# Lab 2

Start Assignment

- Due Sep 10 by 11:59pm
- Points 100
- Submitting a file upload
- File Types gz

## Collaboration Policy

Before starting this assignment, you must read the syllabus' Collaboration and Cheating section. It is critical that you understand the ways in which your work cannot be shared (e.g. including as training for Generative AI tools).

## Introduction

In this lab you will build a key/value server for a single machine that ensures that each operation is executed exactly once despite network failures and that the operations are **linearizable** **(https://pdos.csail.mit.edu/6.824/papers/linearizability-faq.txt)** . Later labs will replicate a server like this one to handle server crashes.

Clients can send three different RPCs to the key/value server: `Put(key, value)`, `Append(key, arg)`, and `Get(key)`. The server maintains an in-memory map of key/value pairs. Keys and values are strings. `Put(key, value)` installs or replaces the value for a particular key in the map, `Append(key, arg)` appends arg to key's value *and* returns the old value, and `Get(key)` fetches the current value for the key. A `Get` for a non-existent key should return an empty string. An `Append` to a non-existent key should act as if the existing value were a zero-length string. Each client talks to the server through a `Clerk` with Put/Append/Get methods. A `Clerk` manages RPC interactions with the server.

Your server must arrange that application calls to `Clerk` Get/Put/Append methods be linearizable. If client requests aren't concurrent, each client Get/Put/Append 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.

Linearizability is convenient for applications because it's the behavior you'd see from a single server that processes requests one at a time. For example, if one client gets a successful response from the server for an update request, subsequently launched reads from other clients are guaranteed to see the effects of that update. Providing linearizability is relatively easy for a single server.

## Getting Started

You need to **setup Go (https://utah.instructure.com/courses/985754/assignments/14449670)** to do the labs. Fetch the initial lab software with **git (https://git-scm.com/)** (a version control system). To learn more about git, look at the **Pro Git book** (https://git-scm.com/book/en/v2) (https://git-scm.com/book/en/v2) or the **git user's manual.** (http://www.kernel.org/pub/software/scm/git/docs/user-manual.html)

```
$ git clone https://github.com/rstutsman/cs6450-labs.git
$ cd cs6450-labs
$ ls
Makefile src
```

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

```
$ cd src/kvsrv
$ go test
...
$
```

## Key/value server with no network failures (easy (https://pdos.csail.mit.edu/6.824/labs/guidance.html) )

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

You'll need to add RPC-sending code to the Clerk Put/Append/Get methods in `client.go`, and implement `Put`, `Append()` and `Get()` RPC handlers in `server.go`.

You have completed this task when you pass the first two tests in the test suite: "one client" and "many clients".

- Check that your code is race-free using `go test -race`.

## Key/value server with dropped messages (easy (https://pdos.csail.mit.edu/6.824/labs/guidance.html) )

Now you should modify your solution to continue in the face of dropped messages (e.g., RPC requests and RPC replies). If a message was lost, then the client's `ck.server.Call()` will return `false` (more precisely, `Call()` waits for a reply message for a timeout interval, and returns false if no reply arrives within that time). One problem you'll face is that a `Clerk` may have to send an RPC multiple times until it succeeds. Each call to `Clerk.Put()` or `Clerk.Append()`, however, should result in just a *single* execution, so you will have to ensure that the re-send doesn't result in the server executing the request twice.

Add code to `Clerk` to retry if doesn't receive a reply, and to `server.go` to filter duplicates if the operation requires it.

Hints:

- You will need to uniquely identify client operations to ensure that the key/value server executes each one just once.
- You will have to think carefully about what state the server must maintain for handling duplicate `Get()`, `Put()`, and `Append()` requests, if any at all.
- Your scheme for duplicate detection 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.

Your code should now pass all tests, like this:

```
$ go test
Test: one client ...
  ... Passed -- t  3.8 nrpc 31135 ops 31135
Test: many clients ...
  ... Passed -- t  4.7 nrpc 102853 ops 102853
Test: unreliable net, many clients ...
  ... Passed -- t  4.1 nrpc   580 ops  496
Test: concurrent append to same key, unreliable ...
  ... Passed -- t  0.6 nrpc    61 ops   52
Test: memory use get ...
  ... Passed -- t  0.4 nrpc     4 ops    0
Test: memory use put ...
  ... Passed -- t  0.2 nrpc     2 ops    0
Test: memory use append ...
  ... Passed -- t  0.4 nrpc     2 ops    0
Test: memory use many puts ...
  ... Passed -- t 11.5 nrpc 100000 ops    0
Test: memory use many gets ...
  ... Passed -- t 12.2 nrpc 100001 ops    0
PASS
ok      6.5840/kvsrv    39.000s
```

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

# Handin procedure

Before submitting, please run *all* the tests one final time. **You** are responsible for making sure your code works. The late day policy is stated on the syllabus. Please discuss/ask questions on the **Lab 2 channel in MS Teams (https://teams.microsoft.com/l/channel/19%3A440bb6d5ec974fa7871e10c5ffd3f318%40thread.tacv2/Lab%202?groupId=ebbe7cd7-cfc6-4d05-b84f-6164f484dc15&tenantId=5217e0e7-539d-4563-b1bf-7c6dcf074f91)**.

When you are ready to turn in your solution, create the submission file like this:

```
$ cd ..
$ make lab2
```

This will create `lab2-handin.tar.gz` which you upload here on Canvas.