HW 4: **HTTP Server**

CS 162

Due: 9:00 PM November 9, 2019

Contents

1   Introduction

The Hypertext Transport Protocol (HTTP) is the most commonly used application protocol on the Internet today. Like many network protocols, HTTP uses a client-server model. An HTTP client opens a network connection to an HTTP server and sends an HTTP request message. Then, the server replies with an HTTP response message, which usually contains some resource (file, text, binary data) that was requested by the client.

In this assignment, you will implement an HTTP server that handles HTTP GET requests. You will provide functionality through the use of HTTP response headers, add support for HTTP error codes, create directory listings with HTML, and create a HTTP proxy. The request and response headers must comply with the HTTP 1.0 protocol found here[1].

1.1   Getting Started

Log in to your VM and grab the skeleton code from the staff repository:

```
$ cd ~/code/personal
$ git pull staff master
$ cd hw4
```

Run `make` to build the code. Four binaries should be created: `httpserver_basic`, `httpserver_process`, `httpserver_thread`, and `httpserver_pool`.

1.2   Setup Details

The CS 162 Vagrant VM is set up with a special host-only network that will allow your host computer (e.g. your laptop) to connect directly to your VM. The IP address of your VM is `192.168.162.162`.

You should be able to run `ping 192.168.162.162` from your host computer (e.g. your laptop) and receive ping replies from the VM. If you are unable to ping the VM, you can try setting up port forwarding in Vagrant instead (more information here[2]).

---

[1]http://www.w3.org/Protocols/HTTP/1.0/spec.html
[2]https://docs.vagrantup.com/v2/networking/forwarded_ports.html

2 Background

2.1 Structure of HTTP Request

The format of a HTTP request message is:

- an HTTP request line (containing a method, a query string, and the HTTP protocol version)

- zero or more HTTP header lines

- a blank line (i.e. a CRLF by itself)

The line ending used in HTTP requests is CRLF, which is represented as `\r\n` in C.

Below is an example HTTP request message sent by the Google Chrome browser to a HTTP web server running on localhost (127.0.0.1) on port 8000 (the CRLF's are written out using their escape sequences):

```
GET /hello.html HTTP/1.0\r\n
Host: 127.0.0.1:8000\r\n
Connection: keep-alive\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
User-Agent: Chrome/45.0.2454.93\r\n
Accept-Encoding: gzip,deflate,sdch\r\n
Accept-Language: en-US,en;q=0.8\r\n
\r\n
```

Header lines provide information about the request[3]. Here are some HTTP request header types:

- **Host**: contains the hostname part of the URL of the HTTP request (e.g. `inst.eecs.berkeley.edu` or `127.0.0.1:8000`)

- **User-Agent**: identifies the HTTP client program, takes the form "Program-name/x.xx", where x.xx is the version of the program. In the above example, the Google Chrome browser sets User-Agent as `Chrome/45.0.2454.93`.

---

[3] For a deeper understanding, open the web developer view on your web browser and look at the headers sent when you request any webpage

2.2   Structure of HTTP Response

The format of a HTTP response message is:

- an HTTP response status line (containing the HTTP protocol version, the status code, and a description of the status code)

- zero or more HTTP header lines

- a blank line (i.e. a CRLF by itself)

- the content requested by the HTTP request

The line ending used in HTTP requests is CRLF, which is represented as `\r\n` in C.

Here is a example HTTP response with a status code of 200 and an HTML file attached to the response (the CRLF's are written out using their escape sequences):

```
HTTP/1.0 200 OK\r\n
Content-Type: text/html\r\n
Content-Length: 128\r\n
\r\n
<html>\n
<body>\n
<h1>Hello World</h1>\n
<p>\n
Let's see if this works\n
</p>\n
</body>\n
</html>\n
```

Typical status lines might be `HTTP/1.0 200 OK` (as in our example above), `HTTP/1.0 404 Not Found`, etc.

The status code is a three-digit integer, and the first digit identifies the general category of response:

- 1xx indicates an informational message only

- 2xx indicates success

- 3xx redirects the client to another URL

- 4xx indicates an error in the client

- 5xx indicates an error in the server

Header lines provide information about the response. Here are some HTTP response header types:

- **Content-Type**: the MIME type of the data attached to the response, such as `text/html` or `text/plain`

- **Content-Length**: the number of bytes in the body of the response

3   Your Assignment

From a network standpoint, your basic HTTP web server should implement the following:

1. Create a listening socket and bind it to a port
2. Wait a client to connect to the port
3. Accept the client and obtain a new connection socket
4. Read in and parse the HTTP request
5. Serve a file from the local file system, or yield a 404 Not Found

The skeleton code already implements steps 1-4 for you.

3.1   Usage

Running `make` in your terminal will generate 4 executables:

- `httpserver_basic` — server process sends the HTTP response

- `httpserver_process` — server process forks a child process that sends the HTTP response

- `httpserver_thread` — server process creates a thread that sends the HTTP response

- `httpserver_pool` — server process creates a work request that is served by a pool of threads

Here is the usage string for the executables. The argument parsing step has been implemented for you:

```
$ ./httpserver_basic --help
Usage: ./httpserver_basic --files files/ [--port 8000 --concurrency 1]

$ ./httpserver_process --help
Usage: ./httpserver_process --files files/ [--port 8000 --concurrency 1]

$ ./httpserver_thread --help
Usage: ./httpserver_thread --files files/ [--port 8000 --concurrency 1]

$ ./httpserver_pool --help
Usage: ./httpserver_pool --files files/ [--port 8000 --concurrency 1]
```

The available options are:

- `--files` — Selects a directory from which to serve files. You should be serving files from the hw4/ folder (e.g. if you are currently cd'ed into the hw4/ folder, you should just use "`--files files/`".

- `--port` — Selects which port the http server listens on for incoming connections. Default port is 8000.

- `--concurrency` — Indicates the number of threads in your thread pool that are able to concurrently serve client requests. This argument is unused by `httpserver_basic`, `httpserver_process`, and `httpserver_thread`. Default value is 1.

If you want to use a port number between 0 and 1023, you will need to run your http server as root. These ports are the "reserved" ports, and they can only be bound by the root user. You can do this by running "`sudo ./httpserver_basic --files files/`".

3.2   Accessing the HTTP server

Check that your HTTP server works by sending HTTP requests with the `curl` program, which is installed
on your VM. An example of how to use curl is:

```
$ curl -v http://192.168.162.162:8000/
```

You can also open a connection to your HTTP server directly over a network socket using netcat
(nc), and type out your HTTP request (or pipe it from a file):

```
$ nc -v 192.168.162.162 8000
Connection to 192.168.162.162 8000 port [tcp/*] succeeded!
(Now, type out your HTTP request here.)
```

3.3   Common error messages

3.3.1   Failed to bind on socket: Address already in use

This means you have an httpserver running in the background. This can happen if your code leaks
processes that hold on to their sockets, or if you disconnected from your VM and never shut down your
httpserver. You can fix this by running "`pkill -9 httpserver`". If that doesn't work, you can specify a
different port by running "`httpserver_basic --files files/ --port 8001`", or you can reboot your
VM with "`vagrant reload`".

3.3.2   Failed to bind on socket: Permission denied

If you use a port number that is less than 1024, you may receive this error. Only the root user can
use the "well-known" ports (numbers 1 to 1023), so you should choose a higher port number (1024 to
65535).

3.4 Your Assignment

1. Implement `handle_files_request(int socket_fd)`, `serve_file`, and `serve_directory` to handle HTTP GET requests for files. This function takes in the connection socket fd obtained in step 3 of the outline above. Your handler should:

   - Use the value of the `--files` command line argument, which contains the path where the files are. (This is stored in the global variable `char *server_files_directory`)

   - If the HTTP request's path corresponds to a file, respond with a 200 OK and the full contents of the file. (e.g. if `GET /index.html` is requested, and a file named `index.html` exists in the files directory) You should also be able to handle requests to files in subdirectories of the files directory (e.g. `GET /images/hero.jpg`)

     Hints:

     – Look in `libhttp.h` for a bunch of useful helper functions! An example of their usage is provided in the skeleton code and some documentation can be found in the appendix.

     – Make sure you set the correct `Content-Type` HTTP header. A helper function in `libhttp.h` will return the MIME type of a file. (This is really the only header you need to implement to get images/documents to display properly.)

     – Also make sure you set the correct `Content-Length` HTTP header. The value of this header should be the size of the HTTP response body, measured in bytes. For example, `Content-Length: 7810`.

     – HTTP request paths **always begin with a /**, even if you are requesting the home page (e.g. `http://inst.eecs.berkeley.edu/` would have a request path of /).

   - If the HTTP request's path corresponds to a directory and the directory contains an `index.html` file, respond with a 200 OK and the full contents of the index.html file. (You may not assume that directory requests will have a trailing slash in the query string.)

     Hints:

     – To tell the difference between files and directories, you may find the **stat()** function and the **S_ISDIR** or **S_ISREG** macros useful

     – You do not need to handle file system objects other than files and directories (e.g. you do not need to handle symbolic links, pipes, special files)

     – Make helper functions to re-use similar code when you can. It will make your code easier to debug!

   - If the request corresponds to a directory and the directory does not contain an `index.html` file, respond with an HTML page containing links to all of the immediate children of the directory (similar to `ls -1`), **as well as a link to the parent directory**. (A link to the parent directory looks like `<a href="../">Parent directory</a>`)

     Hints:

     – To list the contents of a directory, good functions to use are `opendir()` and `readdir()`

     – Links in HTTP can use relative paths or absolute paths. It is just like how `cd usr/` and `cd /usr/` do two entirely different things.

     – You don't need to worry about extra slashes in your links (e.g. `//files///a.jpg` is perfectly fine). Both the file system and your web browser are tolerant of it.

     – Don't forget to set the `Content-Type` header.

   - Otherwise, return a 404 Not Found response (the HTTP body is optional). There are many things that can go wrong during an HTTP request, but we only expect you to support the 404 Not Found error message for a non-existent file.

- You only need to handle one HTTP request/response per connection when serving files. You do not need to implement connection keep-alive or pipelining for this section.

- After correctly implementing this task, `httpserver_basic` gives you a fully functional HTTP web server. Take a look at "See my files" and add in your own files to the `files` directory if you wish.

2. Implement `httpserver_process.c` to create children processes to send HTTP responses

   - When the original process receives a new HTTP request, it should create a new child process to send an HTTP response. The parent process does not need to wait for the child process to finish executing — it should resume listening for new requests as soon as possible.

3. Implement `httpserver_thread.c` to create threads to send HTTP responses

   - Use the `pthreads` thread library that we've discussed in section. The section handout is a good resource.

   - When the original process receives a new HTTP request, it should create a new thread to send the HTTP response. This new thread does not need to be `join`'d with the original.

4. Implement a fixed-sized thread pool for handling multiple client request concurrently in `httpserver_pool.c`.

   - Use the `pthreads` thread library that we've discussed in section. The section handout is a good resource.

   - Your thread pool should be able to concurrently serve exactly `--concurrency` clients and no more. Note that we typically use `--concurrency + 1` threads in our program: the original thread is responsible for `accept()`-ing client connections in a while loop and dispatching the associated requests to be handled by the threads in the thread pool.

   - You'll need to make your server create `--concurrency` new threads that each run the `handle_clients` function.

   - When a new HTTP request is dispatched by the original thread, the client socket fd should be `wq_push`'d onto the dispatcher's work queue.

   - Threads in the pool should make calls to `wq_pop` for the next client socket file descriptor. If the queue is empty, calls to `wq_pop` will block.

   - After successfully popping a to-be-served client socket fd, call the appropriate `request_handler` to handle the client request.

   - Once the thread is finished serving the client request, it will either (1) serve the next request in the queue or (2) wait until a new request is received.

5. We can use the `ab` utility to measure the performance of a server. Try it out:

   (a) Run `./httpserver_basic --files files/` in your terminal.

   (b) In a separate window in your terminal, run the command:
       `ab -n 10 -c 1 http://192.168.162.162:8000/`

   (c) `ab` reports (1) the mean time per request and (2) the mean time across concurrent requests. Read `man ab` to learn about the optional arguments `n` and `c` (**WARNING:** if you Google `man ab`, you will not only get the man pages for `ab` but also images of chiseled and defined abdominal muscles.). What happens to the mean time per request as `n` grows large (test n=10, 25, 50, 100)? What happens when `c` grows large (test c=1, 10, 25, 100)?

   (d) Open up `hw4.txt` and answer the questions inside. The questions are reproduced here:

       i. Run `ab` on `httpserver_basic`. What happens when `n` and `c` grow large?

    ii. Run `ab` on `httpserver_process`. What happens when `n` and `c` grow large? Compare these results with your answer in the previous question.

    iii. Run `ab` on `httpserver_thread`. What happens when `n` and `c` grow large? Compare these results with your answers in the previous questions.

    iv. Run `ab` on `httpserver_pool`. What happens when `n` and `c` grow large? Compare these results with your answers in the previous questions.

3.5 Submission

To submit and push to autograder, first commit your changes, then do:

`git push personal master`

    Within 30 minutes you should receive an email from the autograder. (If you haven't received an email within half an hour, please notify the instructors via a private post on Piazza.)

A Function reference: libhttp

We have provided some helper functions to deal with the details of the HTTP protocol. They are included in the skeleton as `libhttp.c` and `libhttp.h`. These functions only implement a small fraction of the entire HTTP protocol, but they are more than enough for this assignment.

A.1 Example usage

Reading a HTTP request from a socket `fd` only involves a single function call.

```
// Returns NULL if an error was encountered.
struct http_request *request = http_request_parse(fd);
```

Sending a HTTP response is a multi-step process. First, you should send the HTTP status line using `http_start_response`. Then, you can send any number of headers with `http_send_header`. After all the headers are sent, you MUST call `http_end_headers` (even if you didn't send a single header). Finally, you can use `http_send_string` (for null-terminated C strings) or `http_send_data` (for binary data) to send your data.

```
http_start_response(fd, 200);
http_send_header(fd, "Content-type", http_get_mime_type("index.html"));
http_send_header(fd, "Server", "httpserver/1.0");
http_end_headers(fd);
http_send_string(fd, "<html><body><a href='/'>Home</a></body></html>");
http_send_data(fd, "<html><body><a href='/'>Home</a></body></html>", 47);
close(fd);
```

A.2 Request object

A `http_request` struct pointer is returned by `http_request_parse`. This struct contains just two members:

```
struct http_request {
  char *method;
  char *path;
};
```

A.3 Functions

- `struct http_request *http_request_parse(int fd)`
  Returns a pointer to a `http_request` struct containing the HTTP method and the path that of a request that is read from the socket. This function will return `NULL` if the request is invalid. This function will block until data is available on `fd`.

- `void http_start_response(int fd, int status_code)`
  Writes the HTTP status line to `fd` to start the HTTP response. For example, when `status_code` is 200, the function will produce `HTTP/1.0 200 OK\r\n`

- `void http_send_header(int fd, char *key, char *value)`
  Writes a HTTP response header line to `fd`. For example, if `key` is equal to `"Content-Type"` and the `value` is equal to `"text/html"` this function will write `Content-Type: text/html\r\n`

- `void http_end_headers(int fd)`
  Writes a CRLF (`\r\n`) to `fd` to indicate the end of the HTTP response headers.

- `void http_send_string(int fd, char *data)`
  Alias for `http_send_data(fd, data, strlen(data))`.

- `void http_send_data(int fd, char *data, size_t size)`
  Sends `data` to `fd`. If `data` is too large to be written all at once, this function will call `write()` in a loop to send the data one piece at a time.

- `char *http_get_mime_type(char *file_name)`
  Returns a string for the correct `Content-Type` based on `file_name`.