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.

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

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 c ⓘ ges.

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.

Your Lab 3 sharded server, Lab 3 shard master, and Lab 2 kvraft must all use the same Raft implementation. You may change your Raft implementation as needed.

Lab Setup

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

To get up and running, execute the following commands:

```
$ git checkout lab-kv-solution # ensure you are in the raft solution branch first.
$ git checkout -b lab-shard-solution # create a new branch based on your raft
solution.
$ wget http://seojinpark.net/downloads/dslab_lab3.patch # get the patch file
$ git am -3 --ignore-whitespace --ignore-space-change < dslab_lab3.patch # apply
the patch file. See the notes below if any error occurs
$ rm dslab_lab3.patch # clean up
$ git push --set-upstream origin lab-shard-solution # push updates to the remote
repository
$ python3 waf configure build --enable-raft-test # build the project (the same
command as before)
$ build/deptran_server -f config/shard_lab_test.yml # run the project
```

Note:

If you encountered any merge conflict when executing `git am -3 --ignore-whitespace --ignore-space-change < dslab_lab3.patch`, please follow these steps:

1. Locates all files that have conflicts (with merge marker).
    1. If the conflict is located in the skeleton code, such as `src/deptran/raft/frame.cc` or `.vscode/settings.json`, simply accept the incoming changes (changes come with the patch).
    2. If the conflict is located in your written code for previous labs, please resolve these conflicts manually.
    3. If you are really uncertain about the resolution, please post on Piazza. We will guide you through.
2. After resolving the conflicts and saving all changes, run `git am --continue` to finish up the patch. If it goes through successfully, you should see "**Applying: Template for Lab 3**". Otherwise, check the remaining conflicts and repeat this step until it finishes.

Part 0: Adapting to the New Skeleton Code

Previously, we assumed all PRCs were sent to partition 0 in the `commo.cc` file. In this lab, each partition will have a Raft group (servers) to maintain the replications (details are listed in the following section). Therefore, you might want to adjust your codebase correspondingly to send PRCs to the correct partition.

Part A: The Shard Master

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/service.h, 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 return code `ret` is similar to what you have in the kv lab: `KV_SUCCESS`, `KV_TIMEOUT`, `KV_NOTLEADER`.

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. 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

group will serve more than one shard), in order that load can be shifted at a fairly fine granularity.

Tip: You may want to serialize a simple data structure into a string, `std::stringstream` and `boost::serialization(archive)` may come in handy. The compiling script is ready for you to use them (library linking is added). If you want to use other means of serialization, make sure they are header-only, i.e., you cannot use heavy serialization libraries like ProtoBuf.

Part B: Sharded Key/Value Store

Now you'll build shardkv, a sharded fault-tolerant key/value storage system. You'll modify files in the shardkv directory.

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. The system uses Key2Shard() 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 Get(), Put(), Append() methods must appear to have affected all replicas in the same order. A Get() should see the value written by the most recent Put/Append to the same key. This must be true even when Gets and Puts 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.

You will need to modify code in the client 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.

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

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 the kv-lab. 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.

N ⓘ that your 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.

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.

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.

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

```
$ build/deptran_server -f config/shard_lab_test.yml

TEST 1: Basic shard operations

TEST 1 Passed

TEST 2: Concurrent shard operations

TEST 2 Passed

TEST 3: Minimal transfers after joins

TEST 3 Passed

TEST 4: Minimal transfers after leaves

TEST 4 Passed

TEST 5: Static shards, put

TEST 5 Passed

TEST 6: Shard joining and leaving with append

TEST 6 Passed
```