Assignment 4
CMPE 105: Principles of Computer System Design, Winter 2019

Due: March 15 at 5:00PM

Goals

The goals for Assignment 4 are to modify the HTTP server that you already implemented to add support for *aliases*. An alias is an alternate name that may be used to refer to an object stored in the HTTP server's key-value store. This will involve implementing a version of the PATCH command in HTTP that will add a name as an alias for an existing object.

As usual, you must have a design document and writeup along with your `README.md` in your `git` repository. Your code only needs to build `httpserver` using `make`.

Programming assignment: adding aliases to the HTTP server

Design document

Before writing code for this assignment, as with every other assignment, you must write up a design document. Your design document must be called `DESIGN.pdf`, and must be in PDF. You can easily convert other document formats, including plain text, to PDF. Scanned-in design documents are fine, as long as they're legible and they reflect the code you *actually* wrote.

Your design should describe the design of your code in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, non-trivial algorithms and formulas, and a description of each function with its purpose, inputs, outputs, and assumptions it makes about inputs or outputs.

Assignment 4 involves implementing a second key-value store that maps one generic string (a name) to another. You may reuse your key-value store code from Assignment 3, though you'll likely find it useful to make some small changes to efficiently do this mapping.

Write your design document *before* you start writing code. It'll make writing code a lot easier. Also, if you want help with your code, the first thing we're going to ask for is your design document. We're happy to help you with the design, but we can't debug code without a design any more than you can.

You **must** commit your design document *before* you commit the code specified by the design document. You're welcome to do the design in pieces (*e.g.*, making the small changes first), as long as you don't write code for the parts that aren't well-specified. We **expect** you to commit multiple versions of the design document; your commit should specify *why* you changed the design document if you do this (*e.g.*, "original approach had flaw X", "detailed design for module Y", etc.). **If you commit code before it's designed, or you commit your design a few minutes before the working code that the design describes, you will lose points.** We want you to get in the habit of designing components before you build them.

Program functionality

You may not use standard libraries for HTTP or the key-value store; you have to implement them yourself. You may use standard networking (and file system) system calls, but not any `FILE *` or `iostream` calls except for printing to the screen (*e.g.*, error messages). You may use open-source hash functions (such as `cityhash`), but you must cite your sources. Your code may be either C or C++, but all source files must have a `.cpp` suffix and be compiled by `clang++`.

Your code must handle all requirements from Assignment 3. In addition, your server now needs to handle a PATCH command that maps one string to another using a persistent key-value store. The total length of the two strings (key and value) will be no longer than 128 bytes, including the NULL terminators on the two strings. If you encounter a mapping whose total length would be longer, your server may return an error, or it may choose to handle the mapping properly, but it must do so *consistently*. If you return SUCCESS, the server muste later recognize the alias.

If you had difficulty with Assignment 3, focus on the code for Assignment 4 and include notes in your `README.md` file.

Removing leading slash

Your server should remove a leading slash from *all* names that it processes, including httpnames. For example, the name `/abcdef0123456789` must be processed as `abcdef0123456789`. However, `a/b` would still be processed as `a/b`, since the slash isn't the first character in the name.

Do this part first; it should be a quick change if you've properly designed your server.

Name mapping

Your HTTP server must support the PATCH command to update an existing object. The PATCH command is sent by the client to the server, and the body of the command specifies the "patch" to make to an *existing* object. For now, the only thing the PATCH body can include is a single line that looks like this:

```
ALIAS existing_name new_name\r\n
```

This will make *new_name* an alias for *existing_name*. *existing_name* may be an httpname (40 hex characters) or an already-existing alias. If *existing_name* doesn't already exist on the server, it should return a 404 error (not found).

When the client does a GET, the server follows aliases until it either gets an actual object, or until a name doesn't exist. Yes, in the default implementation names would always exist, but a later implementation might allow for names to be deleted, so you should take this into account.

Names are resolved at GET time, *not when the alias is created*. When the alias is created, the server only makes sure that the destination name exists (late binding of the full alias path).

Key-value store

You're going to need another key-value store for this assignment, but you can reuse some (or much) of the KV store from your previous assignment.

Since your key and value are both relatively short (under 128 characters **combined**, including the NULL terminators), it'll be sufficient to use something like the index from the previous assignment. Each entry in the index can be 128 characters, and there's no need for a block to be associated with the entry. Reuse as much code from your last assignment as you can, but include the design in your design document.

The mapping file is a separate file, similar to the KV store file you created for Assignment 3. Its name is specified by the `-m` option, similar to how the file was specified for Assignment 3 using the `-f` option. If no mapping file is specified, the server should exit with an error. If the mapping file doesn't exist, it should be created as an empty file with no aliases in it.

The mapping file must be able to handle 10,000 aliases.

You may want to write a small program to load data into this file, or to dump data out. If you do so, you may turn in the code for these as well, but **this is optional, not required**. However, you'll find that it's a very good idea to have standalone code that you can use to examine your key-value store file.

As with Assignment 3, your design and implementation for Assignment 4 must be thread-safe, but you need not worry about consistency—that is, two threads updating the same object or name should not cause race-conditions in the organization of the KVS and in the hash table, but may cause the object or name to contain an unpredictable result.

README and Writeup

Your repository must also include a README file (`README.md`) writeup (`WRITEUP.pdf`). The README may be in either plain text or have MarkDown annotations for things like bold, italics, and section headers. **The file must always be called `README.md`**; plain text will look "normal" if considered as a MarkDown document. You can find more information about Markdown at `https://www.markdownguide.org`.

The `README.md` file should be short, and contain any instructions necessary for running your code. You should also list limitations or issues in `README.md`, telling a user if there are any known issues with your code.

Your `WRITEUP.pdf` is where you'll describe the testing you did on your program and answer any short questions the assignment might ask. The testing can be unit testing (testing of individual functions or smaller pieces of the program) or whole-system testing, which involves running your code in particular scenarios.

For Assignment 4, please answer the following questions:

- Explain the difference between fully resolving a name (to an `httpname`) when the name is created and the approach that you're taking for this assignment. Give an example of when it might be useful.
- Was it easier to modify your existing KV store code from Assignment 3 as compared to getting it to work the first time?
- What did you learn about system design from this class? In particular, describe how each of the basic techniques (abstraction, layering, hierarchy, and modularity) helped you by simplifying your design, making it more efficient, or making it easier to design.

Submitting your assignment

All of your files for Assignment 4 must be in the `asgn4` directory in your `git` repository. When you push your repository to GITLAB@UCSC, the server will run a program to check the following:

- There are no "bad" files in the `asgn4` directory (*i.e.*, object files).
- Your assignment builds in `asgn4` using `make` to produce `httpserver`.
- All required files (`DESIGN.pdf`, `README.md`, `WRITEUP.pdf`) are present in `asgn4`.

If the repository meets these minimum requirements for Assignment 4, there will be a green check next to your commit ID in the GITLAB@UCSC Web GUI. If it doesn't, there will be a red X. **It's OK to commit and push a repository that doesn't meet minimum requirements for grading.** However, we will only *grade* a commit that meets these minimum requirements.

Note that the *minimum* requirements say nothing about correct functionality—the green check only means that the system successfully ran `make` and that all of the required documents were present, with the correct names.

If you commit any file containing the actual hash table data at any time, you will lose 40 points on the assignment. These are likely to be large files, and should never be committed!

You must submit the commit ID you want us to grade via Google Form, linked to the assignment page on Canvas. This must be done before the assignment deadline.

Hints

- Start early on the design. This program builds on Assignment 2. If you didn't get Assignment 2 to work, please see Prof. Miller or Daniel ASAP for help getting it to work.
- Reuse your code from Assignment 3. No need to cite this; we expect you to do so. But you must do your work in the `asgn4` directory.
- Reuse as much of the KV store code as you can from Assignment 3 for Assignment 4. You won't need the block storage part, but can modify the index to be all you need for the mapping store (mapping one name to another).
- Go to section March 5–6 for additional help with the program. This is especially the case if you don't understand something in this assignment!
- You'll need to use (at least) the system calls from Assignment 3. You likely won't need any more system calls.
- Work through the design for the mapping store *carefully* before you start to implement it. Make sure you've answered any questions you might have in advance.
- Make sure you handle the case where it might take 3 or more indirections to resolve a name. Really, you should keep going until you get an `httpname`.
- You may assume that any name of exactly 40 hex characters is an `httpname` and should be looked up as one. Any other name you encounter in the name resolution process should be looked up in the name mapping store.
- You don't need to handle deleting name aliases, but you *do* need to handle overwriting an alias with a different mapping. For example, if "myfile" points to "test", you might get a request to map "myfile" to "bar". This should succeed.

Grading

As with all of the assignments in this class, we will be grading you on *all* of the material you turn in, with the ***approximate*** distribution of points as follows: design document (35%); coding practices (20%); functionality (35%); writeup (10%).

If you submit a commit ID without a green checkmark next to it or modify `.gitlab-ci.yml` in any way, your maximum grade is 5%. Make sure you submit a commit ID with a green checkmark.

Extra Credit

Extra credit will be announced on March 8th.

