162 HW2

David Culler, Arka Bhattacharya, William Liu

September 2014

Contents

1 Introduction

The Hypertext Transport Protocol (HTTP) is the most commonly used protocol on the Web today. Like most network protocols, HTTP uses the client-server model: an HTTP client opens a connection and sends a request message to an HTTP server; the server then returns a response message, usually containing the resource that was requested. After delivering the response, the server closes the connection (making HTTP a stateless protocol, i.e. not maintaining any connection information between transactions).

This assignment has you implement a simple HTTP server that gives you a chance to service GET requests, 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.

The appendices provide some examples of how you could extend this simple HTTP web server. Implementing those extensions are optional (you could turn them in for epsilons).

Pull the skeleton code from the staff repository.

```
git pull staff master
cd hw2
```

1.1 Setup Details

Background : Currently your vagrant virtual machine is set up to be on an internal private network accessing the outside network through a Virtual Box NAT implementation. This allows network connections to originate in your virtual machine, but prevents connections that originate outside to connect to your virtual machine. Thus, if you run your http server on your virtual machine, it will not be able to access connections from any client on the internet, not even your host operating system.

One solution is to set up *port forwarding* on your Virtual Box implementation. Once port forwarding is set up, Virtual Box listens to all packets received on that port on the host operating system, and forwards those packets to the virtual machine on the specified port.

Enabling Port Forwarding : To set up port forwarding, open the Virtual Box graphical user interface where all your virtual machines are listed. Choose the settings option of your virtual machine, and go to the "Network" pane. You will find a button labeled `Port Forwarding`. Clicking that button will open a dialog box where you can specify the port mapping from your host operating system to your guest operating system. For example, if you map port 50000 on your host machine to port 55000 on your guest machine, you can run your HTTP server on port 55000 on your guest machine, and you would send it packets at the address 127.0.0.1:50000.

Another way to enable Port Forwarding: You can also enable port forwarding by changing your Vagrantfile. Follow the instructions specified here.

1.2 Structure of HTTP Request

The format of a HTTP request message is as follows: an initial request line, zero or more header lines, a blank line (i.e. a CRLF by itself). Given below is a HTTP request message sent by the Google Chrome browser to a HTTP web server running on the local machine (127.0.0.1) on port 50000 :

```
GET /hello.html HTTP/1.0\r\n
Host: 127.0.0.1:50000\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[1] . Here are some example header fields:

Host: gives the IP address and port number of the host where the resource exists.

User-Agent: identifies the program that's making the request, in the form "Program-name/x.xx", where x.xx is the (mostly) alphanumeric version of the program. In the above example, the Google Chrome browser sets User-Agent as `Chrome/37.0.2062.94`.

1.3 Structure of HTTP Response

A HTTP web server responds as follows:

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

The HTTP response has two parts — the response headers and the message-body. The first line of the response header is called the status line, also has three parts separated by spaces: the HTTP version, a response status code that gives the result of the request, and an English reason phrase describing the status code. 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 of some kind
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 in the body, such as text/html or text/plain.

Content-Length: gives the number of bytes in the body.[2]

1.4 HTTP Webserver Skeleton

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

[1] To get a deeper understanding, open the web developer view on your favorite browser and look at the header sent when you request any webpage

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

1. Create a listening socket
2. Accept a connection with it
3. Obtain a new connection socket
4. Fork a child process to service the new connection socket
5. Parent process goes back to accept more connections on the original socket
6. Child Process reads in the HTTP request header
7. Child Process sends the appropriate HTTP response header
8. Child sends the entity requested (e.g. an HTML document), or an error message.

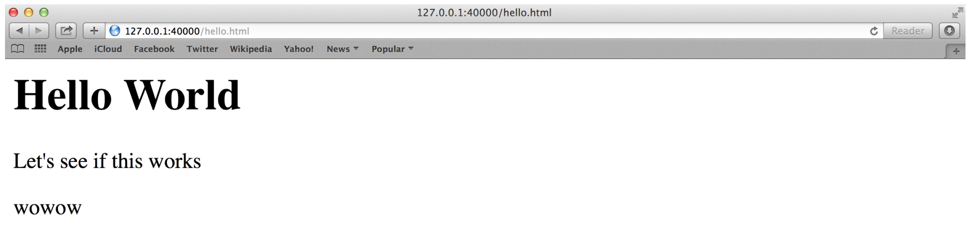
Currently, the skeleton code does the first five steps listed above. It implements a web server which responds to all connections by echoing the request to the client with the header `Content-Type` set to `text/plain`. A majority of the networking steps are done for you in the function: **server(int portno)**.

1.5 Your Assignment (∼40 lines of code)

1. Replace the echo logic currently in `proces_http_request(int httpsockfd)` to implement a buffered reader to service `GET` requests only. Read the requested resource and send it over the socket. Your response headers should contain the appropiate status line and the field `Content-Type` set to `text/html`. You do not need to implement any other HTTP header. Keep in mind that resources specified in the GET request should be treated as a pathname relative to the `www` directory in `hw2` (i.e. the request 127.0.0.1:50000/hello.html should resolve to `hw2/www/hello.html` on your virtual machine). Close the connection socket to the client once you have served a request.

2. If it is not a GET request, return a `400 Bad Request` error. Construct the header in the correct format [3] and use the `400.html` file in the `hw2` folder as the message-body. If the document is not found, return a `404 Not Found` error in the same manner (construct the header yourself and send `404.html` in the `hw2` folder as the message-body.

   Check that your webserver works by making HTTP requests from your favorite browser. For instance, below is a screenshot when Safari sends a `GET` request for `hello.html` (which exists) on a webserver running on port 40000.


127.0.0.1:40000/hello.html
Reader
Apple iCloud Facebook Twitter Wikipedia Yahoo! News Popular

Hello World

Let's see if this works

wowow

[3]as specified in the HTTP spec

Push your code to the autograder branch ag/hw2 on github.

```
git add .
git commit -m "Implemented http web server"
git push personal master
git checkout -b ag/hw2
git push personal ag/hw2
```

Push the final code release to the branch `release/hw2` on github.

```
git add .
git commit -m "hw2 submission"
git push personal master
git checkout -b release/hw2
git push personal release/hw2
```

2 Experience with pthreads

We will be providing some additional exercises soon to give you a chance to get experience with thread programming, but these will not need to be turned in for grading.

Appendices

There are several things you could do to make your http server more robust and more complete. Feel free to extend it. The appendix provides a few example developments that you could try. Feel free to turn these in for epsilons.

A [Optional] Resolving content-type

A.1 Background

The HTTP server can provide a client (e.g, a browser) with hints on how to interpret the content of the webpage returned. Suppose that the server returns the following content.

```
<html>
<body>
<b> Bold letters </b>
</body>
</html>
```

The web browser can interpret this content as either plain text, in which case it display the following on the browser screen :

```
<html>
<body>
<b> Bold letters </b>
</body>
</html>
```

However, if the browser interprets this content as html text, it would simply display :

Bold Letters.

A.2 Implementation (∼10 lines of code)

Change the `Content-Type` field in your HTTP response to specify whether a returned file should be interpreted as a 'plain text' or an 'html' file. You may assume that files to be interpreted as 'html' ends in `.html`. All other files should be interpreted as text files.

B [Optional] Resolving GET directory requests

B.1 Background

In this section we shall deal with the case when a user's requested resource evaluates to a directory instead of a file on the web-server.

Example: Suppose the `www` folder in your homework folder hw2 has a sub-directory `dir`, which has two files `test1.html` and `test2.html`. What happens when the client makes a request `127.0.0.1:50000/dir/` ?
There can be multiple ways of handling this case – but we shall be implementing one specific method.

B.2 Implementation (∼60 lines of code)

If the requested directory has an `index.html` file, then web-server should return this index file. If, however, the requested directory does not contain an `index.html` file, the web-server should return a webpage listing all the files (and sub-directories) in that directory.

C [Optional] Ensuring user does not escape the web directory

C.1 Background

The web server you have implemented till now has a security flaw. A malicious client might use various escape characters in her resource request to obtain restricted files on the webserver. A user might make a request `GET ../../etc/apache2/apache2.conf` to escape out of the `www` folder and find out the exact configurations of your apache server.

C.2 Implementation (∼20 lines of code)

Your webserver should ensure that a malicious user cannot escape the directory which stores your webpages (www). Your webserver should only return pages which are contained within the `www` directory or in its sub-tree in the file system. If a `GET` request tries to request a resource which is outside the directory structure of `www`, you should return a `403 Forbidden` error message. Again, construct the header for the 403 Forbidden response yourself, and return the contents of `403.html` as the message body.

Push your final release code to `release/hw2`