CS 118 (Professor Songwu Lu)

Project 1: Concurrent Web Server Using BSD Sockets

Fall 2013

Zach North
(603 885 768)

Rory Snively
(803 889 336)

For this project, we created a simple web-server. This server followed the HTTP protocol to retrieve files located in its file system. By processing the HTTP request message sent to it by a client (in this case, our machine's Internet browser) and sending an appropriate HTTP response message, our server was able to deliver information to the requestor.

I. High-Level Description of Server Design

At the highest level, before doing any serving of files, the server must establish a connection with the machine's sockets to be able to interface with the network. Once the server is capable of communication with the network over a given port number, it has one job: to await – and respond to – requests from clients.

Each request was handled individually by the server – through spawning a new process for every connection attempt. For every request, the server parsed the HTTP request message for the desired file to be served. If the file could be found in the server's file system, it was written back through the socket to the client (in an appropriate HTTP response format). Otherwise, an HTTP message was generated alerting the client that the file could not be found.

The code is all contained in a single file (server.c). It is fairly well-documented if the reader is unsatisfied with this explanation.

II. Difficulties Faced

While the process of creating a server seems very straightforward, there are a lot of ins-and-outs of the programming that are very confusing. Many of the low level C functions dealing with strings, file descriptors, sockets, and processes are extremely particular. However, a refresher on multiple C programming websites sufficed to provide us with the tools we needed.

Another roadblock we faced in implementing our web server was understanding exactly how to format HTTP responses. In particular, figuring out the format for images was challenging. It took a while to get the body to be of exactly the right length to make things work.

III. Compiling and Running Source Code

After unzipping the project directory, the code can be compiled by running

```
$ make
```

To start running the web service at a given port number (it is recommended not to use ports 0 through 1024), type

```
$ ./server <port_number>
```

for example,

```
$ ./server 8000
```

This starts a process with the server running in the background. Now, if you open your browser to the port number, and supply a file, the server will attempt to retrieve it. For example, pointing your browser to

```
http://localhost:8000/index.html
```

should dump the HTTP request message generated to the console. Furthermore, the server will look in the directory in which it is running for a file named "index.html". If such a file exists in your directory, then its contents will be served to the client, and appear in the browser. If the file does not exist, then a 404 message will be shown instead.

IV. Sample Outputs

1 – A simple html file

Suppose we have an html file named "`hello.html`" in the same directory as our server code with the following contents:

```
<html>
<head>
     <title>hello</title>
</head>
<body>
     <h1>hello world</h1>
</body>
</html>
```

Then when we point our browser to `http://localhost:8080/hello.html`, our web server dumps the following request message to the console:

```
HTTP Request Message:
```

```
     GET /hello.html HTTP/1.1
     Host: localhost:8000
     User-Agent: Mozilla/5.0 (X11;  Linux i686; rv:7.0.1)
Gecko /20100101 Firefox/7.0.1
     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
     Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
     Connection: keep-alive
```

Our server then generates an HTTP response message containing the contents of the `hello.html` file and sends them back to the client. Thus we can view the contents on our browser.

2 – A non-existent file

Suppose the file `hello.html` was not located in our current directory, and we point our browser to the same location. The same request message is still generated and dumped to the console. However, since the file is not found, a different response is generated, namely the one containing a 404 message. This message can be seen in the `create_404_response()` function located in `server.c`.