# 6.824 - Spring 2016

# 6.824 Lab 3: Fault-tolerant Key/Value Service

### Due Part A: Fri Mar 11 11:59pm

### Due Part B: Fri Apr 1 11:59pm

---

## Introduction

In this lab you will build a fault-tolerant key-value storage service using your Raft library from lab 2. You will build your key-value service as a replicated state machine, consisting of several key-value servers that coordinate their activities through the Raft log. 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.

Your system will consist of clients and key/value servers, where each key/value server also acts as a Raft peer. Clients send `Put()`, `Append()`, and `Get()` RPCs to key/value servers (called kvraft servers), who then place those calls into the Raft log and executes them in order. A client can send an RPC to any of the kvraft servers, but should retry by sending to a different server if the server is not currently a Raft leader, or if there's a failure. If the operation is committed to the Raft log (and hence applied to the key/value state machine), its result is reported to the client. If the operation failed to commit (for example, if the leader was replaced), the server reports an error, and the client retries with a different server.

This lab has two parts. In part A, you will implement the service without worrying that the Raft log can grow without bound. In part B, you will implement snapshots (Section 7 in the paper), which will allow Raft to garbage collect old log entries.

- **Hint:** This lab doesn't require you to write much code, but you will most likely spend a substantial amount of time thinking and staring at debugging logs to figure out why your implementation doesn't work. Debugging will be more challenging than in the Raft lab because there are more components that work asynchronously of each other. Start early.
- **Hint:** You should reread the extended Raft paper, in particular Sections 7 and 8. For a wider perspective, have a look at Chubby, Raft Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication, and Bolosky et al.

## 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

Do a `git pull` to get the latest lab software. We supply you with new skeleton code and new 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:

```
$ setup ggo_v1.5
$ cd ~/6.824
$ git pull
...
$ cd src/kvraft
$ GOPATH=~/6.824
$ export GOPATH
$ go test
...
$
```

When you're done, your implementation should pass all the tests in the `src/kvraft` directory:

```
$ go test
Test: One client ...
  ... Passed
Test: concurrent clients ...
  ... Passed
Test: unreliable ...
  ... Passed
...
PASS
ok      kvraft  345.032s
```

## Part A: Key/value service without log compaction

The service supports three RPCs: `Put(key, value)`, `Append(key, arg)`, and `Get(key)`. It maintains a simple database of key/value pairs. `Put()` replaces the value for a particular key in the database, `Append(key, arg)` appends arg to key's value, and `Get()` fetches the current value for a key. An `Append` to a non-existant key should act like `Put`.

You will implement the service as a replicated state machine consisting of several kvservers. Your kvraft client code (`Clerk` in `src/kvraft/client.go`) should try different kvservers it knows about until one responds positively. As long as a client can contact a kvraft server that is a Raft leader in a majority partition, its operations should eventually succeed.

**TASK**

Your first task is to implement a solution that works when there are no dropped messages, and no failed servers. Note that your service 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 `kvraft/client.go` must appear to have affected all kvservers in the same order, and have at-most-once semantics. A `Clerk.Get(key)` should see the value written by the most recent `Clerk.Put(key, …)` or `Clerk.Append(key, …)` (in the total order).

A reasonable plan of attack may be to first fill in the `Op` struct in `server.go` with the

"value" information that kvraft will use Raft to agree on (remember that `Op` field names must start with capital letters, since they will be sent through RPC), and then implement the `PutAppend()` and `Get()` handlers in `server.go`. The handlers should enter an `Op` in the Raft log using `Start()`, and should reply to the client when that log entry is committed. Note that you **cannot** execute an operation until the point at which it is committed in the log (i.e., when it arrives on the Raft `applyCh`).

You have completed this task when you **reliably** pass the first test in the test suite: "One client". You may also find that you can pass the "concurrent clients" test, depending on how sophisticated your implementation is.

**Note:** Your kvraft servers should not directly communicate; they should only interact with each other through the Raft log.

- **Hint:** After calling `Start()`, your kvraft servers will need to wait for Raft to complete agreement. Commands that have been agreed upon arrive on the `applyCh`. You should think carefully about how to arrange your code so that your code will keep reading `applyCh`, while `PutAppend()` and `Get()` handlers submit commands to the Raft log using `Start()`. It is easy to achieve deadlock between the kvserver and its Raft library.
- **Hint:** Your solution needs to handle the case in which a leader has called Start() for a client RPC, but loses its leadership before the request is committed to the log. In this case you should arrange for the client 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 a different request has appeared at the index returned by Start(), or that the term reported by Raft.GetState() has changed. 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 client 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:** A kvraft server 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.

In the face of unreliable connections and node failures, your clients may send RPCs multiple times until it finds a kvraft server that replies positively. One consequence of this is that you must ensure that each application call to `Clerk.Put()` or `Clerk.Append()` must appear in that order just once (i.e., write the key/value database just once).

**TASK**

Add code to cope with duplicate client requests, including situations where the client sends a request to a kvraft leader in one term, times out waiting for a reply, and re-sends the request to a new leader in another term. The client request should always execute just once. To pass part A, your service should **reliably** pass all tests through `TestPersistPartitionUnreliable()`.

- **Hint:** You will need to uniquely identify client operations to ensure that they execute just once. You can assume that each clerk has only one outstanding `Put`, `Get`, or `Append`.
- **Hint:** You must 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.

## Part B: Key/value service with log compaction

In order to allow Raft to discard old log entries so that the log doesn't grow without bounds, you will implement snapshots as described in Section 7 of extended Raft paper. Section 7 provides only an outline; you will have to figure out the details.

You should spend some time figuring out what the interface will be between your Raft library and your service so that your Raft library can discard log entries. Think about how your Raft will operate while storing only the tail of the log, and how it will discard old log entries. You should discard them in a way that causes the underlying memory to be free (so that the Go garbage collector can re-use the memory).

The kvraft 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). Whenever your key/value server detects that the Raft state size is approaching this threshold, it should save a snapshot, and tell the Raft library that it has snapshotted, so that Raft can discard old log entries.

**TASK**

Modify your Raft library (in `src/raft/raft.go`) so that it can discard old log entries and still operate with only the tail of the log. Make sure you pass all the Raft tests after making these changes.

**TASK**

Modify your kvraft server so that it detects when the persisted Raft state grows too large, and then saves a snapshot and tells Raft that it can discard old log entries. Save each snapshot with persister.SaveSnapshot() (don't use files).

**TASK**

Modify your Raft leader code to send an InstallSnapshot RPC to a follower when the leader has discarded the log entries the follower needs. When a follower receives an InstallSnapshot RPC, your Raft code will need to send the included snapshot to its kvraft. You can use the `applyCh` for this purpose — see the `UseSnapshot` field. Your solution is complete when you pass the remaining tests **reliably**.

**Note:** The `maxraftstate` limit applies to the GOB-encoded bytes your Raft passes to `persister.SaveRaftState()`.

- **Hint:** Think about when a kvserver should snapshot its state and what should be included in the snapshot. You must store each snapshot in the persister object using

`SaveSnapshot()`, **not** along with the other Raft state in `SaveRaftState()`. You can read the latest stored snapshot using `ReadSnapshot()`.

- **Hint:** Remember that your kvserver must be able to detect duplicate client requests across checkpoints, so any state you are using to detect them must be included in the snapshots.
- **Hint:** Make sure you pass `TestSnapshotRPC` before moving on to the other Snapshot tests.

## Handin procedure

**Important:**

Before submitting, please run *all* the tests one final time. **You** are responsible for making sure your code works.

```
$ go test
```

Submit your code via the class's submission website, located at 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 the lab from the console as follows.

```
$ cd "$GOPATH"
$ echo "XXX" > api.key
$ make lab3
```

**Important:**

Check the submission website to make sure you submitted a working lab!

**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.*