Assignment 1
CMPE 105: Principles of Computer System Design, Winter 2019

Due: February 1 at 5:00PM

Goals

The goals for Assignment 1 are to implement a simple HTTP server and client. The server is single-threaded, as is the client. The server will use simple GET and PUT commands to read and write (respectively) "files" named by 40-character hexadecimal names. The server will persistently store files in a directory on the server, so it can be restarted or otherwise run on a directory that already has files.

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

Programming assignment: HTTP client and 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).

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.

You should have a *single* design document that covers both your server and client, each of which should have its own section. You're encouraged to have them share code, if that makes sense for your design. If you do so, you should have a third section that covers "common" routines that are used by both client and server.

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.*, overall design and detailed design of HTTP handling functions but not of HTTP client and server), 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; you have to implement this yourself. You may use standard networking (and file system) system calls, but not any `FILE *` calls except for printing to the screen (*e.g.*, error messages).

Your code may be either C or C++, but all source files must have a `.cpp` suffix and be compiled by `clang++`.

HTTP protocol

The HTTP protocol is used by clients and servers for a significant amount of web communication. It's designed to be simple, easy to parse (in code), and easy to read (by a human). Both your client and server will need to send and receive files via `http`, so the best approach will likely allow them to share code for managing connections, copying data between file descriptors (which you should be well-acquainted with after asgn0!) as well as for the limited amount of parsing that `http` requires.

The `http` protocol that you need to implement is very simple. The client sends a request to the server that either asks to send a file from client to server (PUT) or fetch a file from server to client (GET).

Both types of requests consist of a *single line* specifying the action, with PUT including the data immediately following the single line. The single line is "`ACTION RESOURCE-NAME HTTP-VERSION`", and looks like this (characters preceded by \ are control characters):

```
PUT 0123456789abcdeabcde0123456789abcdeabcde HTTP/1.1\r\n
```

The name that the server will bind to the data is `0123456789abcdeabcde0123456789abcdeabcde`. All valid resources names in this assignment will be 40 characters. For a PUT, there is a second `\r\n` followed immediately by the data; there are no restrictions on the form of the data. For a GET, there is no additional data in the request; the server places the data into the response.

The server must respond to a PUT or GET with a "response", which is a response header optionally followed by data. An example response header looks like this:

```
HTTP/1.1 200 OK\r\n
```

The `200` is a status code—200 means "OK". The `OK` message is an informational description of the code. For example, the `404` status code could say "File not found". The server must fill in the appropriate status code and message. This line is followed by `\r\n` (an additional blank line) and then the body for a GET response; the body contains the file that the client is requesting. For a PUT response, the single line is all that the response contains.

You can find a list of HTTP status codes at:
`https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`.

The only status codes you'll *need* to implement are 200 (OK), 201 (Created), 400 (Bad Request), 403 (Forbidden), 404 (Not Found), and 500 (Internal Server Error). You may use additional status codes if you like, but these are the only required ones. Look at the link above to determine when to use each one.

A simplifying assumption you may make *without penalty* is that a *valid* request header is always the same length (see the PUT line above) because the resource name, for this assignment, is 40 characters. Note, however, that both your client and server will need to be able to handle malformed requests and responses (error message, not crashing). Additionally, you may assume that all *response* header lines (*e.g.* the line above starting with HTTP/1.1) will fit within 4KB, including the blank line at the end of the GET response header. Note that that does not mean the entire response will fit within 4KB! The file data that is part of a GET request may be longer!

HTTP client

Your client binary must be called `httpclient`.

The first argument to the client will be the address of the HTTP server to contact, specified as `address:`*`port`* (*e.g.*: `localhost:80`). `address` may be specified as a hostname or an IP address; your software will handle either one. The port number may be omitted, in which case the client will assume the standard HTTP port, port 80.

Following the server address, your HTTP client will take one command line argument for each request that will be sent to the server. Each argument is of one of two forms (send and receive), where *filename* is the path to the data, and *httpname* is the name that will be provided (via HTTP) by the client to the server:

- **Send a file from client to server:** argument is of the form `s:filename:httpname`.
- **Receive a file from server to client:** argument is of the form `r:httpname:filename`.

If no files are specified on the command line, your client should simply exit, with no error message—you've done all that the user asked. A single invocation can have both send and receive requests, in any order. Individual files may be up to 2 MiB in length, but you may have no buffer larger than 64 KiB.

`httpclient` should process its requests in argument order, skipping any that don't complete because of an error. Your client should check for errors as much as possible, including files that aren't found, errors connecting to the server, naming errors (the HTTP names we're using in this assignment must be 40 hex characters) and problems with transmission. If an error occurs, print a useful message to `stderr` and move on to the next request.

HTTP server

Your server binary must be called `httpserver`.

Your HTTP server is a single-threaded server that will listen on a specified port and respond to HTTP PUT and GET requests on that port. The address to listen to and the port number are specified on the command line in the same way as for the HTTP client. (Yes, it *is* necessary to specify the server address, since your computer has multiple Internet addresses, including `localhost`.) Your server will use the directory in which it's run to store files that are PUT, and load files for which a GET request is made.

Extra Credit

As described above, your server should be able to interoperate with `curl`, handling GET requests. However, a browser will not render any data if you try to connect to your server. That's because they expect to see an additional response line before the data: Content-Length. More details can be found here, but the idea is that when your server responds to a GET, it should add a line directly after the response header but before the blank line, as so:

```
Content-Length: 1024\r\n
```

This line should tell the client what the length of the data is, and will make Firefox and Chrome display the data.

For extra credit, update your server to provide this information, and make sure your client can handle the additional response line. Create a simple HTML file that contains whatever you like along with the text "Assignment 1 Extra Credit (your cruzid)", and screenshot your browser displaying the file (*including the address bar so we can see you're connecting to localhost*). Place the screenshot in your write-up.

If you do this, then congrats, you have a working HTTP server that a browser will talk to! Probably don't put it on the open internet, but feel free to brag to your friends. Building your own HTTP server is pretty cool! :)

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 1, please answer the following question:

- *Briefly* discuss (but do not implement) the changes you'd need to make to allow the client to send and receive from `stdin` and `stdout`, respectively.
- What would happen in your implementation if, during a transfer, the connection was closed, ending the communication early? This extra concern was not present in your implementation of mycat. Why not? Hint: this is an example of complexity being added by an extension of requirements (in this case, data transfer over a network). You do not need to test or implement this, just discuss it.

- Work with another student in the class and use *your* client to talk to *their* server, and *their* client to talk to *your* server. **You should each run the client and server on your own VM.** Report on your experiences, including the name and CruzID of the person you worked with. You may do this with more than one student if you like. **REMEMBER:** while we want you to try your server and their client (and *vice versa*), **you may not show your *design or code* to another student**. Just try client-server interoperation, and let the protocol do the talking.

Submitting your assignment

All of your files for Assignment 1 must be in the `asgn1` 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 `asgn1` directory (*i.e.*, object files).
- Your assignment builds in `asgn1` using `make` to produce `httpclient` and `httpserver`.
- All required files (`DESIGN.pdf`, `README.md`, `WRITEUP.pdf`) are present in `asgn1`.

If the repository meets these minimum requirements for Assignment 1, 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.

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 is a more difficult program than mycat!
- Go to section on January 22 or 23 for details on the code you need to set up an HTTP server connection. While you'll need this code for your server (obviously), you can "include" it in your design with a simple line that says "set up the HTTP server connection at address X and port Y".
- You'll need to use (at least) the system calls `socket`, `bind`, `listen`, `accept`, `connect`, `send`, `recv`, `open`, `read`, `write`, `close`. The last four calls should be familiar from Assignment 0, and `send` and `recv` are very similar to `write` and `recv`, respectively. You should read the `man` pages or other documentation for the other system calls. Don't worry about the complexity of opening a socket; we'll discuss in section (see above).
- Test your server and client separately. You may use an existing Web server to test your client, and an existing Web client (`curl`, `wget`) to test your server.
- Make a simple HTTP library that both your client and server use so fixes happen in a single place.
- Aggressively check for and report errors. Both the client and server may skip transfers if errors occur.

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.