion of that shard, and 2. $maxAppliedOpIds$.

4 Test Results

See Figure 1, Figure 2, Figure 3, and Figure 4 at the end of the paper for test results.

5 Lessons Learned

There are a handful of things we learned during this process. Each one of them caused us great headache and many hours of debug time. I'd like to highlight them here because I think they are also very useful in building other distributed systems:

1. **Good architecture saves all**: I found myself constantly facing design decisions during this process. It is almost always the case that "yeah, we can do things this way; Or, we can do things in another seemingly equivalent way". But soon we realize that some of the decisions we made earlier were not good in the sense that we need to consider whether existing pieces would work with the introduction of this new element. If not, what are the possible cases, and very quickly we found our brain out of capacity to think and reason about a larger number of branches and

things. A good architecture allows us to maintain the things our brain need to reason about at a manageable level without exploding. For example, when I first approached the Sharded part, I made the $PollConfigAndTransferShard()$ thread to directly query the latest configuration because I thought "ohh, why updating to intermediate configurations when the clerks are also using the latest configuration to send requests? Every group should be striving to update directly to the latest configuration." Soon, I found out this is problematic because of multiple reasons; one is that the deadlock we mentioned above; the other one is that the group has no idea of the evolution of the group $Join/Leave$'s so it does not really know where to request the up-to-date shards. Then, I changed to that each group will query the next configuration and process all the configuration changes one by one, in order. This gives a way for our groups to incrementally transfer shards between each other.

2. **Establish good invariants and hold onto them** : When building the distributed system, I often had this feeling that there are so many things going on at the same time and there are so many cases I need to reason about – e.g. "If this happened first, then that happened, and then this happened, will this solution still work?" It is like trying to walk your way out of a foggy forest. Establishing general, good invariants is like a compass to guide our thinking and reasoning in that foggy forest. We think this is really the essence of why coding distributed systems is hard and how to tackle that difficulty, at least partially: there is a fixed ordering of statement execution in serial programs whereas there might be multiple possible ordering of statement execution and we need to make sure our solution works in all those ordering no matter how unlikely some of them are[12]. As a reaction, we try to establish invariants that always holds no matter which one of the execution ordering occurs. For example, the single most important invariant we replied upon, over and over again, is that operations popped from *applyChan* will be the same ones in the same order within each Raft RSM. This gives us a convenient way to ensure the same sequence of operations will be seen and processed by all servers in the cluster. In fact, I don't think there is any state update operation that is not applied through *applyChan* in *Run*().

3. **A comprehensive testing suite is essential** : All bugs in my implementation are found out by tests. In this work, the testing part is provided to us and we just need to use it. In practice, we should not only spend serious amount time into writing tests and in fact should prioritize writing tests over writing the solution. A recurrence to me is that I thought my code is now correct after multiple rounds of modification but then one of tests failed; After inspecting it I realize there is actually a path for the system state to reach a state that was previously thought impossible. What we can do when writing the code is to make the least amount of assumptions. For example, while I was implementing the handler for *InstallSnapshot* where I need to trim log entries to match the last included entry of the received snapshot. I had this implicit assumption in my mind that "yeah, the snapshot just received should be "ahead" of my application state. After all, this is why I received the snapshot in the first place – I was lagged behind and asked for some entries the leader has trimmed." so I don't have logic for the case that the received snapshot is actually "behind" my local snapshot. This was discovered in tests and then I passed it by adding the logic to discard such stale snapshots.

4. **Premature optimization is the root of all evils** : Sometimes I was tempted to do some optimizations when I was first writing the logic for some component of the distributed system, thinking such an optimization would be "benign". Then, in a much later stage where I struggle to debug one of the failed tests, I realize that in certain unusual cases, those "benign" optimizations becomes "harmful", causing correct-

[12]As the Murphy's law concisely outlined: anything can happen will happen.

ness issues in the system. What I've now learned is to write the plainest solution with minimum assumption first and optimize later. Also, premature optimization is additional mental burden. Having optimization added another layer of complexity that our mind has to constantly think and reason. Our mind is already heavily loaded so do ourselves a favor to worry about optimizations later.

5. **Never wait while holding a lock** : There are multiple things we know that require waiting – waiting for something from a channel, making a remote procedure call, etc. It might seem easier to hold the lock while doing all these but the performance incurrence is prohibitive. We should also release locks before any of such events and there is always a way to make it work. When we said "worry about optimization later", this is not one of them.

6 Future Work

We list a few things that can/should be improved:

1. **A race-free implementation of the Raft RSM** : When implementing the Raft RSM, we wrote a small fraction of the code in a race-ful way. That is, code that does not use locks to guard access to shard resources. We carefully analyzed all possible cases and came to the conclusion that this is fine. This was further justified by running the tests hundreds of times without any failure. We believe that an industrial-strength product should not do such a hacky thing.

2. **Handle read-only operations without writing them to the log** : For read-only operations (e.g. $Get(k)$, $Query(num)$), we don't want to write it to log and only apply it when it's committed to a majority of servers. We want to respond with up-to-date data without writing to the log.

3. **Delete keys of a shard once the shard is assigned to others** : When a shard is transferred to a different group in a new configuration, that old group responsible for this shard should delete all key/value pairs of that shard from its *kvStore*. It is wasteful for it to keep key/value pairs it no longer owns and serves.

4. **Do not block client requests of uninvolved shards during reconfiguration** : Say a replication group is responsible for two shards. If this group is assigned a additional shard during a configuration change, then this group's servers are blocked to wait until the completion of the reconfiguration, although this group is perfectly capable of serving requests from the previous two shards during reconfiguration.

7 Summary

We built a sharded, replicated key/value store service with strong consistency guarantee. It exposes three interface requests to clients: $Get(k)$, $Put(k, v)$, and $Append(k, v)$. It also provides four interface requests for system administrators about sharding configuration: $Join(servers)$, $Leave(GIDs)$, $Move(shard, GID)$, and $Query(num)$.

Our service is crash-fault tolerant and sharded. It endures failure of a minority of servers. Servers are automatically joined back after restart without manual intervention. The service is capable of dynamically scaling up (resp. down) throughput by adding more servers and changing configurations.

Our service provides strong consistency (aka linearizability) guarantee. For non-concurrent requests, their real time order will be honored in the results; For concurrent requests, the results can be interpreted as if they are executed on a single machine one by one in some order.

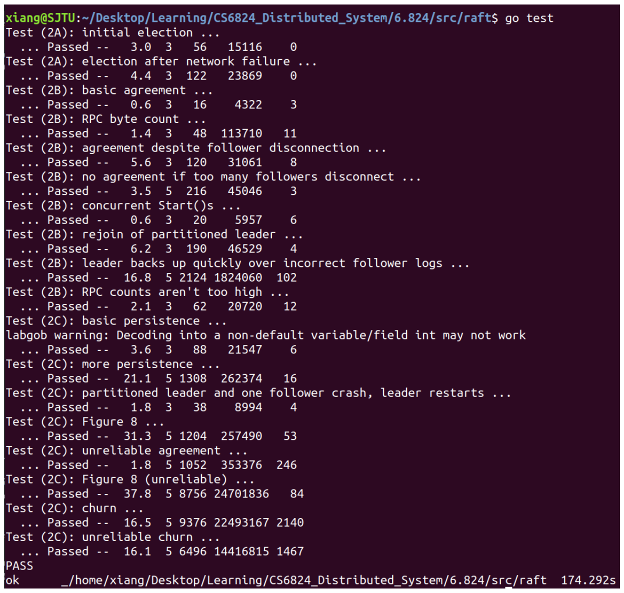
We also detailed our approach towards building such a system. We first implemented a Raft replicated state machine. Then we add client/server logic to make it a key/value store service with necessary optimization to the underlying raft replicated state machine. Finally, we add components to make it a sharded service by incorporating another replicated service that maintains the sharding configuration in it.

We also discussed our experiences and lessons learned from this process. We believe those useful lessons transfer to building other distributed systems.

References

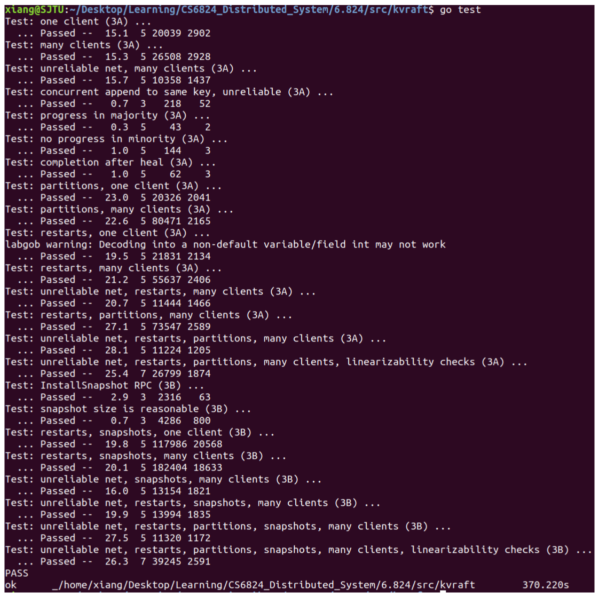
[1] Robert Morris. “MIT6.824: Distributed Systems”. In: MIT OpenCourseWare. Cambridge MA, 2020. URL: `http://nil.lcs.mit.edu/6.824/2020/schedule.html`.

[2] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.



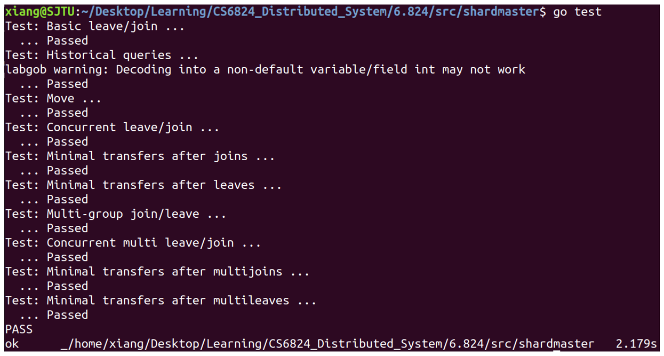
```
xiang@SJTU:~/Desktop/Learning/CS6824_Distributed_System/6.824/src/raft$ go test
Test (2A): initial election ...
  ... Passed --   3.0  3   56   15116    0
Test (2A): election after network failure ...
  ... Passed --   4.4  3  122   23869    0
Test (2B): basic agreement ...
  ... Passed --   0.6  3   16    4322    3
Test (2B): RPC byte count ...
  ... Passed --   1.4  3   48  113710   11
Test (2B): agreement despite follower disconnection ...
  ... Passed --   5.6  3  120   31061    8
Test (2B): no agreement if too many followers disconnect ...
  ... Passed --   3.5  5  216   45046    3
Test (2B): concurrent Start()s ...
  ... Passed --   0.6  3   20    5957    6
Test (2B): rejoin of partitioned leader ...
  ... Passed --   6.2  3  190   46529    4
Test (2B): leader backs up quickly over incorrect follower logs ...
  ... Passed --  16.8  5 2124 1824060  102
Test (2B): RPC counts aren't too high ...
  ... Passed --   2.1  3   62   20720   12
Test (2C): basic persistence ...
labgob warning: Decoding into a non-default variable/field int may not work
  ... Passed --   3.6  3   88   21547    6
Test (2C): more persistence ...
  ... Passed --  21.1  5 1308  262374   16
Test (2C): partitioned leader and one follower crash, leader restarts ...
  ... Passed --   1.8  3   38    8994    4
Test (2C): Figure 8 ...
  ... Passed --  31.3  5 1204  257490   53
Test (2C): unreliable agreement ...
  ... Passed --   1.8  5 1052  353376  246
Test (2C): Figure 8 (unreliable) ...
  ... Passed --  37.8  5 8756 24701836   84
Test (2C): churn ...
  ... Passed --  16.5  5 9376 22493167 2140
Test (2C): unreliable churn ...
  ... Passed --  16.1  5 6496 14416815 1467
PASS
ok      _/home/xiang/Desktop/Learning/CS6824_Distributed_System/6.824/src/raft  174.292s
```

Figure 1: Test result for 3.1



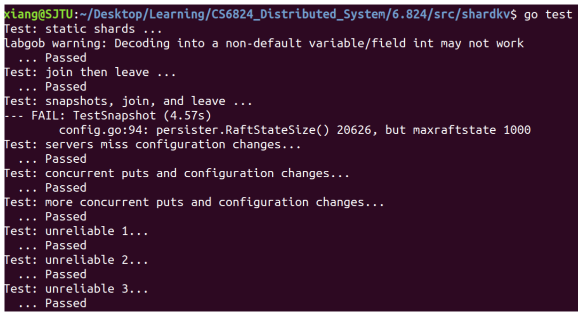
```
xiang@SJTU:~/Desktop/Learning/CS6824_Distributed_System/6.824/src/kvraft$ go test
Test: one client (3A) ...
  ... Passed --  15.1  5 20039 2902
Test: many clients (3A) ...
  ... Passed --  15.3  5 26508 2928
Test: unreliable net, many clients (3A) ...
  ... Passed --  15.7  5 10358 1437
Test: concurrent append to same key, unreliable (3A) ...
  ... Passed --   0.7  3   218   52
Test: progress in majority (3A) ...
  ... Passed --   0.3  5    43    2
Test: no progress in minority (3A) ...
  ... Passed --   1.0  5   144    3
Test: completion after heal (3A) ...
  ... Passed --   1.0  5    62    3
Test: partitions, one client (3A) ...
  ... Passed --  23.0  5 20326 2041
Test: partitions, many clients (3A) ...
  ... Passed --  22.6  5 80471 2165
Test: restarts, one client (3A) ...
labgob warning: Decoding into a non-default variable/field int may not work
  ... Passed --  19.5  5 21831 2134
Test: restarts, many clients (3A) ...
  ... Passed --  21.2  5 55637 2406
Test: unreliable net, restarts, many clients (3A) ...
  ... Passed --  20.7  5 11444 1466
Test: restarts, partitions, many clients (3A) ...
  ... Passed --  27.1  5 73547 2589
Test: unreliable net, restarts, partitions, many clients (3A) ...
  ... Passed --  28.1  5 11224 1205
Test: unreliable net, restarts, partitions, many clients, linearizability checks (3A) ...
  ... Passed --  25.4  7 26799 1874
Test: InstallSnapshot RPC (3B) ...
  ... Passed --   2.9  3  2316   63
Test: snapshot size is reasonable (3B) ...
  ... Passed --   0.7  3  4286  800
Test: restarts, snapshots, one client (3B) ...
  ... Passed --  19.8  5 117986 20568
Test: restarts, snapshots, many clients (3B) ...
  ... Passed --  20.1  5 182404 18633
Test: unreliable net, snapshots, many clients (3B) ...
  ... Passed --  16.0  5 13154 1821
Test: unreliable net, restarts, snapshots, many clients (3B) ...
  ... Passed --  19.9  5 13994 1835
Test: unreliable net, restarts, partitions, snapshots, many clients (3B) ...
  ... Passed --  27.5  5 11320 1172
Test: unreliable net, restarts, partitions, snapshots, many clients, linearizability checks (3B) ...
  ... Passed --  26.3  7 39245 2591
PASS
ok      _/home/xiang/Desktop/Learning/CS6824_Distributed_System/6.824/src/kvraft        370.220s
```

Figure 2: Test result for 3.2



```
xiang@SJTU:~/Desktop/Learning/CS6824_Distributed_System/6.824/src/shardmaster$ go test
Test: Basic leave/join ...
  ... Passed
Test: Historical queries ...
labgob warning: Decoding into a non-default variable/field int may not work
  ... Passed
Test: Move ...
  ... Passed
Test: Concurrent leave/join ...
  ... Passed
Test: Minimal transfers after joins ...
  ... Passed
Test: Minimal transfers after leaves ...
  ... Passed
Test: Multi-group join/leave ...
  ... Passed
Test: Concurrent multi leave/join ...
  ... Passed
Test: Minimal transfers after multijoins ...
  ... Passed
Test: Minimal transfers after multileaves ...
  ... Passed
PASS
ok      _/home/xiang/Desktop/Learning/CS6824_Distributed_System/6.824/src/shardmaster   2.179s
```

Figure 3: Test result for the ShardMaster service mentioned in 3.3



```
xiang@SJTU:~/Desktop/Learning/CS6824_Distributed_System/6.824/src/shardkv$ go test
Test: static shards ...
labgob warning: Decoding into a non-default variable/field int may not work
  ... Passed
Test: join then leave ...
  ... Passed
Test: snapshots, join, and leave ...
--- FAIL: TestSnapshot (4.57s)
        config.go:94: persister.RaftStateSize() 20626, but maxraftstate 1000
Test: servers miss configuration changes...
  ... Passed
Test: concurrent puts and configuration changes...
  ... Passed
Test: more concurrent puts and configuration changes...
  ... Passed
Test: unreliable 1...
  ... Passed
Test: unreliable 2...
  ... Passed
Test: unreliable 3...
  ... Passed
```

Figure 4: Test result for 3.3