6.5840 - Spring 2024

6.5840 Lab 4: Fault-tolerant Key/Value Service

Collaboration policy // Submit lab // Setup Go // Guidance // Piazza

---

Introduction

In this lab you will build a fault-tolerant key/value storage service using your Raft library from Lab 3. Your key/value service will be a replicated state machine, consisting of several key/value servers that each maintain a database of key/value pairs, as in Lab 2, but additionally use Raft for replication. Your key/value service should continue to process client requests as long as a majority of the servers are alive and can communicate, in spite of other failures or network partitions. After Lab 4, you will have implemented all parts (Clerk, Service, and Raft) shown in the diagram of Raft interactions.

Clients will interact with your key/value service in much the same way as Lab 2. In particular, clients can send three different RPCs to the key/value service:

- `Put(key, value)`: replaces the value for a particular key in the database
- `Append(key, arg)`: appends arg to key's value (treating the existing value as an empty string if the key is non-existent)
- `Get(key)`: fetches the current value of the key (returning the empty string for non-existent keys)

Keys and values are strings. Note that unlike in Lab 2, neither `Put` nor `Append` should return a value to the client. Each client talks to the service through a `Clerk` with Put/Append/Get methods. The `Clerk` manages RPC interactions with the servers.

Your service must arrange that application calls to `Clerk` Get/Put/Append methods be linearizable. If called one at a time, the Get/Put/Append methods should act as if the system had only one copy of its state, and each call should observe the modifications to the state implied by the preceding sequence of calls. For concurrent calls, the return values and final state must be the same as if the operations had executed one at a time in some order. Calls are concurrent if they overlap in time: for example, if client X calls `Clerk.Put()`, and client Y calls `Clerk.Append()`, and then client X's call returns. A call must observe the effects of all calls that have completed before the call starts.

Providing linearizability is relatively easy for a single server. It is harder if the service is replicated, since all servers must choose the same execution order for concurrent requests, must avoid replying to clients using state that isn't up to date, and must recover their state after a failure in a way that preserves all acknowledged client updates.

This lab has two parts. In part A, you will implement a replicated key/value service using your Raft implementation, but without using snapshots. In part B, you will use your snapshot implementation from Lab 3D, which will allow Raft to discard old log entries. Please submit each part by the respective deadline.

You should review the extended Raft paper, in particular Sections 7 and 8. For a wider perspective, have a look at Chubby, Paxos Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication, and Bolosky et al.

Start early.

Getting Started

We supply you with skeleton code and tests in `src/kvraft`. You will need to modify `kvraft/client.go`, `kvraft/server.go`, and perhaps `kvraft/common.go`.

To get up and running, execute the following commands. Don't forget the `git pull` to get the latest software.

```
$ cd ~/6.5840
$ git pull
...
$ cd src/kvraft
$ go test
...
$
```

Part A: Key/value service without snapshots (moderate/hard)

Each of your key/value servers ("kvservers") will have an associated Raft peer. Clerks send `Put()`, `Append()`, and `Get()` RPCs to the kvserver whose associated Raft is the leader. The kvserver code submits the Put/Append/Get operation to Raft, so that the Raft log holds a sequence of Put/Append/Get operations. All of the kvservers execute operations from the Raft log in order, applying the operations to their key/value databases; the intent is for the servers to maintain identical replicas of the key/value database.

A `Clerk` sometimes doesn't know which kvserver is the Raft leader. If the `Clerk` sends an RPC to the wrong kvserver, or if it cannot reach the kvserver, the `Clerk` should re-try by sending to a different kvserver. If the key/value service commits the operation to its Raft log (and hence applies the operation to the key/value state machine), the leader reports the result to the `Clerk` by responding to its RPC. If the operation failed to commit (for example, if the leader was replaced), the server reports an error, and the `Clerk` retries with a different server.

Your kvservers should not directly communicate; they should only interact with each other through Raft.

TASK

Your first task is to implement a solution that works when there are no dropped messages, and no failed servers.

Feel free to copy over your client code from Lab 2 (`kvsrv/client.go`) into `kvraft/client.go`. You will need to add logic for deciding which kvserver to send each RPC to. Recall that `Append()` no longer returns a value to the Clerk.

You'll also need to implement `Put()`, `Append()`, and `Get()` RPC handlers in `server.go`. These handlers should enter an `Op` in the Raft log using `Start()`; you should fill in the `Op` struct definition in `server.go` so that it describes a Put/Append/Get operation. Each server should execute `Op` commands as Raft commits them, i.e. as they appear on the `applyCh`. An RPC handler should notice when Raft commits its `Op`, and then reply to the RPC.

You have completed this task when you **reliably** pass the first test in the test suite: "One client".

- **Hint:** After calling `Start()`, your kvservers will need to wait for Raft to complete agreement. Commands that have been agreed upon arrive on the `applyCh`. Your code will need to keep reading `applyCh` while `Put()`, `Append()`, and `Get()` handlers submit commands to the Raft log using `Start()`. Beware of deadlock between the kvserver and its Raft library.
- **Hint:** A kvserver should not complete a `Get()` RPC if it is not part of a majority (so that it does not serve stale data). A simple solution is to enter every `Get()` (as well as each `Put()` and `Append()`) in the Raft log. You don't have to implement the optimization for read-only operations that is described in Section 8.
- **Hint:** You should not need to add any fields to to the Raft `ApplyMsg`, or to Raft RPCs such as `AppendEntries`, but you are allowed to do so.
- **Hint:** It's best to add locking from the start because the need to avoid deadlocks sometimes affects overall code design. Check that your code is race-free using `go test -race`.

Now you should modify your solution to continue in the face of network and server failures. One problem you'll face is that a `Clerk` may have to send an RPC multiple times until it finds a kvserver that replies positively. If a leader fails just after committing an entry to the Raft log, the `Clerk` may not receive a reply, and thus may re-send the request to another leader. Each call to `Clerk.Put()` or `Clerk.Append()` should result in just a single execution, so you will have to ensure that the re-send doesn't result in the servers executing the request twice.

TASK

Add code to handle failures, and to cope with duplicate `Clerk` requests, including situations where the `Clerk` sends a request to a kvserver leader in one term, times out waiting for a reply, and re-sends the request to a new leader in another term. The request should execute just once. These notes include guidance on duplicate detection. Your code should pass the `go test -run 4A` tests.

- **Hint:** Your solution needs to handle a leader that has called Start() for a Clerk's RPC, but loses its leadership before the request is committed to the log. In this case you should arrange for the Clerk to re-send the request to other servers until it finds the new leader. One way to do this is for the server to detect that it has lost leadership, by noticing that Raft's term has changed or a different request has appeared at the index returned by Start(). If the ex-leader is partitioned by itself, it won't know about new leaders; but any client in the same partition won't be able to talk to a new leader either, so it's OK in this case for the server and client to wait indefinitely until the partition heals.
- **Hint:** You will probably have to modify your Clerk to remember which server turned out to be the leader for the last RPC, and send the next RPC to that server first. This will avoid wasting time searching for the leader on every RPC, which may help you pass some of the tests quickly enough.
- **Hint:** You should use a duplicate detection scheme similar to Lab 2. It should free server memory quickly, for example by having each RPC imply that the client has seen the reply for its previous RPC. It's OK to assume that a client will make only one call into a Clerk at a time. You may find that you need to make changes to what information you store in your duplicate detection table from Lab 2.

Your code should now pass the Lab 4A tests, like this:

```
$ go test -run 4A
Test: one client (4A) ...
  ... Passed --  15.5  5  4576  903
Test: ops complete fast enough (4A) ...
  ... Passed --  15.7  3  3022    0
Test: many clients (4A) ...
  ... Passed --  15.9  5  5884 1160
Test: unreliable net, many clients (4A) ...
  ... Passed --  19.2  5  3083  441
Test: concurrent append to same key, unreliable (4A) ...
  ... Passed --   2.5  3   218   52
Test: progress in majority (4A) ...
  ... Passed --   1.7  5   103    2
Test: no progress in minority (4A) ...
  ... Passed --   1.0  5   102    3
Test: completion after heal (4A) ...
  ... Passed --   1.2  5    70    3
Test: partitions, one client (4A) ...
  ... Passed --  23.8  5  4501  765
Test: partitions, many clients (4A) ...
  ... Passed --  23.5  5  5692  974
Test: restarts, one client (4A) ...
  ... Passed --  22.2  5  4721  908
Test: restarts, many clients (4A) ...
  ... Passed --  22.5  5  5490 1033
Test: unreliable net, restarts, many clients (4A) ...
  ... Passed --  26.5  5  3532  474
Test: restarts, partitions, many clients (4A) ...
  ... Passed --  29.7  5  6122 1060
Test: unreliable net, restarts, partitions, many clients (4A) ...
  ... Passed --  32.9  5  2967  317
Test: unreliable net, restarts, partitions, random keys, many clients (4A) ...
  ... Passed --  35.0  7  8249  746
PASS
ok      6.5840/kvraft   290.184s
```

The numbers after each `Passed` are real time in seconds, number of peers, number of RPCs sent (including client RPCs), and number of key/value operations executed (`Clerk` Get/Put/Append calls).

Part B: Key/value service with snapshots (hard)

As things stand now, your key/value server doesn't call your Raft library's `Snapshot()` method, so a rebooting server has to replay the complete persisted Raft log in order to restore its state. Now you'll modify kvserver to cooperate with Raft to save log space, and reduce restart time, using Raft's `Snapshot()` from Lab 3D.

The tester passes `maxraftstate` to your `StartKVServer()`. `maxraftstate` indicates the maximum allowed size of your persistent Raft state in bytes (including the log, but not including snapshots). You should compare `maxraftstate` to `persister.RaftStateSize()`. Whenever your key/value server detects that the Raft state size is approaching this threshold, it should save a snapshot by calling Raft's `Snapshot`. If `maxraftstate` is -1, you do not have to snapshot. `maxraftstate` applies to the GOB-encoded bytes your Raft passes as the first argument to to `persister.Save()`.

TASK

Modify your kvserver so that it detects when the persisted Raft state grows too large, and then hands a snapshot to Raft. When a kvserver server restarts, it should read the snapshot from `persister` and restore its state from the snapshot.

- **Hint:** Think about when a kvserver should snapshot its state and what should be included in the snapshot. Raft stores each snapshot in the persister object using `Save()`, along with corresponding Raft state. You can read the latest stored snapshot using `ReadSnapshot()`.
- **Hint:** Your kvserver must be able to detect duplicated operations in the log across checkpoints, so any state you are using to detect them must be included in the snapshots.
- **Hint:** Capitalize all fields of structures stored in the snapshot.
- **Hint:** You may have bugs in your Raft library that this lab exposes. If you make changes to your Raft implementation make sure it continues to pass all of the Lab 3 tests.
- **Hint:** A reasonable amount of time to take for the Lab 4 tests is 400 seconds of real time and 700 seconds of CPU time. Further, `go test -run TestSnapshotSize` should take less than 20 seconds of real time.

Your code should pass the 4B tests (as in the example here) as well as the 4A tests (and your Raft must continue to pass the Lab 3 tests).

```
$ go test -run 4B
Test: InstallSnapshot RPC (4B) ...
  ... Passed --   4.0  3   289   63
Test: snapshot size is reasonable (4B) ...
  ... Passed --   2.6  3  2418  800
Test: ops complete fast enough (4B) ...
  ... Passed --   3.2  3  3025    0
Test: restarts, snapshots, one client (4B) ...
  ... Passed --  21.9  5 29266 5820
Test: restarts, snapshots, many clients (4B) ...
  ... Passed --  21.5  5 33115 6420
Test: unreliable net, snapshots, many clients (4B) ...
  ... Passed --  17.4  5  3233  482
Test: unreliable net, restarts, snapshots, many clients (4B) ...
  ... Passed --  22.7  5  3337  471
Test: unreliable net, restarts, partitions, snapshots, many clients (4B) ...
  ... Passed --  30.4  5  2725  274
Test: unreliable net, restarts, partitions, snapshots, random keys, many clients (4B) ...
  ... Passed --  37.7  7  8378  681
PASS
ok      6.5840/kvraft   161.538s
```