Project 0: Implementing a Key-Value database server

Due Thursday, September 22, 2016 at 11:59pm

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 simple key-value database server in Go, with a slight twist to it: the result of every *get()* query sent by any client should be returned to every connected client. A possible practical application for this twist is a set of replicated web servers that have eventually-consistent distributed in-memory state loaded from a centralized key-value store.

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 key-value pair into the store. In case K already has some value in the store, it will be updated to V. This is because every key can only be associated with a unique value.
3. **get(K string) []byte** - This function returns the value stored for key K

You can assume that all keys and values are of the form **[a-z][a-z0-9_]***. Your key-value server must have the following characteristics:

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

   These are the only possible messages that will be sent 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

   to all connected clients, including the client that sent the message. **No response should be sent to any of the clients for a *put* request**.

4. The server **should** assume that *all* messages are line-oriented: every message that is sent or received is terminated by the newline (`\n`) character.

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

6. Your server **can** assume that clients will *never* do a *get()* request for keys that aren't already stored on the server.

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

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 connection's 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 of the buffered messages in its queue is written back to the client. (Hint: use a buffered channel to implement this property).

Requirements

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

1. The project **must** be done individually. You are not allowed to use any code that you have not written yourself.

2. Your code **must 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). Furthermore, solutions using any locking, mutexes or implementing mutex-like behaviour using Go channels will incur a grading penalty.

```
_ =<- my_mutex_channel

// some critical section stuff

my_mutex_channel <- 1
```

   This code snippet is an example of using channels as mutexes, which isn't allowed.

3. You may only use the following packages: `bufio`, `fmt`, `net`, `bytes` and `strconv`.

4. 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.

Sanity Check

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, recitation slides and the Go-related posts on Piazza. **Use the AFS clusters to develop and test your implementations.**

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-F16/p0.git
```

The starter code consists of four 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

2. `kv_impl.go` contains the key-value API that you'll be using to perform operations on your database. These 3 functions should be directly used in server_impl.go as you implement your key-value database server.
3. `server_api.go` contains the interface and documentation for the `KeyValueServer` 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.

Evaluation

- You can earn upto 15 points from this project. There is no extra credit or bonus.
  1. **Basic Tests** - 6 points
  2. **Count Tests** - 2 points
  3. **Slow Client Tests** - 2 points
  4. **Go Formatting** - 1 point
  5. **Manual Grading** - 4 points
     You can obtain full points from this if you satisfy these conditions:
     - Your submission *doesn't* use mutexes of any form. (See Requirements)
     - Your submission only uses the permitted packages. (See Requirements)
     - Your submission has good coding style that adheres to the the Formatting and Naming conventions of Go.
- You are allowed a maximum of **20** submissions on Autolab. We will only consider your **final** submission for grading.
- There will be no hidden test cases in this Project. However, you can expect them in future projects.
- There will be no late days. You will lose 10% of the total grade, each day. Autolab will no longer accept submissions after September 25th, 11:59pm.

Advice and Hints

1. **Start Early!**
2. We will not be holding office hours on deadline day (i.e. 9/22). Another reason for you to **Start Early!**. However, we will answer questions on Piazza even on deadline day.
3. The main goal of this project is for you to develop an intuition of concurrent programming. This would require you to ignore analying the problem in terms of finding patches of code that require locking. We want you to try and figure out how to use Go constructs that inherently provide you with the mutual exclusion that you're looking for, as you will be using this approach for most of your future projects. Focussing on this section of the Tour of Go should suffice for this.
4. Approach this project in a sequential basis. First try to get a single client and server to connect and communicate with each other. Then try to figure out how you can outsource the communication work to Goroutines using Channels. Explore the Tour of Go and see how you can expand your world using the available constructs. You will eventually find that other features can be easily added, incrementally, to finally attempt the Autograder. Get started with coding as soon as possible, as Go would be a new programming language for most of you.
5. We have provided 2 driver programs **srunner** and **crunner**. srunner starts your server on some port and runs forever. crunner can be implemented by you to run simple test clients. Your crunner implementation does not count for the evaluation, but we do recommend using it at early stages of development!
6. Keep in mind the data types being used by the Key-Value API functions.