- **Due: Oct 17, 2023, 23:59 PDT**

## Introduction

In this lab, you will build a fault-tolerant key/value storage service using your Raft library from the previous lab. Your key/value service will be a replicated state machine, consisting of several key/value servers that use Raft to maintain 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.

The service supports three operations: 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-existent key should act like Put.

Your service must provide strong consistency to applications calls to the Get/Put/Append methods. Here's what we mean by strong consistency. 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 Put(), then client Y calls Append(), and then client X's call returns. Furthermore, a call must observe the effects of all calls that have completed before the call starts (so we are technically asking for linearizability).

Strong consistency is convenient for applications because it means that, informally, all clients see the same state and they all see the latest state. Providing strong consistency 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, and must avoid replying to clients using state that isn't up to date.

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.

## Getting Started

Go to the same repo you used for lab 1. Let's create a separate branch for your lab 2 submission and get the template code for lab2 software.

```
$ git checkout lab-raft-solution # ensure you are in the raft solution branch
first.
```

```
$ git am < dslab_lab2.patch # apply the patch file
$ rm dslab_lab2.patch # clean up
$ git add .
$ git commit -m "init lab2"
$ git push --set-upstream origin lab-kv-solution # push updates to the remote
repository
```

We supply you with skeleton code and tests in src/kv. **You will need to modify kv/server.cc/h**. Don't change other files.

To get up and running, execute the following commands:

```
$ python3 waf configure build --enable-raft-test # same build command
$ build/deptran_server -f config/kv_lab_test.yml # different run command
```

...

### ARM Mac users (unsupported by staff):

This lab is tested on x64 version of linux, but you may try ubuntu ARM as well. It looks the Multipass setup on this website installs aarch64 Ubuntu for ARM Mac computers.
For compilation, please use the following command instead.

```
$ python3 waf configure build --enable-raft-test --boost-libs=/usr/lib/aarch64-
linux-gnu
```

## The Code

Each of your key/value servers ("kvservers") will have an associated Raft peer. Clients 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 client sometimes doesn't know which kvserver is the Raft leader. If the client sends an RPC to the wrong kvserver, or if it cannot reach the kvserver, the client 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 client 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 client retries with a different server.

Your first task is to implement a solution that works when there are no dropped messages, and no fa l servers. You'll need to implement the Get/Put/Append functions in the server. These functions should submit log entries by using the Start() function you implemented in the previous lab. In the you can pass the "concurrent clients" test, depending on how sophisticated your implementation is. Your kvservers should not directly communicate; they should only interact with each other through the Raft log.

- After calling Start(), your kvservers will need to wait for Raft to complete agreement. Commands that have been agreed upon arrive on the `OnNextCommand` function call. You should think carefully about how to arrange your code so that it will keep processing the nextcommands, while Put/Append() and Get() handlers submit commands to the Raft log using Start().
- For kvservers, Put/Append/Get/OnNextCommand are all called in the same thread; this is the same thread as the RaftServer RPC handling; thread-blocking calls in any of these will block others. So be careful when you trigger such calls. You can use coroutines in this thread.
- For the Put/Get/Append operations, return `KV_SUCCESS` if successful, KV_NOTLEADER if the server is not a leader, or `KV_TIMEOUT` in any other case.
- A kvserver should not complete a Get() RPC if it is not part of a majority (so that it does not serve stale data). Return KV_TIMEOUT in this case. 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.

Your code should now pass the tests, like this:

```
$ build/deptran_server -f config/kv_lab_test.yml
TEST 1: Basic kv operations
TEST 1 Passed
TEST 2: Concurrent kv operations
TEST 2 Passed
TEST 3: Progress with a majority
TEST 3 Passed
TEST 4: No progress with a minority
TEST 4 Passed
TEST 5: Kv operations through re-election
TEST 5 Passed
TEST 6: Progress with a majority and concurrent requests
TEST 6 Passed
TEST 7: Progress with a majority writing the same key
TEST 7 Passed
TEST 8: Progress with a majority testing linearizability
TEST 8 Passed
TEST 9: Progress in unreliable net
TEST 9 Passed
ALL TESTS PASSED
```

## Submission

Leave your latest modification and implementation on the "`lab-kv-solution`" branch. Please do

## DO NOT

Make sure you do not do any of the following; otherwise, you will fail the labs, even if the code may pass the testing scripts.

- Change/create files other than the files allowed.
- Use file/network/IPC interface other than the RPC interface given to you.

Acknowledgment

Thank Shuai Mu, Julie Lee for sharing this lab. It was originally inspired by MIT 6.824, and the code is based on the academic prototypes of previous research works, including Rococo/Janus/Snow/DRP/Rolis/DepFast.