Client-Server Token Manager

You are to implement a client-server application in Go for the management of **tokens**.

Token ADT
A token is an abstract data type, with the following properties: id, name, domain, and state. Tokens are uniquely identified by their id, which is a string. The name of a token is another string. The domain of a token consists of three uint64 integers: low, mid, and high. The state of a token consists of two unit64 integers: a partial value and a final value, which is defined at the integer x in the range [low, mid) and [low, high), respectively, that minimizes **h(name, x)** for a hash function h. The hash **h** to use for this project is the SHA256 as below:

```
import (
        "crypto/sha256"
        "encoding/binary"
        "fmt"
)
// Hash concatentates a message and a nonce and generates a hash value.
func Hash(name string, nonce uint64) uint64 {
        hasher := sha256.New()
        hasher.Write([]byte(fmt.Sprintf("%s %d", name, nonce)))
        return binary.BigEndian.Uint64(hasher.Sum(nil))
}
```

Tokens support the following methods:

- **create**(id): create a token with the given id. Reset the token's state to "undefined/null". Return a success or fail response.
- **drop**(id): to destroy/delete the token with the given id
- **write**(id, name, low, high, mid):
  1. set the properties name, low, mid, and high for the token with the given id. Assume uint64 integers low <= mid < high.
  2. compute the partial value of the token as argmin_x H(name, x) for x in [low, mid), and reset the final value of the token.
  3. return the partial value on success or fail response
- **read**(id):
  1. find argmin_x H(name, x) for x in [mid, high)
  2. set the token's final value to the minimum of the value in step#1 and its partial value
  3. return the token's final value on success or fail response

Task

Project 2, CMSC 621, Spring 2022, UMBC [Dr. Kalpakis]

Implement a client-server solution for managing tokens. Your server should maintain an initially empty (non-persistent) collection of tokens. Clients issue RPC calls to the server to execute create, drop, read-write methods on tokens. The server executes such RPCs and returns an appropriate response to each call. Client-server communication is assumed **synchronous**.

Your solution should in developed in Go and utilize the **gRPC** and Google **Protocol Buffer** frameworks.

Your server, upon completing an RPC call by a client for a token, it should "dump" on stderr or stdout all the information it has for that token, followed by a list of the ids of all of its tokens. The server upon starting, it gets the port number to listen to from the command line arguments/flags, eg.

tokenserver -port 50051

The server terminates upon receiving a CTRL-C (SIGINT) signal. Your server should allow for maximum concurrency among non-conflicting RPC calls (eg any pair of concurrent RPC calls on the same token conflict unless both are read); conflicting RPC calls should be executed in a serial manner.

Your client executes a single RPC call (to a server on the specified host and port) upon fetching the command line arguments/flags, eg

tokenclient -create -id 1234 -host localhost -port 50051
tokenclient -write -id 1234 -name abc -low 0 -mid 10 -high 100 -host localhost -port 50051
tokenclient -read -id 1234 -host localhost -port 50051
tokenclient -drop 1234 -host localhost -port 50051

Your client should print on stdout the response it received from the RPC call to the server, and then terminate.

Alternatively, you may have a -config command line flag to specify the filename of a simple YAML (configuration) file with the intended server's connection information, eg.

host: localhost
port: 50051

What to submit:

Submit a .tar.gz archive with your

- complete Go code (and any supporting files)
- a bash script demonstrating running your server and sequence of client executions.
- README file (with relevant documentation and usage guide)

References

- Quickstart for the gRPC Framework
- Tutorial on Google Protocol Buffers
- gRPC with ProtoBuf Basics Tutorial
- Goroutines and gRPC Serve() method of gRPC Server
- sync package: Mutexes in Go or Mutex Tour or RWMutex or Map

- Command Line Flags in Go
- YAML configurations for Go

This project was inspired from a similar project of the 15-440/640, Distributed Systems class at CMU.