

## Assignment 4: Key-Value Storage Service

Deadline: Thursday, 9<sup>th</sup> December 2021 at 11:55 PM

In this assignment, you will build a fault-tolerant key-value storage service using your Raft library from the previous assignments. Your key-value service will be structured as a replicated state machine with 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.

**Note:** You should post any assignment related query only on Campuswire. Do not directly email the course staff.

**Note:** Course policy about **plagiarism** is as follows:

- Students must not share the actual program code with other students.
- Students must be prepared to explain any program code they submit.
- Students must indicate with their submission any assistance received.
- All submissions are subject to plagiarism detection.
- Students cannot copy code from the Internet.
- Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

### Overview

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 execute them in order. A client can send an RPC to any of the kvraft servers, but if that server is not currently a Raft leader, or if there's a failure, the client should retry by sending it to a different server. 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.

## Software

**Make sure to place `src/raft`, `src/labrpc`, and `src/kvraft` in your `$GOPATH` folder.**

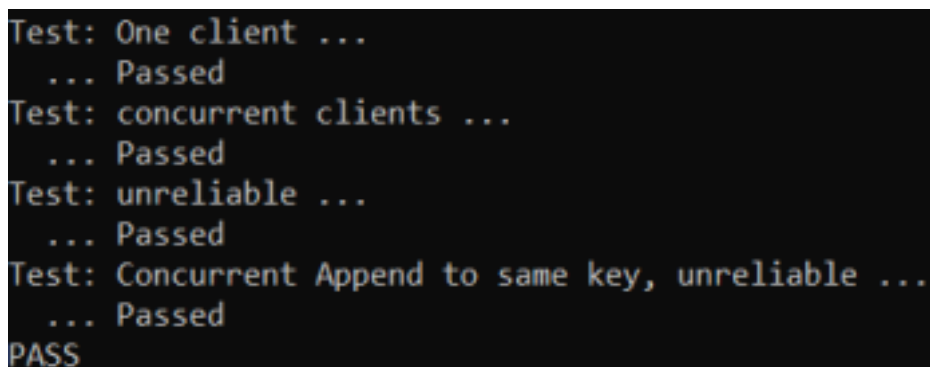
```
\---src
+---kvraft
| client.go (Implement this file)
| common.go (Modify if required)
| config.go (Used by test cases, no need to modify)
| server.go (Implement this file)
| test_test.go (Test cases for your implementation, no need to modify)
+---labrpc
| labrpc.go (RPC like system using go channels, no need to modify)
| test_test.go (For testing RPCs, no need to modify)
|
\---raft
```

### Your raft's implementation in this folder

```
config.go
persister.go
raft.go
test_test.go
util.go
```

We have supplied you with skeleton code and tests under the `src/kvraft`. You will need to modify **`kvraft/client.go`**, **`kvraft/server.go`**, and perhaps **`kvraft/common.go`**. (Even if you don't modify `common.go`, you should submit it as-provided.) You will be using your implementation of the raft for this assignment.

To execute the test cases, run “**go test**” in the **`kvraft`** folder. The following figure shows the passing of all four test cases.



```
Test: One client ...
... Passed
Test: concurrent clients ...
... Passed
Test: unreliable ...
... Passed
Test: Concurrent Append to same key, unreliable ...
... Passed
PASS
```

All the four test cases typically take 50 seconds for execution but this time can vary from system to system.

## Your Task

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 the key's value, and **`Get()`** fetches the current value for a key. An **`Append`** to a non-existent 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.

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

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**).

After calling **Start()**, your kvraft servers will need to wait for Raft to complete the 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.

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 resend 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. More generally, a kvraft server should not complete a **Get()** RPC if it is not part of a majority.

If your implementation is correct, you should be able to 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.

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).

You have 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.

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.

For stability, 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.

## Resources and Advice

This assignment 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 assignment 3 because more components work asynchronously with each other. Start early!

You should implement the service without worrying about the Raft log's growing without bound. You do not need to implement snapshots (from Section 7 in the paper) to allow garbage collection of old log entries.

As noted, 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 are described in Section 8.

You should probably 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 in every RPC.

## Evaluation

You can earn up to 25 points from this assignment. There is no extra credit or bonus.

1. **Test one client with a reliable network** - 5 Points
2. **Test concurrent clients with a reliable network**- 5 Points
3. **Test multiple clients with an unreliable network** - 5 Points
4. **Test concurrent Append to the same key with an unreliable network** - 5 Points
5. **Go Formatting** - 1 point
6. **Manual Grading** - 4 points

You can obtain full points from this if you satisfy these conditions:

- Your submission only uses the permitted packages.
- Your submission has a good coding style that adheres to the formatting and

naming conventions of Go.

## Submission

Submit the **server.go**, **client.go**, and **common.go** in a folder (with folder name equal to your roll number **R**, e.g., **22100112.zip**) on LMS.

**Late Submissions:** You can submit your Assignment by the final day of classes which is 14 December. You can either utilize your remaining grace days which will not result in any penalty or submit it late based on the following penalties.

- 90% for work submitted up to 24 hours late
- 80% for work submitted up to 2 days late
- 70% for work submitted up to 3 days late
- 60% for work submitted up to 5 days late
- 50% for work submitted after 5 days late