Assignment #1. File Server

This assignment is to build a simple file server, with a simple read/write interface (there's no open/delete/rename). The file contents can be in memory, but if you use a database like leveldb to store the files, you get extra nice points! There are two features that this file system has that traditional file systems don't. The first is that each file has a version, and the API supports a "compare and swap" operation based on the version (see below). The second is that files can optionally expire after some time.

Don't try to be overly general, and don't look for reuse.

Due: Thu, Jan 22. (Although it is due in two weeks, we will have a checkpoint in one week **Thu Jan 14**, to ensure that it is not left up to the last moment.

On-the-wire specification

This specification describes an on-the-wire protocol, the exchange of commands and their results on a TCP connection.

The client opens up a TCP connection and sends text commands to insert/update/delete/cas files. Filenames are text strings, and file contents are **binary* strings.

Each command is a line in this format.The options are explained in the next section. The actual data is on the following line.

1. Write: create a file, or update the file's contents if it already exists.

   ```
   write <filename> <numbytes> [<exptime>]\r\n
   <content bytes>\r\n
   ```

   The server responds with the following:

   ```
   OK <version>\r\n
   ```

   where *version* is a unique 64-bit number (in decimal format) assosciated with the filename.

2. Read: Given a filename, retrieve the corresponding file:

   ```
   read <filename>\r\n
   ```

   The server responds with the following format (or one of the errors described later)

```
CONTENTS <version> <numbytes> <exptime> \r\n
<content bytes>\r\n
```

3. Compare and swap. This replaces the old file contents with the new content provided the version is still the same.

   ```
   cas <filename> <version> <numbytes> [<exptime>]\r\n
   <content bytes>\r\n
   ```

   The server responds with the new version if successful (or one of the errors described later)

   ```
     OK <version>\r\n
   ```

4. Delete file

   ```
   delete <filename>\r\n
   ```

   Server response (if successful)

   ```
    OK\r\n
   ```

Options:

1. filename : an ascii text string (max 250 bytes) without spaces.
2. numbytes: length of the content block, not including the trailing `\r\n` . It is in an ascii text format.
3. version: A 64-bit number generated by the server, in ascii text format.
4. exptime: An optional interval (in seconds) after which the content may not be available. The file is not expired if exptime is not specified or if the content is 0.

Errors that can be returned.

1. `ERR_VERSION <newversion>\r\n` (the contents were not updated because of a version mismatch. The latest version is returned)
2. `ERR_FILE_NOT_FOUND\r\n` (the filename doesn't exist)
3. `ERR_CMD_ERR\r\n` (the command is not formatted correctly)
4. `ERR_INTERNAL\r\n` (any other error you wish to report that is not covered by the rest (optional))

A note on \r\n
Like most internet protocols (such as telnet, smtp) this command set is command-line friendly. The `\r\n` pair is there for command-line testing. In this protocol, the first line is actually all text ending in `\r\n` . However, don't let this fool you. The second "line" for the content is not really a text line. It is binary-valued (of length *numbytes*).

Also, keep in mind that TCP is stream-oriented, not a line-oriented protocol. Don't expect io.Read() to always return entire lines or entire commands.

A note on syntactic errors and message framing
While parsing a message, there are some errors that make it impossible to say with certainty where the current message ends and where the next one begins. Examples are: unknown command (not read/write/cas/delete), non-numeric , not finding '\r\n' where expected etc. These errors are unrecoverable because you don't want to make a guess and get the beginning of the next command wrong as well. The only option is for the server to write back an error and to close the socket.

There are other syntactic errors where it is possible to figure out where the current message ends. For example, having more fields than necessary, non-numeric version number or expiry time etc. While the server still reports them as errors, it does not need to terminate the connection.

For the purpose of this assignment, there's no need to go too fancy on syntax validation. It is better to spend your time on building and testing the concurrent behaviour of the file system.

Submission instructions

1. Create a repository on Github or Gitlab (git.cse.iitb.ac.in). For example, say github.com/*myname*
2. On your local machine, the directory path will be under $GOPATH/src/github.com/*myname*/cs733/assignment1. This directory structure is important
3. Under the cs733 directory, create an empty file called ROLL_NO_xxxxxxxxxxx (with your roll number), so that we can automatically identify the repository.
4. Check this file in.
5. Write your file server code in the following form:

```
func serverMain() {
      ... all the stuff you would have normally put into main.
}
```

```
func main() {
     serverMain()
}
```

The purpose of this is to help us mix different people's test cases with different servers.

6. The server should listen on port :8080.

7. Write test files (can call them anything you want.). They should be of the following form:

```
import (
     "testing"
     "time"
)

func TestMain(m *testing.M) {
       go serverMain()    // launch the server as a goroutine.
       time.Sleep(1 * time.Second)
}

... the actual test cases...
```

8. Test with `go test -race` (to detect race conditions). You need your server to be robust and your tests to be creative to find bugs in other people's servers.

9. After you are done committing to the repository, test doing a 'go get' with a different GOPATH and see if the test works for you.

10. A note on testing. I want to emphasize automated testing, which means that nothing should be printed out for manual verification. It is not a demo; it is an automated test. The only output I want to see from `go test` is one line saying `OK` .

11. Create good documentation in the README.md file. **I** am not the audience for the README (I'm merely one who hands out grades :). Think of the audience as someone outside that you want to impress. The documentation must show what you are building and convinces them that it is a well-tested. It is fine to include this document.

12. At all times, please do not hesitate to ask if you find your self getting stuck at any time, including the initial design. Talk to me, to the TAs, or post to the Piazza

forum. Life is too precious to be stuck in a compile-debug cycle.