CS 162 HW2

David Culler, Arka Bhattacharya, William Liu, Roger Chen

Due: March 2 2015

Contents

1 Introduction

The Hypertext Transport Protocol (HTTP) is the most commonly used protocol on the Web today. Like most 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; the server then replies with an HTTP response message, usually containing some resource (file, text, binary data) that was requested by the client.

In this assignment, you will implement an HTTP server that services GET requests. You will play around with some HTTP response headers, add protection on the server side through the use of processes, and return error code pages. 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 hw2
```

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.

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/37.0.2062.94\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 or response, or about the object sent in the message body[3]. Here are some example header fields:

- **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/37.0.2062.94`.

[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.

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
\r\n
<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:

- 1xx indicates an informational message only
- 2xx indicates success
- 3xx redirects the client to another URL
- 4xx indicates an error on the client's part
- 5xx indicates an error on the server's part

Some example response headers are as follows:

- **Content-Type**: gives the MIME type of the data attached to the response, such as `text/html` or `text/plain`.
- **Content-Length**: gives the number of bytes in the body.[4]

[4] In this assignment, you do not have to implement the Content-Length header field for an HTTP response.

3 Your Assignment

3.1 HTTP Webserver Outline

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

1. Create a listening socket
2. Accept a connection with the socket
3. Obtain a new connection socket
4. Fork a child process to handle the new connection socket
5. Parent process goes back to accept more connections on the original socket
6. Child Process reads in and parses the HTTP request
7. Child Process will 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

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

Currently, the skeleton code does steps 1-5 already. There are helper functions in `libhttp.c` to help you do step 6 and step 8. **Your main goal is to implement step 7.**

3.2 Usage ./httpserver

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

```
vagrant@development [00:00:00] hw2 $ ./httpserver --help
Usage: ./httpserver --files www_directory/ --port 8000
       ./httpserver --proxy inst.eecs.berkeley.edu:80 --port 8000
```

The available options are:

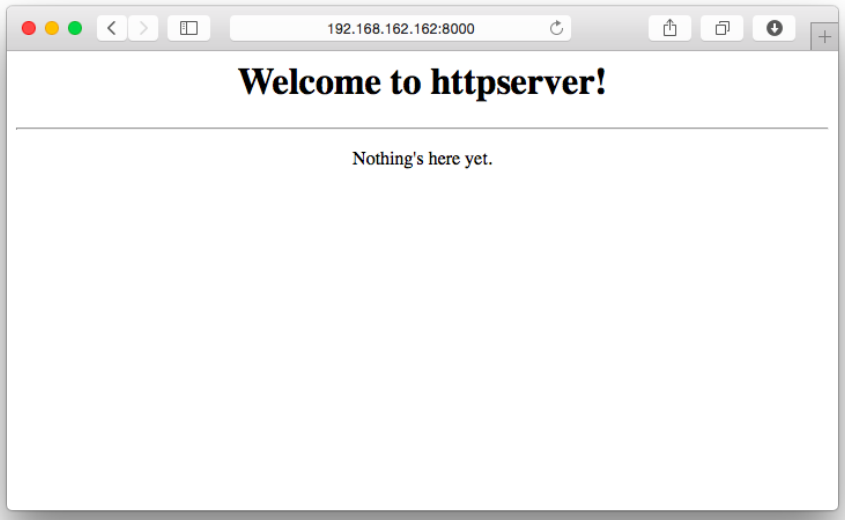
- `--files` — Selects a directory from which to serve files. You should be serving files from the hw2/ folder (e.g. if you are currently cd'ed into the hw2/ folder, you should just use `--files ./`, which is also the default behavior.
- `--proxy` — Selects an "upstream" http server to proxy. The argument 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.)

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.

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 [OPTIONS]`.

3.3 Accessing the http server

Check that your http server works by opening your web browser and going to http://192.168.162.162:8000/.



You can also 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/
```

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):

```
vagrant@development [00:00:00] hw2 $ nc -v 192.168.162.162 8000
Connection to 192.168.162.162 8000 port [tcp/*] succeeded!
(Now, type out your HTTP request here.)
```

3.4 Your Assignment

1. Implement `handle_files_request(int fd)` 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.
     – 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.)
     – 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 you don't find a slash at the beginning, you can just error out or ignore the request.
   - 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.
   - Otherwise, return a 404 Not Found response (the HTTP body is optional). Since there are many things that can go wrong during an HTTP request, we will not expect you to support any error messages other than 404 Not Found in response to a non-existent file.
   - **You only need to handle one HTTP request/response per connection.** In other words, you do not need to implement HTTP connection keep-alive or HTTP pipelining.

2. Implement `handle_proxy_request(int fd)` to proxy HTTP requests to another HTTP server. Your handler should:

- 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`.
- Do a DNS lookup of the server_proxy_hostname, which will look up the IP address of the hostname.

  Hints:
  - Use the `gethostbyname` function or `getaddrinfo` function (or related functions) to do this.
  - If there are multiple DNS IP address results, you can just pick any one. You do not need to implement DNS retries or IPv6. Feel free to look up example code on how to get an IP address from a hostname in C.
- Create a network socket and connect it to the IP address that you got from DNS.

  Hints:
  - Good functions to use are: `socket()`, `connect()`.
  - You will need the `htons()` function to set the socket's port number (integers in memory are little-endian, whereas network stuff expects big-endian).
  - HTTP is a SOCK_STREAM protocol.
  - There are some similar functions and structs used in the `serve_forever()` function. You may want to take a peek for some syntax tips.
- Wait for new data on both sockets (the HTTP client fd, and the upstream 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 upstream HTTP server.

  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 have two straightforward options on how to implement 2-way communication: select() and pthreads.
  - If you are using select(), good functions to use are: `select()`, `read()`, `write()`, and the macros `FD_ZERO`, `FD_SET`, `FD_ISSET`.
  - If you are using pthreads, consider using two threads to facilitate the two-way communication. One from A to B and the other from B to A.
- If either of the sockets closes, communication cannot continue, so you should close the other socket and exit the child process.

3.5 Submission

Push your code to the autograder branch `ag/hw2` on github, and push the final code release to the branch `release/hw2` on github.