6.824 - Spring 2018

6.824 Lab 4: Sharded Key/Value Service

Part A Due: Apr 20 at 11:59pm

Part B Due: May 11 at 11:59pm

Introduction

In this lab you'll build a key/value storage system that "shards," or partitions, the keys over a set of replica groups. A shard is a subset of the key/value pairs; for example, all the keys starting with "a" might be one shard, all the keys starting with "b" another, etc. The reason for sharding is performance. Each replica group handles puts and gets for just a few of the shards, and the groups operate in parallel; thus total system throughput (puts and gets per unit time) increases in proportion to the number of groups.

Your sharded key/value store will have two main components. First, a set of replica groups. Each replica group is responsible for a subset of the shards. A replica consists of a handful of servers that use Raft to replicate the group's shards. The second component is the "shard master". The shard master decides which replica group should serve each shard; this information is called the configuration. The configuration changes over time. Clients consult the shard master in order to find the replica group for a key, and replica groups consult the master in order to find out what shards to serve. There is a single shard master for the whole system, implemented as a fault-tolerant service using Raft.

A sharded storage system must be able to shift shards among replica groups. One reason is that some groups may become more loaded than others, so that shards need to be moved to balance the load. Another reason is that replica groups may join and leave the system: new replica groups may be added to increase capacity, or existing replica groups may be taken offline for repair or retirement.

The main challenge in this lab will be handling reconfiguration -- changes in the assignment of shards to groups. Within a single replica group, all group members must agree on when a reconfiguration occurs relative to client Put/Append/Get requests. For example, a Put may arrive at about the same time as a reconfiguration that causes the replica group to stop being responsible for the shard holding the Put's key. All replicas in the group must agree on whether the Put occurred before or after the reconfiguration. If before, the Put should take effect and the new owner of the shard will see its effect; if after, the Put won't take effect and client must re-try at the new owner. The recommended approach is to have each replica group use Raft to log not just the sequence of Puts, Appends, and Gets but also the sequence of reconfigurations. You will need to ensure that at most one replica group is serving requests for each shard at any one time.

Reconfiguration also requires interaction among the replica groups. For example, in configuration 10 group G1 may be responsible for shard S1. In configuration 11, group G2 may be responsible for shard S1. During the reconfiguration from 10 to 11, G1 and G2 must use RPC to move the contents of shard S1 (the key/value pairs) from G1 to G2.

> **Note:** Only RPC may be used for interaction among clients and servers. For example, different instances of your server are not allowed to share Go variables or files.

> **Note:** This lab uses "configuration" to refer to the assignment of shards to replica groups. This is not the same as Raft cluster membership changes. You don't have to implement Raft cluster membership changes.

This lab's general architecture (a configuration service and a set of replica groups) follows the same general pattern as Flat Datacenter Storage, BigTable, Spanner, FAWN, Apache HBase, Rosebud, Spinnaker, and many others. These systems differ in many details from this lab, though, and are also typically more sophisticated and capable. For example, the lab doesn't evolve the sets of peers in each Raft group; its data and query models are very simple; and handoff of shards is slow and doesn't allow concurrent client access.

> **Note:** Your Lab 4 sharded server, Lab 4 shard master, and Lab 3 kvraft must all use the same Raft implementation. We will re-run the Lab 2 and Lab 3 tests as part of grading Lab 4, and your score on the older tests will count towards your total Lab 4 grade. **These tests are worth 10 points out of your overall Lab 4 grade.**

Collaboration Policy

You must write all the code you hand in for 6.824, except for code that we give you as part of the assignment. You are not allowed to look at anyone else's solution, you are not allowed to look at code from previous years, and you are not allowed to look at other Raft implementations. You may discuss the assignments with other students, but you may not look at or copy each others' code. Please do not publish your code or make it available to future 6.824 students -- for example, please do not make your code visible on GitHub (instead, create a private repository on MIT's GitHub deployment).

Getting Started

> **Important:**
>
> Do a `git pull` to get the latest lab software.

We supply you with skeleton code and tests in `src/shardmaster` and `src/shardkv`.

To get up and running, execute the following commands:

```
$ cd ~/6.824
$ git pull
...
$ cd src/shardmaster
$ GOPATH=~/6.824
$ export GOPATH
$ go test
--- FAIL: TestBasic (0.00s)
        test_test.go:11: wanted 1 groups, got 0
FAIL
exit status 1
```

```
FAIL    shardmaster     0.008s
$
```

When you're done, your implementation should pass all the tests in the `src/shardmaster` directory, and all the ones in `src/shardkv`.

Part A: The Shard Master (30 points)

First you'll implement the shard master, in `shardmaster/server.go` and `client.go`. When you're done, you should pass all the tests in the `shardmaster` directory:

```
$ cd ~/6.824/src/shardmaster
$ go test
Test: Basic leave/join ...
  ... Passed
Test: Historical queries ...
  ... 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      shardmaster     13.127s
$
```

The shardmaster manages a sequence of numbered configurations. Each configuration describes a set of replica groups and an assignment of shards to replica groups. Whenever this assignment needs to change, the shard master creates a new configuration with the new assignment. Key/value clients and servers contact the shardmaster when they want to know the current (or a past) configuration.

Your implementation must support the RPC interface described in `shardmaster/common.go`, which consists of `Join`, `Leave`, `Move`, and `Query` RPCs. These RPCs are intended to allow an administrator (and the tests) to control the shardmaster: to add new replica groups, to eliminate replica groups, and to move shards between replica groups.

The `Join` RPC is used by an administrator to add new replica groups. Its argument is a set of mappings from unique, non-zero replica group identifiers (GIDs) to lists of server names. The shardmaster should react by creating a new configuration that includes the new replica groups. The new configuration should divide the shards as evenly as possible among the full set of groups, and should move as few shards as possible to achieve that goal. The shardmaster should allow re-use of a GID if it's not part of the current configuration (i.e. a GID should be allowed to Join, then Leave, then Join again).

The `Leave` RPC's argument is a list of GIDs of previously joined groups. The shardmaster should create a new configuration that does not include those groups, and that assigns those groups' shards to the remaining groups. The new configuration should divide the shards as evenly as possible among the groups, and should move as few shards as possible to achieve that goal.

The `Move` RPC's arguments are a shard number and a GID. The shardmaster should create a new configuration in which the shard is assigned to the group. The purpose of `Move` is to allow us to test your software. A `Join` or `Leave` following a `Move` will likely un-do the `Move`, since `Join` and `Leave` re-balance.

The `Query` RPC's argument is a configuration number. The shardmaster replies with the configuration that has that number. If the number is -1 or bigger than the biggest known configuration number, the shardmaster should reply with the latest configuration. The result of `Query(-1)` should reflect every `Join`, `Leave`, or `Move` RPC that the shardmaster finished handling before it received the `Query(-1)` RPC.

The very first configuration should be numbered zero. It should contain no groups, and all shards should be assigned to GID zero (an invalid GID). The next configuration (created in response to a `Join` RPC) should be numbered 1, &c. There will usually be significantly more shards than groups (i.e., each group will serve more than one shard), in order that load can be shifted at a fairly fine granularity.

TASK

Your task is to implement the interface specified above in `client.go` and `server.go` in the `shardmaster/` directory. Your shardmaster must be fault-tolerant, using your Raft library from Lab 2/3. Note that we will re-run the tests from Lab 2 and 3 when grading Lab 4, so make sure you do not introduce bugs into your Raft implementation. You have completed this task when you pass all the tests in `shardmaster/`.

- **Hint:** Start with a stripped-down copy of your kvraft server.
- **Hint:** You should implement duplicate client request detection for RPCs to the shard master. The shardmaster tests don't test this, but the shardkv tests will later use your shardmaster on an unreliable network; you may have trouble passing the shardkv tests if your shardmaster doesn't filter out duplicate RPCs.
- **Hint:** Go maps are references. If you assign one variable of type map to another, both variables refer to the same map. Thus if you want to create a new `Config` based on a previous one, you need to create a new map object (with `make()`) and copy the keys and values individually.
- **Hint:** The Go race detector (go test -race) may help you find bugs.

Part B: Sharded Key/Value Server (60 points)

Important:

Do a `git pull` to get the latest lab software.

Now you'll build shardkv, a sharded fault-tolerant key/value storage system. You'll modify `shardkv/client.go`, `shardkv/common.go`, and `shardkv/server.go`.

Each shardkv server operates as part of a replica group. Each replica group serves `Get`, `Put`, and `Append` operations for some of the key-space shards. Use `key2shard()` in `client.go` to find which shard a key belongs to. Multiple replica groups cooperate to serve the complete set of shards. A single instance of the `shardmaster` service assigns shards to replica groups; when this assignment changes, replica groups have to hand off shards to each other, while ensuring that clients do not see inconsistent responses.

Your storage system must provide a linearizable interface to applications that use its client interface. That is, completed application calls to the `Clerk.Get()`, `Clerk.Put()`, and `Clerk.Append()` methods in `shardkv/client.go` must appear to have affected all replicas in the same order. A `Clerk.Get()` should see the value written by the most recent `Put`/`Append` to the same key. This must be true even when `Get`s and `Put`s arrive at about the same time as configuration changes.

Each of your shards is only required to make progress when a majority of servers in the shard's Raft replica group is alive and can talk to each other, and can talk to a majority of the `shardmaster` servers. Your implementation must operate (serve requests and be able to re-configure as needed) even if a minority of servers in some replica group(s) are dead, temporarily unavailable, or slow.

A shardkv server is a member of only a single replica group. The set of servers in a given replica group will never change.

We supply you with `client.go` code that sends each RPC to the replica group responsible for the RPC's key. It re-tries if the replica group says it is not responsible for the key; in that case, the client code asks the shard master for the latest configuration and tries again. You'll have to modify client.go as part of your support for dealing with duplicate client RPCs, much as in the kvraft lab.

When you're done your code should pass all the shardkv tests other than the challenge tests:

```
$ cd ~/6.824/src/shardkv
$ go test
Test: static shards ...
  ... Passed
Test: join then leave ...
  ... Passed
Test: snapshots, join, and leave ...
  ... Passed
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: shard deletion (challenge 1) ...
  ... Passed
Test: concurrent configuration change and restart (challenge 1)...
  ... Passed
Test: unaffected shard access (challenge 2) ...
  ... Passed
Test: partial migration shard access (challenge 2) ...
  ... Passed
PASS
ok      shardkv 206.132s
```

```
$
```

Note: Your server should not call the shard master's `Join()` handler. The tester will call `Join()` when appropriate.

TASK

Your first task is to pass the very first shardkv test. In this test, there is only a single assignment of shards, so your code should be very similar to that of your Lab 3 server. The biggest modification will be to have your server detect when a configuration happens and start accepting requests whose keys match shards that it now owns.

Now that you solution works for the static sharding case, it's time to tackle the problem of configuration changes. You will need to make your servers watch for configuration changes, and when one is detected, to start the shard migration process. If a replica group loses a shard, it must stop serving requests to keys in that shard immediately, and start migrating the data for that shard to the replica group that is taking over ownership.If a replica group gains a shard, it needs to wait for the previous owner to send over the old shard data before accepting requests for that shard.

TASK

Implement shard migration during configuration changes. Make sure that all servers in a replica group do the migration at the same point in the sequence of operations they execute, so that they all either accept or reject concurrent client requests. You should focus on passing the second test ("join then leave") before working on the later tests. You are done with this task when you pass all tests up to, but not including, `TestDelete`.

Note: Your server will need to periodically poll the shardmaster to learn about new configurations. The tests expect that your code polls roughly every 100 milliseconds; more often is OK, but much less often may cause problems.

Note: Servers will need to send RPCs to each other in order to transfer shards during configuration changes. The shardmaster's `Config` struct contains server names, but you need a `labrpc.ClientEnd` in order to send an RPC. You should use the `make_end()` function passed to `StartServer()` to turn a server name into a `ClientEnd`. `shardkv/client.go` contains code that does this.

- **Hint:** Add code to `server.go` to periodically fetch the latest configuration from the shardmaster, and add code to reject client requests if the receiving group isn't responsible for the client's key's shard. You should still pass the first test.
- **Hint:** Your server should respond with an `ErrWrongGroup` error to a client RPC with a key that the server isn't responsible for (i.e. for a key whose shard is not assigned to the server's group). Make sure your `Get`, `Put`, and `Append` handlers make this decision correctly in the face of a concurrent re-configuration.
- **Hint:** Process re-configurations one at a time, in order.

- **Hint:** If a test fails, check for gob errors (e.g. "gob: type not registered for interface ..."). Go doesn't consider gob errors to be fatal, although they are fatal for the lab.
- **Hint:** You'll need to provide at-most-once semantics (duplicate detection) for client requests across shard movement.
- **Hint:** Think about how the shardkv client and server should deal with `ErrWrongGroup`. Should the client change the sequence number if it receives `ErrWrongGroup`? Should the server update the client state if it returns `ErrWrongGroup` when executing a `Get/Put` request?
- **Hint:** After a server has moved to a new configuration, it is acceptable for it to continue to store shards that it no longer owns (though this would be regrettable in a real system). This may help simplify your server implementation.
- **Hint:** When group G1 needs a shard from G2 during a configuration change, does it matter at what point during its processing of log entries G2 sends the shard to G1?
- **Hint:** You can send an entire map in an RPC request or reply, which may help keep the code for shard transfer simple.
- **Hint:** If one of your RPC handlers includes in its reply a map (e.g. a key/value map) that's part of your server's state, you may get bugs due to races. The RPC system has to read the map in order to send it to the caller, but it isn't holding a lock that covers the map. Your server, however, may proceed to modify the same map while the RPC system is reading it. The solution is for the RPC handler to include a copy of the map in the reply.
- **Hint:** If you put a map or a slice in a Raft log entry, and your key/value server subsequently sees the entry on the `applyCh` and saves a reference to the map/slice in your key/value server's state, you may have a race. Make a copy of the map/slice, and store the copy in your key/value server's state. The race is between your key/value server modifying the map/slice and Raft reading it while persisting its log.
- **Hint:** During a configuration change, a pair of groups may need to move shards in both directions between them. If you see deadlock, this is a possible source.

Challenge exercises

For this lab, we have two challenge exercises, both of which are fairly complex beasts, but which are also essential if you were to build a system like this for production use.

Garbage collection of state (15 bonus points)

When a replica group loses ownership of a shard, that replica group should eliminate the keys that it lost from its database. It is wasteful for it to keep values that it no longer owns, and no longer serves requests for. However, this poses some issues for migration. Say we have two groups, G1 and G2, and there is a new configuration C that moves shard S from G1 to G2. If G1 erases all keys in S from its database when it transitions to C, how does G2 get the data for S when it tries to move to C?

CHALLENGE

Modify your solution so that each replica group will only keep old shards for as long as is absolutely necessary. Bear in mind that your solution must continue to work even if all the servers in a replica group like G1 above crash and are then brought back up. You have completed this challenge if you pass `TestChallenge1Delete` and `TestChallenge1Concurrent`.

Client requests during configuration changes (15 bonus points)

The simplest way to handle configuration changes is to disallow all client operations until the transition has completed. While conceptually simple, this approach is not feasible in production-level systems; it results in long pauses for all clients whenever machines are brought in or taken out. A better solution would be if the system continued serving shards that are not affected by the ongoing configuration change.

> CHALLENGE
>
> Modify your solution so that, if some shard S is not affected by a configuration change from C to C', client operations to S should continue to succeed while a replica group is still in the process of transitioning to C'. You have completed this challenge when you pass `TestChallenge2Unaffected`.

While the optimization above is good, we can still do better. Say that some replica group G3, when transitioning to C, needs shard S1 from G1, and shard S2 from G2. We really want G3 to immediately start serving a shard once it has received the necessary state, even if it is still waiting for some other shards. For example, if G1 is down, G3 should still start serving requests for S2 once it receives the appropriate data from G2, despite the transition to C not yet having completed.

> CHALLENGE
>
> Modify your solution so that replica groups start serving shards the moment they are able to, even if a configuration is still ongoing. You have completed this challenge when you pass `TestChallenge2Partial`.

Handin procedure

> **Important:**
>
> Before submitting, please run *all* the tests one final time.
>
> Also, note that your Lab 4 sharded server, Lab 4 shard master, and Lab 3 kvraft must all use the same Raft implementation. We will re-run the Lab 2 and Lab 3 tests as part of grading Lab 4.
>
> Before submitting, double check that your solution works with:

```
$ go test raft/...
$ go test kvraft/...
$ go test shardmaster/...
$ go test shardkv/...
```

Submit your code via the class's submission website, located at https://6824.scripts.mit.edu/2018/handin.py/.

You may use your MIT Certificate or request an API key via email to log in for the first time. Your API key (`XXX`) is displayed once you logged in, which can be used to upload the lab from the console as follows.

For part A:

```
$ cd "$GOPATH"
$ echo "XXX" > api.key
$ make lab4a
```

For part B:

```
$ cd "$GOPATH"
$ echo "XXX" > api.key
$ make lab4b
```

Important:

Check the submission website to make sure it sees your submission!

Note: You may submit multiple times. We will use the timestamp of your **last** submission for the purpose of calculating late days. Your grade is determined by the score your solution **reliably** achieves when we run the tester on our test machines.

Please post questions on Piazza.