HW 4: HTTP Server

CS 162

Due: March 10, 2020

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
```

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

3.1 HTTP Server Outline

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. Do **one** of two things: (determined by command line arguments)
    - Serve a file from the local file system, or yield a 404 Not Found
    - Proxy the request to another HTTP server.

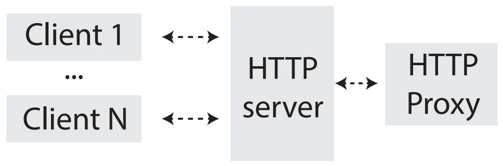


Figure 1: when using a proxy, the http server serves requests by streaming them to a remote http server (proxy). responses from the proxy are sent back to clients.

    The httpserver will be in **either** file mode or proxy mode. It does not do both things at the same time.
6. Send the appropriate HTTP response header and attached file/document back to the client (or an error message)

The skeleton code already implements steps 2-4. Part 1 of this assignment is to bind the server socket to an address and to listen for incoming connections. This will be done with the `bind()` and `listen()` syscalls. Parts 2 and 3 of this assignment is to serve files and directories to the client. Part 4 of this assignment is to implement a proxy server. You will then implement a variety of methods to handle client requests (Parts 5-7) and test the performance of each implementation (Part 8).

3.2 Usage ./httpserver

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

```
$ ./httpserver --help
Usage: ./httpserver --files any_directory_with_files/ [--port 8000 --num-threads 5]
       ./httpserver --proxy inst.eecs.berkeley.edu:80 [--port 8000 --num-threads 5]
```

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 www/`".
- `--proxy` — Selects an "upstream" http server to proxy. The argument can have a port number after a colon (e.g. inst.eecs.berkeley.edu:80). If a port number is not specified, port 80 is the default.

- `--port` — Selects which port the http server listens on for incoming connections. Use in both files mode and proxy mode. (This is different from the proxy port.) If a port number is not specified, port 8000 is the default.
- `--num-threads` — Indicates the number of threads in your thread pool that are able to concurrently serve client requests. This argument is initially unused and it is up to you to use it properly.

You should not specify both `--files` and `--proxy` at the same time, or the later option will override any earlier one. The `--proxy` option can also take an IP address.

The `--num-threads` argument is used to specify the amount of worker threads in the thread pool. This will only be used in Part 7 of the assignment.

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 --files www/`".

Running `make` will give you 4 executables: `httpserver`, `forkserver`, `threadserver`, and `poolserver`. Parts 1-4 will use `httpserver`. Part 5 is to implement `forkserver`; Part 6 is to implement `threadserver`; Part 7 is to implement `poolserver`. Part 8 is to load test each of these servers to compare/contrast their performances.

3.3 Accessing the HTTP Server

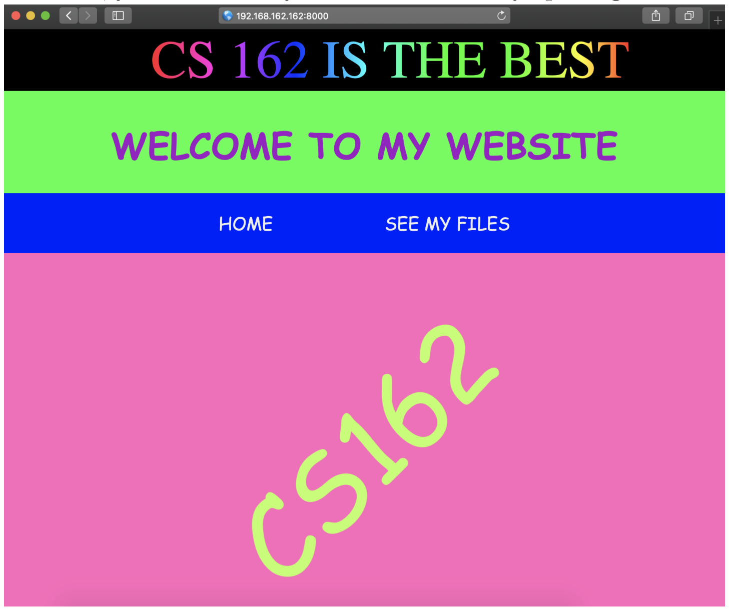
You can send HTTP requests with the `curl` program, which is installed on your VM. An example of how to use `curl` is:

```
$ curl -v http://192.168.162.162:8000/
$ curl -v http://192.168.162.162:8000/index.html
$ curl -v http://192.168.162.162:8000/path/to/file
```

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.)
```

After Part 3, you can access your HTTP server by opening a web browser and going to http://192.168.162.162:8000/.



3.4 Common error messages

3.4.1 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 --files files/ --port 8001`", or you can reboot your VM with "`vagrant reload`".

3.4.2 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).

3.5 Your Assignment

1. Finish setting up the server socket in the `serve_forever()` function.

    - Bind the socket to an IPv4 address and port specified at the command line (i.e. `server_port`) with the `bind()` syscall.
    - Afterwards, begin listening for incoming clients with the `listen()` syscall. At this stage, a value of 1024 is sufficient for the `backlog` argument of `listen()`. When load testing in Part 8, you may play around with this value and comment on how this impacts server performance.
    - After finishing Part 1, `curl` should output "Empty reply from server". There are no autograder tests for Part 1.

2. Implement `handle_files_request(int fd)` to handle HTTP GET requests for files. You will need to call `serve_file()` accordingly. You should also be able to handle requests to files in subdirectories of the files directory (e.g. `GET /images/hero.jpg`).

    - If the file denoted by `path` exists, call `serve_file()` on it. Read the contents of the file and write it to the client socket.
        - 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`. You can use `snprintf()` to convert an integer into a string.
        - You *must* use the `read()` and `write()` syscalls for this assignment. Any implementations using `fread()` or `fwrite()` will not earn any credit. This is purely for pedagogical reasons: we want you to be comfortable with the fact that low-level I/O may or may not perform the entire operation on all the bytes requested.
    - Else serve a 404 Not Found response (the HTTP body is optional) to the client. 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.
    - After finishing Part 2, `curl`ing for index.html should output the contents of the file `index.html`. There are a few autograder tests for Part 2.

3. Implement `handle_files_request(int fd)` to handle HTTP GET requests for both files and directories.

    - You will now need to determine if `path` in `handle_files_request()` refers to a file or a directory. The `stat()` syscall and the `S_ISDIR` or `S_ISREG` macros will be useful for this purpose. After finding out if `path` is a file or a directory, you will need to call `serve_file()` and `serve_directory()` accordingly.
    - If 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.)
        - The `http_format_index()` function in `libhttp.c` may be useful.
    - If 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**.
        - The `http_format_href()` function in `libhttp.c` may be useful.
        - To list the contents of a directory, good functions to use are `opendir()` and `readdir()`
    - If the directory does not exist, serve a 404 Not Found response to the client.

- 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.
- 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)
- Remember to close the client socket before returning from the `handle_files_request()` function.
- Make helper functions to re-use similar code when you can. It will make your code easier to debug!
- After finishing Part 3, `curl`ing for the root directory `/` should output the contents of the file `index.html`. All tests for `[Basic Server]` should pass.

4. Implement `handle_proxy_request(int fd)` to proxy HTTP requests to another HTTP server.

   We've already handled the connection setup code for you. You should read and understand it, but you don't need to modify it. In short, here is what we have done:

   - We use the value of the `--proxy` command line argument, which contains the address and port number of the upstream HTTP server. (These two values are stored in the global variables `char *server_proxy_hostname` and `int server_proxy_port`.
   - We do a DNS lookup of the `server_proxy_hostname`, which will look up the IP address of the hostname (check out `gethostbyname2()`).
   - We create a network socket and connect it to the IP address that we get from DNS. Check out socket() and connect().
   - htons() is used to set the socket's port number (integers in memory are little-endian, whereas network stuff expects big-endian). Also note that HTTP is a `SOCK_STREAM` protocol.

   Now comes your part! Here is what you need to take care of:

   - Wait for new data on both sockets (the HTTP client fd, and the target HTTP server fd). When data arrives, you should immediately read it to a buffer and then write it to the other socket. You are essentially maintaining 2-way communication between the HTTP client and the target HTTP server. **Your proxy must support multiple requests/responses.**
     Hints:
     - This is more tricky than writing to a file or reading from stdin, since you do not know which side of the 2-way stream will write data first, or whether they will write more data after receiving a response. In proxy mode, you will find that multiple HTTP request/responses are sent within the same connection, unlike your HTTP server which only needs to support one request/response per connection.
     - You should use pthreads for this task. Consider using two threads to facilitate the two-way communication, one from A to B and the other from B to A.
     - **Do not use** `select()`, `fcntl()`, or the like. We used to recommended this approach in previous semesters, but we've found this method to be too confusing.
   - If either of the sockets closes, communication cannot continue, so you should close both sockets to terminate the connection.
   - After finishing Part 4, all `[Proxy]` tests should pass on the autograder.

5. Implement `forkserver`. You won't be writing much new code. With the conditional compilation preprocessor directives, we only need to change *how* we call the `request_handler()` in each of these different servers.

- The child process should call the `request_handler()` with the client socket fd. After serving a response, the child process will terminate.
- The parent process will continue listening and accepting incoming connections. It will *NOT* wait for the child.
- Remember to close sockets appropriately in both the parent and child process.

6. Implement `threadserver`.
   - Create a new pthread to send the proper response to the client.
   - The original thread continues listening and accepting incoming connections. It will *NOT* join with the new thread.
7. Implement a fixed-sized thread pool for handling multiple client request concurrently.
   - Your thread pool should be able to concurrently serve exactly `--num-threads` clients and no more. Note that we typically use `--num-threads + 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.
   - Begin by looking at the functions in `wq.c/h`.
     - The original thread (i.e. the thread you started the `httpserver` program with) should `wq_push` the client socket file descriptors received from `accept` into the `wq_t work_queue` declared at the top of `httpserver.c` and defined in `wq.c/h`.
     - Then, threads in the thread pool should use `wq_pop` to get the next client socket file descriptor to handle.
   - You'll need to make your server spawn `--num-threads` new threads which will spin in a loop doing the following:
     - Make blocking calls to `wq_pop` for the next client socket file descriptor.
     - After successfully popping a to-be-served client socket fd, call the appropriate `request_handler` to handle the client request.
8. Test, measure, and comment on the performance of `httpserver`, `forkserver`, `threadserver`, and `poolserver`.
   - We will be using the Apache HTTP server benchmarking tool (`ab` for short) to load test each server type. To install `ab`, run the following command:

     ```
     $ sudo apt-get install apache2-utils
     ```
   - Run `./httpserver --files www/` in your terminal.
   - In a separate terminal window, run the command:

     ```
     $ ab -n 500 -c 10 http://192.168.162.162:8000/
     ```
   - This command issues 500 requests at a concurrency level of 10 (meaning it dispatches 10 requests at a time). Read `man ab` to learn more about the tool. You can type `man ab` in your terminal or your preferred search engine. However, please note that typing `man ab` into Google will also give you defined images of chiseled male abdominal muscles; do so at your own discretion.
   - Notice how `ab` outputs the mean time per request. Take note of this value and comment on how it changes when we change *how* the server handles requests.

- Use `ab` to load test `forkserver`, `threadserver`, and `poolserver`. Play around with the `n` and `c` variables as well as the size of the thread pool in `poolserver`.
- Answer the questions on Gradescope.

9. Congratulations on implementing your own HTTP server!

3.6 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 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.2 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.

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

- `void http_format_href(char *buffer, char *path, char *filename)`
  Puts `<a href="/path/filename">filename</a><br/>`into the provided buffer. The resulting string in the buffer is null-terminated. It is the caller's responsibility to ensure that the buffer has enough space for the resulting string.

- `void http_format_index(char *buffer, char *path)`
  Puts `path/index.html` into the provided buffer. The resulting string in the buffer is null-terminated. It is the caller's responsibility to ensure that the buffer has enough space for the resulting string.