# Project 0: Implementing a key-value messaging system

Due Thursday, September 13th, 2018

The goal of this assignment is to get up to speed on the Go programming language and to help remind you about the basics of socket programming that you learned in 15-213. In this assignment you will implement a key-value database server in Go, with a slight twist to it: The values are lists of messages which can be retrieved with a *get()* query and added to with a *put()* query. Keys can be completely deleted with a *delete()* query. You can think of this as the basis for extremely simple online messaging system where every client is allowed to read every other client's messages.

## Server Characteristics

We have provided starter code for the key-value server which provide these basic operations:

1. **init_db()** - This function creates an empty database.
2. **put(K string, V []byte)** - This function inserts a value onto the end of the message list (actually a slice) associated with the key $K$. Note that the slice can contain any number of values at one time.
3. **get(K string) []([]byte)** - This function returns the slice of all values associated with key $K$.
4. **delete(K string)** - This function deletes all the messages associated with the key $K$.

You can assume that all keys and values are of the form **[a-z][a-z0-9_]***.

## Requirements

This project is intentionally open-ended and there are many possible solutions. That said, your implementation must meet the following requirements:

1. The server **must** manage and interact with its clients concurrently using Goroutines and channels. Multiple clients should be able to connect/disconnect to the server simultaneously.

2. When a client wants to put a value into the server, it sends the following request string: **put,key,value**. When a client wants to get a value from the server, it sends the following request string: **get,key**. When a client wants to delete a key from the server, it sends the following request string: **delete,key**. These are the only possible messages from the client. You will be responsible for parsing the request string and selecting the appropriate operation.

3. When the server reads a *get()* request message from a client, it should respond with the following string: **key,value**[**newline**] to the client that sent the message. **It should send this string once for every value associated with the given key.** No response should be sent for a *put()* request.

4. Every message that is sent or received should be terminated by the newline (`\n`) character.

5. The server **should not** assume that the key-value API functions listed above are thread-safe. **You will be responsible for ensuring that there are no race conditions while accessing the database.**

6. The server **must** implement a **CountActive()** function that returns the number of clients that are currently connected to it.

7. The server **must** implement a **CountDropped()** function that returns the number of clients that have been disconnected from and thus dropped from the server.

8. The server **must** be responsive to slow-reading clients. To better understand what this means, consider a scenario in which a client does not call `Read` for an extended period of time. If during this time the server continues to write messages to the client's TCP connection, eventually the TCP connections output buffer will reach maximum capacity and subsequent calls to `Write` made by the server will block. To handle these cases, your server should keep a queue of at most **500** outgoing messages to be written to the client at a later time. Messages sent to a slow-reading client whose outgoing message buffer has reached the maximum capacity of 500 should simply be dropped. If the slow-reading client starts reading again in the future, the server should ensure that any buffered messages in its queue are written back to the client (hint: use a buffered channel to implement this properly).

9. All work is to be done individually. You may not hand in any code that you have not written yourself and that we have not provided you.

10. Your code *may not* use locks and mutexes. All synchronization must be done using goroutines, channels, and Go's channel-based `select` statement (not to be confused with the low-level socket select that you might use in C, which is also not allowed).

11. You may only use the following packages: `bufio`, `bytes`, `fmt`, `net`, and `strconv`.
12. You must format your code using `go fmt` and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.

We don't expect your solutions to be overly-complicated. As a reference, our sample solution is a little over 200 lines including sparse comments and whitespace. We do, however, *highly recommend* that you familiarize yourself with Go's concurrency model before beginning the assignment. For additional resources, check out the course lecture notes and the Go-related posts on Piazza.

## Hints

1. You should call *init_db()* in the *Start()* function.
2. Your server **can** assume that clients will never do a *get()* request or *delete()* request for keys that aren't already stored on the server.
3. When you put a new value into the database, you can choose to remove the `\n` ending or not - both values will be accepted.

## Instructions

The starter code for this project is hosted as a read-only repository on GitHub (you can view the repository online here). To clone a copy, execute the following Git command:

```
git clone https://github.com/CMU-440-F18/P0.git
```

The starter code consists of three source files located in the `src/github.com/cmu440/p0` directory:

1. `server_impl.go` is the only file you should modify, and is where you will add your code as you implement your multi-client echo server.
2. `kv_impl.go` contains the key-value API that you'll be using to perform operations on your database. These 4 functions should be directly used in `server_impl.go` as you implement your server.

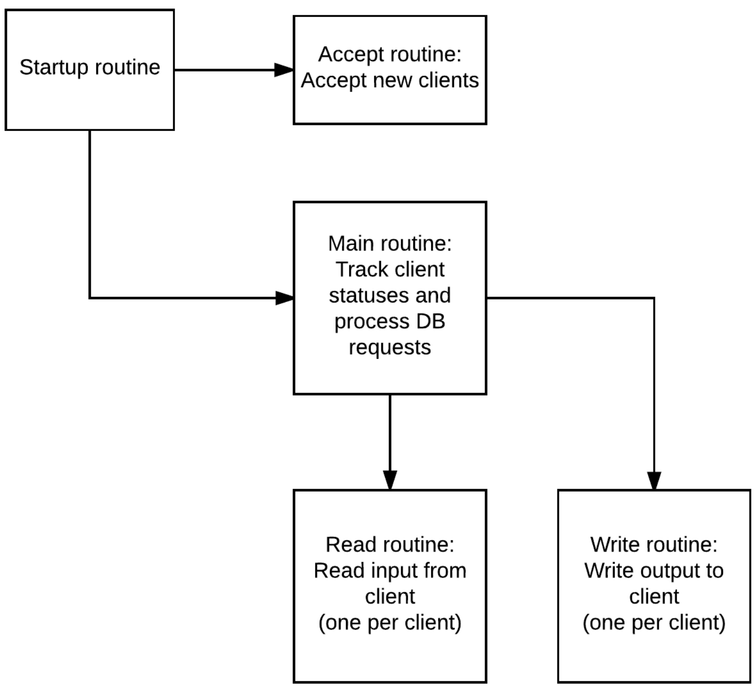
3. `server_api.go` contains the interface and documentation for the key value server you will be implementing in this project. You should **not** modify this file.

4. `server_test.go` contains the tests that we will run to grade your submission.

**For instructions on how to build, run, test, and submit your server implementation, see the `README.md` file in the project's root directory.**

## Structure

Finding the right logical structure for your code is an important aspect concurrent programming. Although you are free to use whatever structure you like, you may find something like the following one easiest:




Here the boxes represent routines, and the arrows are a rough representation of dependence. Note that not all channels that you will need are shown.

## Evaluation

1. You can earn up to 15 points on this project.
   (a) **Basic tests** - 6 points
   (b) **Count tests** - 2 points
   (c) **Slow client tests** - 2 points
   (d) **Go formatting** - 1 point
   (e) **Manual grading** - 4 points
   You can obtain full points from this if you satisfy these conditions:
      i. Your submission does not use mutexes of any form
      ii. Your submission only uses the permitted packages
      iii. Your submission has good coding style that adheres to the Formatting and Naming conventions of Go.
2. You are allowed a maximum of 20 submissions on Autolab. We will only consider the **final** submission for grading.
3. There will be no hidden test cases in this Project. However, you can expect them in future projects.
4. There will be no late days. If you submit late, you will lose 10% of the total grade for each day late you submit it. No submissions will be accepted after September 15th.