6.824 - Spring 2015

6.824 Lab 4: Sharded Key/Value Service

Part A Due: Fri Apr 3 11:59pm

Part B Due: Fri Apr 17 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 Paxos to replicate the group's shard. 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 Paxos.

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 in the replica 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 Paxos to log not just the sequence of Puts, Appends, and Gets but also the sequence of reconfigurations.

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 must send the contents of shard S1 (the key/value pairs) to G2.

You will need to ensure that at most one replica group is serving requests for each shard. Luckily it is reasonable to assume that each replica group is always available, because each group uses Paxos for replication and thus can tolerate some network and server failures. As a result, your design can rely on one group to actively hand off responsibility to another group during reconfiguration. This is simpler than the situation in primary/backup replication (Lab 2), where the old primary is often not reachable and may still think it is primary.

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

This lab's general architecture (a configuration service and a set of replica groups) is patterned at a high level on a number of systems: Flat Datacenter Storage, BigTable, Spanner, FAWN, Apache HBase, Rosebud, and many others. These systems differ in many details from this lab, though, and are also typically more sophisticated and capable. For example, your lab lacks persistent storage for key/value pairs and for the Paxos log; it sends more messages than required per Paxos agreement; it cannot evolve the sets of peers in each Paxos group; its data and query models are very simple; and handoff of shards is slow and doesn't allow concurrent client access.

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, and you are not allowed to look at code from previous years. 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.

Software

Do a `git pull` to get the latest lab software. We supply you with new skeleton code and new tests in `src/shardmaster` and `src/shardkv`.

```
$ add 6.824
$ cd ~/6.824
$ git pull
...
$ cd src/shardmaster
$ go test
Basic leave/join: --- FAIL: TestBasic (0.00 seconds)
test_test.go:37:    wanted 1 groups, got 0
--- FAIL: TestUnreliable (0.32 seconds)
```

```
test_test.go:37:    wanted 20 groups, got 0
FAIL
$
```

Part A: The Shard Master

First you'll implement the shard master, in `shardmaster/server.go`. When you're done, you should pass all the tests in the `shardmaster` directory (after ignoring Go's many complaints):

```
$ 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: Concurrent leave/join, failure ...
  ... Passed
PASS
ok      shardmaster     11.200s
$
```

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. You should not change common.go or client.go.

You don't need to implement duplicate client request detection for RPCs to the shard master that might fail or repeat due to network issues. A real system would need to do so, but these labs don't require it. You'll still need to deal with clients sending multiple Joins or Leaves to the shardmaster.

You do need to detect duplicate client RPCs to the shardkv service in Part B. Please make sure that your scheme for duplicate detection frees server memory quickly, for example by having the client tell the servers which RPCs it has heard a reply for. It's OK to piggyback this information on the next client request.

The Join RPC's arguments are a unique non-zero replica group identifier (GID) and an array of server ports. The shardmaster should react by creating a new configuration that includes the new replica group. 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 Leave RPC's arguments are the GID of a previously joined group. The shardmaster should create a new configuration that does not include the group, and that assigns the group's 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 main purpose of Move is to allow us to test your software, but it might also be useful to fine-tune load balance if some shards are more popular than others or some replica groups are slower than others. 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 that completed before the Query(-1) RPC was sent.

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.

Your shardmaster must be fault-tolerant, using your Paxos library from Lab 3.

Hint: start with a stripped-down copy of your kvpaxos server.

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.

Part B: Sharded Key/Value Server

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 will operate as part of a replica group. Each replica group will serve Get/Put/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 will cooperate to serve the complete set of shards. A single instance of the `shardmaster` service will assign shards to replica groups. When this assignment changes, replica groups will have to hand off shards to each other.

Your storage system must provide sequential consistency 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 will get tricky when Gets and Puts arrive at about the same time as configuration changes.

You are allowed to assume that a majority of servers in each Paxos replica group are alive and can talk to each other, can talk to a majority of the `shardmaster` servers, and can talk to a majority of each other replica group. Your implementation must operate (serve requests and be able to re-configure as needed) if a minority of servers in some replica group(s) are dead, temporarily unavailable, or slow.

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 kvpaxos lab.

When you're done your code should pass the shardkv tests:

```
$ cd ~/6.824/src/shardkv
$ go test
Test: Basic Join/Leave ...
  ... Passed
Test: Shards really move ...
  ... Passed
Test: Reconfiguration with some dead replicas ...
  ... Passed
Test: Concurrent Put/Get/Move ...
  ... Passed
Test: Concurrent Put/Get/Move (unreliable) ...
  ... Passed
PASS
ok      shardkv 62.350s
$
```

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

The two Concurrent test cases above test several clients sending Append and Get operations to different shard groups concurrently while periodically also asking the shard master to move shards between groups. To pass these test cases you must design a correct protocol for handling concurrent operations in the presence of configuration changes. The second concurrent test case is the same as the first one, but the test software drops requests and responses randomly.

Hint: your server will need to periodically check with the shardmaster to see if there's a new configuration; do this in tick().

Hint: you should have a function whose job it is to examine recent entries in the Paxos log and apply them to the state of the shardkv server. Don't directly update the stored key/value database in the Put/Append/Get handlers; instead, attempt to append a Put, Append, or Get operation to the Paxos log, and then call your log-reading function to find out what happened (e.g., perhaps a reconfiguration was entered in the log just before the Put/Append/Get).

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/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: during re-configuration, replica groups will have to send each other the keys and values for some shards.

Hint: you must call px.Done() to let Paxos free old parts of its log.

Hint: When the test fails, **check for gob error (e.g. "rpc: writing response: gob: type not registered for interface ...") in the log** because go doesn't consider the error fatal, although it is fatal for the lab.

Hint: Be careful about implementing at-most-once semantic for RPC. When a server sends shards to another, the server needs to send the clients state as well. Think about how the receiver of the shards should update its own clients state. Is it ok for the receiver to replace its clients state with the received one?

Hint: Think about how should the shardkv client and server 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 view, it can leave the shards that it is not owning in the new view undeleted. This will simplify the server implementation.

Hint: Think about when it is ok for a server to give shards to the other server during view change.

Handin procedure

Submit your code via the class's submission website, located here:

https://6824.scripts.mit.edu:444/submit/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 lab4 from the console as follows. For part A:

```
$ cd ~/6.824
$ echo XXX > api.key
$ make lab4a
```

And for part B:

```
$ cd ~/6.824
$ echo XXX > api.key
$ make lab4b
```

You can check the submission website to check if your submission is successful.

You will receive full credit if your software passes the `test_test.go` tests when we run your software on our machines. We will use the timestamp of your **last** submission for the purpose of calculating late days.

Please post questions on Piazza.