HW 4: HTTP Server

CS 162

Due: July 29, 2021

Contents

1 Introduction 2
  1.1 Getting Started ......................................................... 2
  1.2 Setup Details .............................................................. 2

2 Background 3
  2.1 Structure of a HTTP Request ............................................ 3
  2.2 Structure of a HTTP Response .......................................... 4

3 Your Assignment 5
  3.1 HTTP Server Outline .................................................... 5
  3.2 Usage of httpserver ..................................................... 5
  3.3 Accessing the HTTP Server .............................................. 5
  3.4 Common error messages ................................................ 6
     3.4.1 Failed to bind on socket: Address already in use ............... 6
     3.4.2 Failed to bind on socket: Permission denied .................. 6
  3.5 Your Assignment .......................................................... 7
  3.6 Submission ................................................................. 8

A Reference 9
  A.1 Request object ........................................................... 9
  A.2 Functions ................................................................. 9
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 (e.g. file, text, binary data) that was requested by the client.

In this assignment, you will implement a 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 hw5
```

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 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 a HTTP Request

The format of a HTTP request message is:

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

Header lines provide information about the request. 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`\(^4\)

\(^3\)For a deeper understanding, open your web browser’s developer tools, then click on the “Network” tab and look at the headers sent when you request any webpage.

\(^4\)Or at least this was the idea back in the early days of the web. Now the User-Agent is generally an unholy mess (try Googling “What’s my user agent”). If you’re curious as to why this is the case, the history behind it is amusing.
2.2 Structure of a 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 human-readable description of the status code
- Zero or more HTTP header lines
- A blank line (i.e. a CRLF by itself)
- The body (i.e. content) requested by the HTTP request

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

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 84

<html>
<body>
<h1>Hello World</h1>
<p>
Let's see if this works
</p>
</body>
</html>
```

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:\n
- 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

---

5For those curious, more information can be found [here](#)
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 for 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
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 are to serve files and directories to the client. You will then implement a variety of methods to handle client requests (Parts 4-5).

3.2 Usage of `httpserver`

Below is a description of how to invoke `httpserver` from the shell. The argument parsing step has already been implemented for you:

```
./httpserver --help
Usage: ./httpserver --files any_directory_with_files/ [--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 hw5/ folder (e.g. if you are currently cd’ed into the hw5/ folder, you should just use “--files www/”).
- `--port` — Selects which port the HTTP server listens on for incoming connections. Used in both files mode and proxy mode. 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.

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 --port PORT --files www/”.

Running `make` will give you 3 executables: `httpserver`, `threadserver`, and `poolserver`. Parts 1 – 3 will use `httpserver`. Part 4 is to implement `threadserver`. Part 5 is to implement `poolserver`.

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:

As always, see the `man` pages on your VM for more information.

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 '(...\server')`”. If that doesn’t work, you can specify a different port via `--port`, 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

Part 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()`.
- After finishing Part 1, `curl` should output "Empty reply from server". There are no autograder tests for Part 1.

Part 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, curling for `index.html` should output the contents of the file `index.html`. There are a few autograder tests for Part 2.

Part 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()` or `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 -l`), 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 (i.e. you do not need to handle symbolic links, pipes, or 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, curling for the root directory / should output the contents of the file index.html. All tests for [Basic Server] should pass.

Part 4. 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.

Part 5. Implement a fixed-sized thread pool for handling multiple client requests 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.

Congratulations on implementing your own HTTP server!

3.6 Submission
To submit and push to the autograder, first commit your changes, then run:
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  Reference

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:

```c
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 path of a request that is read from the socket fd. 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

- `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

- `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>` 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.