Concurrent Web Server using BSD Sockets

CS118 WINTER 2015

TAEK-SANG PETER KIM

204-271-299

Introduction

The purpose of this first lab assignment was to implement a simple web server in C, utilizing socket programming. On the surface level, the simple server listens to all connection attempts made by a client (in this case, the client is the local internet browser such as Firefox) and dumps these request messages to the console. Consequently, this server can heed these request by generating an appropriate response header along with the actual data of the file requested, and send these back to the client for display.

High Level Design

To begin with, the web server I implemented is directly based off of the `server.c` skeleton code provided on CCLE.

When the server is first run, it continuously listens to any incoming requests from clients. These clients must be initiating the request to the correct port number, which in this case is set to 1738 by default. When the client successfully connects to the server, a child process is forked to handle and honor each request in two steps. First, as the part (A) of the lab specifications indicate, the raw incoming HTTP request from the client is dumped onto the console. Finally, the process generates a HTTP response message consisting of the requested file preceded by header lines, and consequently sends it to the client.

After dumping out the HTTP request message, the process parses this exact request message and extracts the file name that was requested. If no file was requested, the process sends a 404 status header and a basic 404 Not Found HTML page. Afterwards, the process attempts to open the given file name in the same directory and placing the returned value in a file pointer. If this pointer returns null i.e. the file does not exist in the directory, the same 404 status header and HTML page is sent to the client, similar to before.

Subsequently, an appropriate HTTP response message is dynamically generated by identifying and constructing the following components: status code, connection status, date/time of response formed, server information, date/time of last modification, content length, and content type. Here, the server information is named as a pseudo server of `CS118Lab1Pseudo`. Also, the content type is determined by parsing the file name string, which was obtained earlier in the process, and checking the substrings for possible matches of file extension names. In this design, the extensions `txt`, `html`, `jpeg`, `jpg`, and `gif` are supported as requested by the specification.

Similarly, to the original request message, the constructed HTTP response message is dumped onto the console for analysis before being sent to the client. The actual requested data/file is also then sent to the client and successfully served as one can check through the client itself.

Finally, the server can continue to accept and honor more requests from multiple clients until it is closed forcefully from the console using CTRL-C.

Challenges Faced

The most significant challenge faced while implementing this simple webserver was understanding the proper C way of opening a file, handling its pointer, determining its size, and freeing its memory. I realized that there is significant overhead when opening and allocating enough memory for a file using the methods `fseek`, `ftell`, and `fread`. Also, towards the near completion of the lab, I was constantly faced with segmentation faults and random memory leaks and it turned out that I had forgotten to free the file memory and close the file connection. All in all, it was a great learning experience as I finally understood the file stream functions in the C language.

Compilation & Execution Instruction

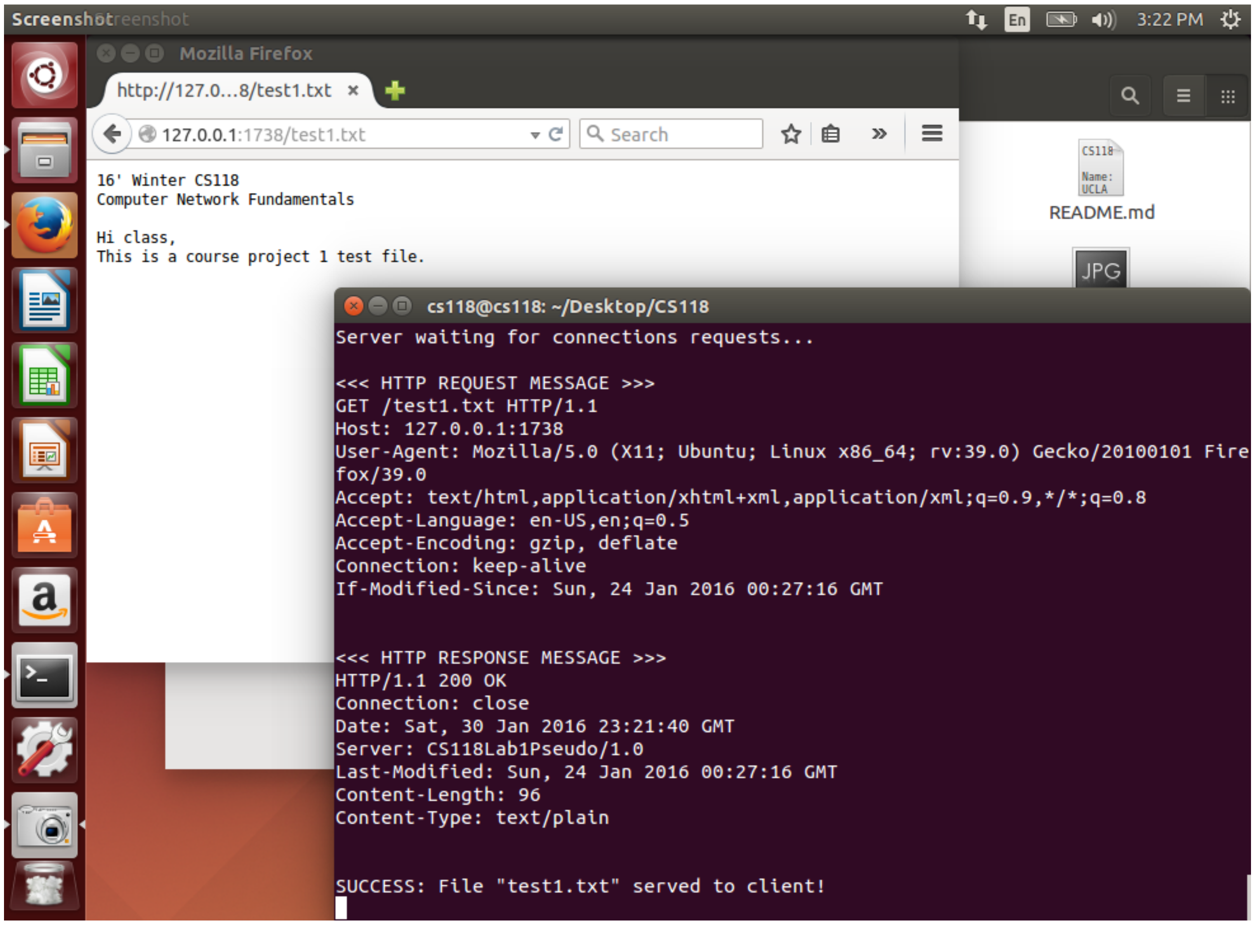
To compile the server, navigate on the console/terminal to the root directory containing the file `webserver.c` , and type `make`. Through the `Makefile`, the executable `webserver` will then be automatically produced.

To run the server, simply type `./webserver` onto the console.

To connect to the running server, open a local browser client such as Firefox and enter the URL `127.0.0.1:1738/some_file.html`. As mentioned in the design explanation, the default port number is set to 1738. One can change this to any four-digit number one wishes to be by simply modifying it in the source code. The file argument `some_file.html` can be replaced with any file that is existent in the same file directory. In the current state of the lab submission, one can request `test1.txt`, `test2.jpg`, and `test3.gif` files.

Sample Outputs

The following section will cover a sample output of the running server. After one complies and executes the webserver, one can request the file test1.txt on the server by navigating to `127.0.0.1:1738/test1.txt` on a local client which in this case is Firefox. The result on both the client and console is shown in the screenshot below.



```
16' Winter CS118
Computer Network Fundamentals

Hi class,
This is a course project 1 test file.
```

```
Server waiting for connections requests...

<<< HTTP REQUEST MESSAGE >>>
GET /test1.txt HTTP/1.1
Host: 127.0.0.1:1738
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:39.0) Gecko/20100101 Fire
fox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
If-Modified-Since: Sun, 24 Jan 2016 00:27:16 GMT


<<< HTTP RESPONSE MESSAGE >>>
HTTP/1.1 200 OK
Connection: close
Date: Sat, 30 Jan 2016 23:21:40 GMT
Server: CS118Lab1Pseudo/1.0
Last-Modified: Sun, 24 Jan 2016 00:27:16 GMT
Content-Length: 96
Content-Type: text/plain


SUCCESS: File "test1.txt" served to client!
```

As one can clearly perceive from the generated output, both the HTTP request message and the HTTP response message is dumped into the console. The latter is dynamically generated through our process and sent by the server to the client prefacing the actual data/file. The former is sent by the client to the server initially, and was parsed on-the-fly to extract the file name.

Conclusion

Ultimately, through this lab assignment, I learned a great deal about designing and implementing a web server via socket programming conventions. Furthermore, I am now familiar with the individual components of a standard HTTP request/response message between clients and servers, and how to construct these messages through C.