Implementing a Key-Value Database Server

Deadline 1: Friday, 2th October 2020 at 11:59 PM
Deadline 2: Sunday, 11th October 2020 at 11:59 PM

The goal of this assignment is to get you up to speed on the Go programming language, help remind you about the basics of socket programming and introduce you to Remote Procedure Calls (RPCs) in Go. In this assignment you will implement a simple key-value database server in Go. The clients of this server will either send a $get()$ query to fetch value of a key or a $put()$ request to store a key-value pair.

You will be implementing two versions of the key-value server:

Model 1: A simple socket-based key-value server

Model 2: An RPC-based key-value server

Note: you are required to submit **ONLY** your implementation of the server.

Note: the Submission section explains the deadlines.

Note: This assignment is to be done **individually.**

Note: You should post any assignment related query ONLY on piazza. Do not directly email the course staff.

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

- Students must not share 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.

Server Characteristics: Both Models

To interact with the database, your server should use the following 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.
3. **get**(K string) []byte - This function returns the value stored for key K.

Note: you do **NOT** have to implement these operations. Assume that a database is already implemented and provides this API to the server you will be coding.

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

Your key-value server implementations in **BOTH** models must have the following characteristics:

1. The server **SHOULD NOT** assume that the three key-value API functions listed above (which provide access to the database) are thread-safe. You will be responsible for ensuring that there are no race conditions while accessing the database.

2. Your server **MAY** assume that clients will never do a $get()$ request for keys for which a $put()$ request was never made.

Server Characteristics: Model 1

Your socket-based key-value server must have the following additional 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. $put()$: when a client wants to put a value into the server, it sends the following request string:

   `put,key,value`

   $get()$: 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 **MUST** implement a `Count()` function that returns the number of clients that are *currently* connected to it.

6. 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. 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 (due to flow control in TCP).

   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 (i.e. messages not dropped) in its queue are written back to the client. (Hint: use a buffered channel to implement this property).

Server Characteristics: Model 2

Your RPC-based key-value server must have the following additional characteristics:

1. The server **MUST** implement `RecvPut` and `RecvGet` RPC APIs. Clients interact with the server by making **ONLY** these RPC calls.
2. `RecvPut(args *PutArgs, reply *PutReply) error`: takes a key-value pair as argument, replies with nothing and returns an error if any.
3. `RecvGet(args *GetArgs, reply *GetReply) error`: takes a key as argument, replies with the key's corresponding value from the key-value store and returns an error if any.
4. The server must **NOT** read from or write directly to a client's `bufio.ReadWriter` or `net.Conn` object.
5. The server must **NOT** maintain any client state.
6. The server **MUST** use the RPC protocol defined in `proto.go` and `rpc.go`.

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. Furthermore, solutions using any locking, mutexes or implementing mutex-like behavior using Go channels will incur a grading penalty.

```
_ = <-my_mutex_channel
// some critical code
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:

   Model 1 – bufio, bytes, fmt, io, net and strconv
   Model 2 – fmt, io, net, net/http, net/rpc and strconv

4. You **MUST** format your code using gofmt 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 220 lines including sparse comments and whitespace. We do, however, highly recommend that you familiarize yourself with Go's concurrency model before beginning the assignment.

Instructions

The starter code consists of four source files:

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. Do **NOT** modify this file. But you should read it carefully.
4. *server_test.go* contains the tests that we will run to grade your submission.
5. *rpc.go* contains the interface and documentation for the additional RPC APIs for Model 2. You will be embedding these functions into your `KeyValueServer`. Do **NOT** modify this file. But you should read it carefully.
6. *proto.go* contains the RPC protocols you are required to follow. Argument and reply structs for each RPC method are already implemented for you. Do **NOT** modify this file. But you should read it carefully.

For setting up Go on your systems, please see the Set Up section below. And for instructions on how to build, run, test, and submit your server implementation, see the Logistics section of this document.

Set Up

System

The assignment requires you to use a Linux distro with x86-64 architecture. Please install a VM (e.g. using VirtualBox) if you do not have this set up.

Go installation

The files have been tested using go1.12. You can download it from here.

After installing it, the `GOPATH` environment variable will be set to `$HOME/go`. You can check the value of it using the `go env` command. You might need to create a folder named "go" (without quotation marks) if it's not already there according to the configured `GOPATH`. Inside this folder, create 3 additional folders: "src", "pkg" and "bin" (without quotation marks).

You may write a simple "hello, world" program in `$GOPATH/src` folder to test the set up. To run it directly, execute `go run <filename>.go` from the directory containing the .go file. You may compile it into a binary using `go build <filename>.go` and run it using `./<filename>`.

If you want to set a custom `GOPATH`, please refer this document.

Files

Place the DS_PA1 folder in `$GOPATH/src`. In this folder you will find the following files:

```
crunner/
        crunner.go
pa1/
        kv_impl.go
        server_api.go
        server_impl.go          // the ONLY file you are required to submit
        server_test.go
rpcs/
        rpc.go
        proto.go
srunner/
        srunner.go
```

Test the files

1. From `DS_PA1/pa1` directory, execute `go test`. If everything is running fine, you will see all test cases failing.
2. From `DS_PA1/srunner` directory, execute `go run srunner.go`. If everything is running fine, you will see `New() returned a nil server. Exiting…` printed to Stdout.

Evaluation

- You can earn up to 20 points from this assignment. There is no extra credit or bonus.

  Deadline 1 (total 5 points)

  1. **Basic Tests w/o race conditions** - 5 points

  Deadline 2 (total 15 points)

  1. **Basic Tests** - 6 points
  2. **Count Tests** - 1 points
  3. **Slow Client Tests** - 1 points
  4. **RPC Tests** - 2 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 doesn't use mutexes of any form.
     - Your submission only uses the permitted packages.
     - Your submission has good coding style that adheres to the formatting and naming conventions of Go.

- There will be no hidden test cases in this assignment. However, you can expect them in future assignments.

Late Submissions

You should submit your assignment on LMS before its due time.

If you submit your work late, we will award you a fraction of the score you would have earned on the assignments had it been turned in on time, according to this sliding scale:

- 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

For example, if you should have earned 8/10 points but submitted 36 hours late, you will instead earn 6.4 points.

Besides the above policy, **you are allowed five "free" late days during the semester** (that can be applied to one of assignments 1 through 3; the final assignment will be due tentatively on final day of classes [12 Dec] and cannot be turned late. **The last day to do any late submission is also the final day of classes, even if you have free late days remaining**). You do not need to tell us that you are applying your "late day" -- we'll remove the late penalty at the end of the semester from the assignment that benefits you the most.

Advice and hints

1. **START EARLY!**
2. The main goal of this project is for you to develop an intuition of concurrent programming. This would require you to ignore analyzing 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. Focusing on this section of the Tour of Go (pages 1 to 6) should suffice for this.

   Keep in mind the following as you design your solution:

   a. Each piece of data should be owned by exactly one goroutine to prevent race conditions. E.g. one goroutine to read and write to the key-value database, and only one goroutine which can read from (or write to) a particular client socket.

   b. Goroutines wanting access to a piece of data and the goroutine owning the data should communicate through channels.

   c. Channels **ARE** goroutine-safe (which is similar in meaning to thread-safe) and allow safe concurrent access. But a slice, array or map are **NOT** even if they contain only channels.

3. Approach this project in a sequential basis. For Model 1, 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 graded tests. Get started with coding as soon as possible, as Go would be a new programming language for most of you.

   Note: you may want to refer to *echo_server.go* and *echo_client.go* from the Go tutorial files on LMS to get started.

4. Model 2 is simpler as you do not need to worry about how a server and client will communicate over sockets. This detail is abstracted away when using RPCs. It is suggested you read the sections on `rpc.NewServer()`, `Server.Register(…)`, `Server.HandleHTTP(…)`, `http.Serve(…)`, `rpc.Call(…)` and `rpc.Go(…)` in the official Go RPC documentation here.

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! Alternatively, you may use Netcat for Model 1. See this section.

6. Keep in mind the data types being used by the Key-Value API functions allowing interaction with the database.

Logistics

If at any point you have any trouble with building, installing, or testing your code, the article titled How to Write Go Code is a great resource for understanding how Go workspaces are built and organized. You might also find the documentation for the go command to be helpful.

Running the official tests

To test your submission, we will execute the following command from inside the `$GOPATH/src/DS_PA1/pa1` directory:

```
$ go test
```

We will also check your code for race conditions using Go's race detector by executing the following command:

```
$ go test -race
```

To execute a single unit test, you can use the `-test.run` flag and specify a regular expression identifying the name of the test to run. For example,

```
$ go test -race -test.run TestBasic1
```

Submission

Submit the ***server_impl.go*** file on LMS after renaming it to ***server_impl_<R>.go***, where <***R***> should be replaced by your roll number, e.g. ***server_impl_17030010.go***. Please submit your code with all print statements removed.

Deadline 1:

Your submission will **ONLY** be tested on the six Basic Tests (see Evaluation) but **WITHOUT** any checks on race conditions. That is, the following command will be executed from within the `$GOPATH/src/DS_PA1/pa1` directory:

```
$ go test -test.run TestBasic*
```

Deadline 2:

Your submission will be tested on the entire test suite, **WITH** checks on race conditions. That is, the following two commands will be executed:

```
$ go test
$ go test -race
```

Testing your implementation using srunner

To make testing your server a bit easier (especially during the early stages of your implementation when your server is largely incomplete), we have given you a simple **srunner** (server runner) program that you can use to create and start an instance of your

`KeyValueServer`. The program simply creates an instance of your server, starts it on a default port, and blocks forever, running your server in the background.

To run the **srunner** program, execute the following command from within **`$GOPATH/src/DS_PA1/srunner`** directory:

```
$ go run srunner.go
```

To compile and then run the **srunner** program's binary, execute the following command from within **`$GOPATH/src/DS_PA1/srunner`** directory:

```
$ go build srunner.go
$ ./srunner
```

The **srunner** program won't be of much use to you without any clients. It might be a good exercise to implement your own **crunner** (client runner) program that you can use to connect with and send messages to your server. We have provided you with an unimplemented **crunner** program that you may use for this purpose if you wish. Whether or not you decide to implement a **crunner** program will not affect your grade for this project.

You could **ALTERNATIVELY** test your Model 1 server using Netcat:

1. Run the **srunner** program
2. In another shell execute **`$ nc localhost 9999`** to start up a client which will interact with the server using command line (i.e. Stdout).
3. Type the message you wish to send and press enter

Note: for $put()$ messages, i.e. **`put,<key>,<value>`**, there would be no response from the server.

Additional Challenges

An additional (and **NON-GRADED**) challenge is to make sure there are no memory leakages when your server is closed using the `Close()` function. This means you must make sure all relevant goroutines exit / return.

Notes

When testing using the entire test suite, you may run into the following error during the RPC tests: **`panic: http: multiple registrations for /_goRPC_`**

This is a known bug in the `net/rpc` package when executing multiple RPC endpoints on the same host. To workaround this bug simply add the following line before your `HandleHTTP(…)` call:

```
http.DefaultServeMux = http.NewServeMux()              <-LINE TO ADD
yourRPCserver.HandleHTTP(rpc.DefaultRPCPath, rpc.DefaultDebugPath)
```