Assignment 1: HTTP Server
CSE 130: Principles of Computer Systems Design

Due: April 27, 2022 at 12:01 AM

Goals This assignment will provide you with experience building a system that uses strong modularity. Additionally, this will provide you with an opportunity to apply the things we will learn about client/server systems in class to the real world. This assignment is a fair bit more complex than Assignment 0; you would do well to get a head start on the task.

Overview You will be building an HTTP server for this assignment. Your server should execute "forever" without crashing (i.e., it should run until a user types `CTRL-C` on the terminal). Your server should process incoming HTTP commands from clients; we provide starter code to help you accept incoming commands.

The code that you submit must be in your repository on `git.ucsc.edu`. In particular, your assignment must build your HTTP server, called `httpserver`, when we execute the command `make` from the directory `asgn1` in your repository. Your repository must include a `README.md` file that includes information about the interesting design decisions and outlines the high-level functions, data-structures, and modules that you used in your code. The `README.md` should also include how your design implements any "vague" requirements listed in this document.

You must submit a 40 character commit ID hash on Canvas in order for us to identify which commit you want us to grade. We will grade the last hash that you submit to the assignment on Canvas and will use the timestamp of your last upload to determine grace days and bonus points. For example, if you post a commit hash 36 hours after the deadline, we will subtract 2 grace days from your total.

Your server must implement the functionality explained below subject to the limitations below. You'll need to handle bad or poorly formatted requests in accordance with the way they are handled by the reference implementation.

Functionality

Your server will create, listen, and accept connections on a *socket* that is listening on an *port*. A socket is an abstraction provided by the operating system and libc that represents a connection with an external entity. For the purposes of this project, you should think of a socket as equivalent to a file descriptor: It is an integer that represents a sequence of characters. You can receive bytes in the order that they were written by the other side (i.e., the client) by calling `read` and using the socket as the file descriptor. You can write bytes to the other side by calling `write` using the socket as the file descriptor; the bytes that you write will be received by the other side (i.e., the client) in the order in which you write them.

Your server should take a single command-line argument, an `int`, to specify the port that your server will listen on. Namely, you should start your server using the following command:

```
./httpserver <port>
```

We have provided starter code alongside this assignment that simplifies working with the socket and managing the command-line arguments[1]. The starter code creates a socket that listens on the `localhost` interface with the `port` provided on the command line. When you test your `httpserver`, you will specify `localhost` and the `port` that you provided to `httpserver` to establish a connection; the Hints section below gives some tips for this process.

The starter code manages the socket such that you can treat network programming as simply as interacting with file descriptor. Your task is to implement the `handle_connection` function in the starter code,

[1] We describe some details of the starter code at the end of this document

which takes a socket as input and should process client HTTP requests. In particular, the `httpserver` should support three types of HTTP operations: `GET`, `PUT`, and `APPEND`. Below, we first describe the HTTP Protocol, and then describe what each of these operations should do for a valid HTTP request.

HTTP Protocol

The HTTP 1.1 protocol is described in detail in Request For Comments: 2616 (abbreviated RFC 2616)[2], which is a standards document produced by the Network Working Group. RFC 2616 describes all of the functionality that an server and client needs to support for the HTTP 1.1 protocol. We have simplified the features that you must support for this project and have created a new operation, `APPEND`, for you to implement.

Requests

Regardless of the type of operation, valid HTTP requests will follow the grammar below. To make your processing easier, you may assume that the total length of the `request-line` and all `header-fields` is less than or equal to 2048 for any valid request:

```
Request =   Request-Line\r\n       ; [Required]
            (Header-Field\r\n)*    ; [Optional, repeated]
            \r\n
            Message-Body           ; [Optional]
```

- Request-Line. Every valid request must include exactly one request-line. The request-line specifies the method to be performed, the object upon which to perform it, and the HTTP version. A client denotes the end of a request-line using the sequence `\r\n`. The format of request-line is as follows:

  ```
  Method URI Version
  ```

  In detail, `Method`, `URI`, and `Version` are as follows:

  - `Method` Valid requests will include a Method, a case-sensitive sequence of characters within the range [a-z, A-Z] (i.e., "HeLlO" and "HELLO" are both valid, but different, methods). All valid requests in this assignment will use a method that uses eight (8) characters or fewer. Your `httpserver` will need to implement the semantics of `GET`, `PUT`, and `APPEND`, as explained below.
  - `URI` Valid requests will include a URI, a case-sensitive sequence of characters starting with the character '/' and including additional characters within the range [a-z], [A-z], [0-9] and including the characters '.' and '_', for a total of 64 characters in the alphabet of valid URIs. Each URI, `/path`, matches to the file `path` within the directory in which the `httpserver` is running. For example, the URI `/foo.txt` matches the file `foo.txt` from the folder in which the `httpserver` is running. All valid URIs in this assignment will contain a path with 19 characters or fewer.
  - `Version` Valid requests will include an HTTP version, of the form `HTTP/#.#` (where each # is a single digit number). Your `httpserver` only needs to support version 1.1, so `HTTP/1.1` is the only version string upon which your server needs to perform `GET`, `PUT`, and `APPEND` operations.

- Header-Field. Every valid request will include zero (0) or more `header-field`s after `request-line`. A `header-field` is key-value pair expressed in the form:

  ```
  key: value
  ```

  The `key` and `value` fields are delimited by the first instance of a ':' character. Valid requests' keys will be in ascii and not contain whitespace. Valid requests will separate each `header-field` using the sequence `\r\n`, and will terminate the list of `header-field`s with a blank header terminating in `\r\n`. (Essentially, the list of `header-field`s end with `\r\n\r\n`).

[2] I believe that there are more current RFCs for HTTP1.1, but we'll be using 2616 for this assignment.

- Message-Body. Valid `PUT` and `APPEND` requests must include a message-body; valid `GET` requests will not include a message-body. Valid requests that include a `Message-Body` will also include a header, `Content-Length`, whose value will indicate the number of bytes in the `Message-Body`.

As a summary, here is an example of a valid `PUT` request to the URI, `foo.txt`:

```
PUT /foo.txt HTTP/1.1\r\nContent-Length: 12\r\n\r\nHello world!
```

Responses

Your `httpserver` must produce a response for each request. Your response must follow the grammar:

```
Request =  Status-Line\r\n        ; [Required]
           (Header-Field\r\n)*    ; [Optional, repeated]
           \r\n
           Message-Body           ; [Optional]
```

- Status-Line The status line indicates the type of response to the request. It consists of three fields:

  HTTP-Version Status-Code Status-Phrase

  `httpserver` must always produce the HTTP-Version string, `HTTP/1.1`, regardless of the `HTTP-Version` provided in the request. A response with a `Status-Code` in the 200s indicates a successful response, in the 400s indicates an erroneous response, and in the 500s indicates an internal server error. The list of `status-codes` that `httpserver` needs to produce, and their associated `status-phrase`, are listed below. When returning a `status-code` listed below, the `status-phrase` field should have the corresponding value. We provide guidance on when to return each code in the sections on methods and errors.

Status-Code	Status-Phrase	Message-Body	Usage
200	OK	OK\n	When a method is Successful
201	Created	Created\n	When a URI's file is created
400	Bad Request	Bad Request\n	When a request is ill-formatted
403	Forbidden	Forbidden\n	When the server cannot access the URI's file
404	Not Found	Not Found\n	When the URI's file does not
500	Internal Server Error	Internal Server Error\n	When an unexpected issue prevents processing
501	Not Implemented	Not Implemented\n	When a request includes an unimplemented Method

- Header-Field. The `status-line` should be followed by zero (0) or more `header-field`s. A response's `header-field`s have the same format as the request header fields, namely:

  `key: value`

  Responses that include a `message-body` must include a `Content-Length` header field. `httpserver` must separate each `header-field` using the sequence \r\n and terminate the list of `header-fields` with a blank header terminating in \r\n. (Essentially, the list of `header-field`s ends with \r\n\r\n).

- Message-Body. `httpserver` must produce a response with a `message-body` for each `GET` request. A `message-body` in a response must always be accompanied by a `Content-Length` header that indicates the number of bytes in the `Message-Body`.

Methods

You must implement three HTTP methods, `GET`, `PUT`, and `APPEND`:

- `GET` A `GET` request indicates that the client would like to receive the content of the file identified by the `URI`. For each valid `GET` request, `httpserver` must produce a response with a `status-code` of `200`, a `message-body` that includes the current state of the file pointed to by `URI`, and a `Content-Length` that indicates the number of bytes in the file.
- `PUT` A `PUT` request indicates that the client would like to update/replace the content of the file identified by the `URI`. If a valid `PUT` request's URI points to a file that does not yet exist, `httpserver` must create the file, set the file's contents equal to the `message-body` in the request, and produce a response with a `status-code` of `201`. If a valid `PUT` request's URI points to a file that does already exist, `httpserver` must replace the file's contents with the `message-body` in the request and produce a response with a `status-code` of `200`.
- `APPEND` An `APPEND` request indicates that the client would like to append content to the end of the file identified by the `URI`. For each valid `APPEND` request, `httpserver` must add the content of the `message-body` to the end of the file pointed to by the URI and produce a response with a `status-code` of `200`.

Additional Functionality

In addition to supporting the methods listed above, your project must do the following:

- `httpserver` must work on binary `message-body`s.
- `httpserver` must produce error messages and return codes that are identical to those of the reference implementation; see the section below on errors.
- `httpserver` should not have any memory leaks.
- `httpserver` must be reasonably efficient.
- Your code should be formatted according to the clang-format provided in your repository and it should compile using `clang` with the `-Wall -Werror -Wextra -pedantic` compiler flags.

Limitations

You must write `httpserver` using the C programming language. Your program cannot call the following functions from the C `stdio.h` library: `fwrite`, `fread`, `fopen`, `fclose`, variants of `put` (e.g., `fptuc`, `putc`, `putc_unlocked`, `putchar`, `putchar_unlocked`, and `putw`), and `get` (e.g., `fgetc`, `fget`, `getc`,`getc_unlocked`, `getchar`, `getchar_unlocked`, and `getw`). You cannot use functions, like `system(3)`, that allow you to execute external programs. **If your submission does not meet these minimum requirements, then the maximum score that you can get is 5%.**

Errors

`httpserver` must never crash and should not produce output to standard out or standard error (the starter code validates and produces errors for the only input to `httpserver`). Your `httpserver` will only need to produce responses that contain a `status-code` listed in the table above. You can consult the reference implementation to identify the precise scenarios in which you should produce each `status-code`. The reference is located (in binary format, of course!) on Git@UCSC. Part of the assignment is enumerating as many errors as you can think of and figuring out when to produce the error messages!

Hints

Here are some hints to help you get going:

- You will likely need to lookup how some system calls (e.g., `read`) and library functions (e.g., `warn`) work. You can always Google them, but you might also find the man pages useful (e.g., try typing `man 2 read` on a terminal).
- There are a few ways to test `httpserver`. Below, we assume that you started `httpserver` on port 1234 by executing the command `./httpserver 1234`. We also assume that you are using your client on the same machine upon which the server is currently executing:
  - You can use an HTTP Client, such as a web browser. We recommend testing with `curl`, a command-line HTTP client. `curl` can produce both `GET` and `PUT` commands. For example, to execute a `GET` of the file `foo.txt` on `httpserver` and place the output into the file `download.txt`, you execute:

    ```
    curl http://localhost:1234/foo.txt -o download.txt
    ```

    `curl` can execute `PUT` and `GET` commands; use `help curl` or `man curl` on a terminal to learn more.
  - You can also use `nc` (often called "netcat"). To connect to your server, execute `nc localhost 1234`. Then, you can type in the text that you wish to send to your server. You can also automate this approach by piping data to `nc`. For example, to send the `PUT` command listed above to your server, execute:

    ```
    printf "PUT /foo.txt HTTP/1.1\r\nContent-Length: 12\r\n\r\nHello World!"
    |nc localhost 1234
    ```
- If you try to start your server immediately after killing a previous instance of it, you will likely see the following error:

  ```
  httpserver: bind error: Address already in use
  ```

  In this case, just restart the server with a different port number. The issue is that the operating system must ensure unique ports are used across the entire system; it often waits to gracefully close ports even after the process that was using them terminates.

Dazzle Points

In addition to the functionality listed above, our reference implementation implements a number of extensions. These extensions fall into two categories (1) additional functionality and (2) supporting the robustness principle. You can earn Dazzle Points by implementing the optional features implemented in the reference implementation.

To make this more fun (or, competitive), we will include a Dazzle Points Scoreboard on Dr. Q's website at this link. To be included in the scoreboard, include a file `scoreboard.txt` in the `asgn1` folder of your repository. The `scoreboard.txt` should include a single word, `name`, which we will use to display your Dazzle Points[3]

Grading

Your submission will be graded using the following high-level rubric:

- Functionality tests: 70%
- README design doc: 15%
- Coding style: 15%

Starter Code

Your starter code does three basic things for you:

- Your starter code parses the command line argument provided by the client and ensures that it is in the correct format.

[3]Keep in mind, while this is anonymous to your peers, the name you choose to display is not anonymous to us. Trying to get Dr. Q to display profane or inappropriate things on his website is not a good strategy for success in this class.

- Your starter code creates a socket, binds that socket to the local interface, and then listens for requests on the socket. For each request, the starter code calls `handle_connection`.
- Your starter code includes a signal handler that ignores the `SIGPIPE` signal. This handler makes it so that socket failures will result linux setting `errno` to `EPIPE` rather than throwing a signal.