**Com Sci 118 Computer Network Fundamentals**

**Project 1: Concurrent Web Server using BSD Sockets**

Jeffrey Shu (004124780) (SEASNET: jeffreys)

Vincent Wong (604139038) (SEASNET: tat)

**High Level Description of Server Design**

As the project specs are fairly vague about how much functionality our webserver needs to have, we decided to implement only the basic protocols necessary to serve resources properly via Mozilla Firefox, disregarding any other caching or efficiency mechanisms. This also means that only the GET method is supported, and anything else (such as POST) are considered bad requests.

The structure of our webserver is similar to the example server code provided in the VM image. We construct our socket and bind it to a port that we specify via command line so that it can begin listening. If we receive a new connection, the webserver will fork a child process to handle the connection and serve the HTTP request. The parent then closes its copy of the socket and continues listening for other new connections. A signal handler is employed to reap any child processes that become zombies after termination.

Serving an HTTP request is a 2-step process. The webserver must first validate the request format and extract each line in the HTTP header. To ease implementation, we chose to enforce a limit of 32 header lines, including the request line and empty line, as well as a request size limit of 4096 bytes. Any request that goes past those limits will be rejected with a HTTP 413 error.

Once the header lines are successfully extracted, we can begin parsing the request. The only important item that we really need is the requested file inside the GET method line. Because our webserver does not support any extra features, we ended up simply ignoring all of the other header lines. We also require the HTTP client to use version 1.1, or else we will send a 505 error.

All of our files are served from a local webroot folder. Any request that ends with a "/" is assumed to be referring to an index page, and so "index.html" will be appended. If the requested file does not exist, we will send a 404 back to the client, and if the request path is badly formatted, we will send a 400 error. Otherwise, we will serve a status 200.

The 2nd step is to construct the HTTP response. All response messages will contain a Server and Date field. The two special cases are 404 and 200 responses. For 404, a special hardcoded error page will be sent along with the header (this is done because Mozilla Firefox doesn't display to the user what error was returned, unless content is also sent), and of course, for 200, the requested file contents will be served. Serving actual files requires adding more fields to the response header: specifically, the Last-Modified time of the file, its Content-Length, and its Content-Type. The webserver collects these information and sends them in the response, before finally closing the TCP connection.

**Project Difficulties/Solutions**

There were two parts of the project that were particularly difficult to sort out. The first was the process by which we needed to parse the request header. Because we cannot assume that the entire request can be extracted from a single read(), it was necessary for us to maintain an input buffer and keep track of each line in the header while we look for the empty line signaling the end-of-header. We ended up using a static buffer to store the entire request, and at every line break we would keep a pointer to that line. If the parsing is successful, the result should be an array of pointers to each line in the header. As mentioned above, to ease implementation of the reading we limited the request size to be 4096 bytes and 32 lines. This is an arbitrary limit, but any requests with headers larger than this should be uncommon enough that we wouldn't need to worry about them (recall that since we don't support POST, we can ignore anything after the header of an HTTP request).

The second difficult part was in actually generating the response. Serving a file requires us to find some way of collecting information about that file in order to insert them into the response. Modified time and file size can be relatively easily obtained through the Linux stat() system call. Figuring out the content type, however, was more troublesome. What we ended up doing was mapping the file extension to a particular MIME type. For instance, a file that has a ".html" extension will be mapped to the "text/html" content type, while a “.xml” extension meant that the content type should be “application/xml.” Because it was unclear what content types we need to support, we were forced to create mappings to many different types of popular file formats, a time-consuming process.

**Manual on Compiling and Running Source Code**

To compile the code, a simple "make" command is all that is needed. However, there is also the option to "make verbose", which will compile the webserver in such a way that all connection attempts, request headers, and responses will be copied to standard output.

To run the webserver, simply enter the command "./webserver PORTNUM", where PORTNUM is whatever port you want the server to listen on. Once the webserver is running, start up a web browser such as Firefox and connect by entering "localhost:PORTNUM" into the URL field. You should be able to see an index page. It is also possible via command line to create a connection using something such as telnet or netcat, and then manually inputting an HTTP GET request. To include your own files, simply place them into the webroot directory.

**Sample Test Results**

As mentioned, using verbose allows the webserver to echo all requests and responses to standard output. Let's see what happens when Firefox makes a request to the server at port 10000:

cs118@ubuntu:~/vm-shared/project1$ ./webserver 10000

--------- Received connection from: 127.0.0.1 ---------

--------- Request data: ---------

GET / HTTP/1.1

Host: localhost:10000

User-Agent: Mozilla/5.0 (X11; Linux i686; rv:7.0.1) Gecko/20100101 Firefox/7.0.1Accept: 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

Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Connection: keep-alive

--------- Response data: --------

HTTP/1.1 200 OK

Server: Webserver(118)

Date: Fri, 31 Oct 2014 03:49:06 GMT

Last-Modified: Mon, 27 Oct 2014 06:14:15 GMT

Content-Type: text/html

Content-Length: 303

<!DOCTYPE html>

<html>

<head>

<title>CS 118 Webserver</title>

</head>

<body>

You reached the index page.<br><br>

<img src="whoathar.jpg" width="480" height="318"></img>

<br><br>

Where do you want to go?<br>

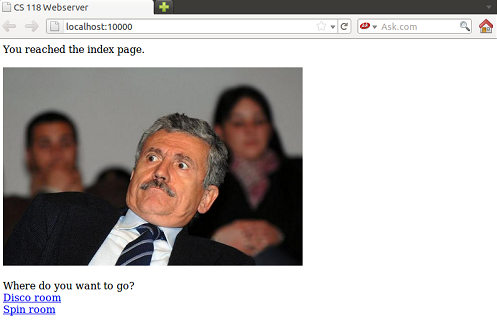
<a href="disco/">Disco room</a><br>

<a href="spin/">Spin room</a>

</body>

</html>

We can see the request headers that Firefox sent as well as the response from the webserver. Immediately after this response, Firefox also sends a request for whoathar.jpg, which the server also returns. The resulting display from Firefox:



As another example, we can try connecting to the webserver via telnet and request the disco page:

cs118@ubuntu:~/vm-shared/project1$ telnet localhost 10000

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^]'.

GET /disco/ HTTP/1.1

HTTP/1.1 200 OK

Server: Webserver(118)

Date: Fri, 31 Oct 2014 04:10:00 GMT

Last-Modified: Mon, 27 Oct 2014 05:36:18 GMT

Content-Type: text/html

Content-Length: 415

<!DOCTYPE html>

<html>

<head>

<title>Disco room</title>

<link rel=stylesheet type="text/css" href="styles.css">

<script src="disco.js"></script>

</head>

<body>

<div class="party" onclick="party()">

<span style="color: red">P</span>

<span style="color: blue">a</span>

<span style="color: green">r</span>

<span style="color: pink">t</span>

<span style="color: orange">y</span>

</div>

</body>

</html>

Connection closed by foreign host.

Notice that the server immediately terminates the connection after serving the file, because we don't support persistent HTTP. As a last example, we can completely automate resource retrieval via the nc program, as long we supply the right request line:

cs118@ubuntu:~/vm-shared/project1$ echo -e "GET /disco/styles.css HTTP/1.1\r\n\r\n" | nc localhost 10000

HTTP/1.1 200 OK

Server: Webserver(118)

Date: Fri, 31 Oct 2014 04:17:58 GMT

Last-Modified: Mon, 27 Oct 2014 05:36:18 GMT

Content-Type: text/css

Content-Length: 37

.party {

font-weight: bold;

}

Feel free to try out our premade disco page and spin page, which tests javascript/css functionality and HTML5 audio playback functionality, respectively.