Concurrent Web Server Using BSD Sockets

Computer Science 118
Spring 2014

Nathan Tung
004-059-195
SEASnet Login: nathant

Mark Iskandar
704-050-889
SEASnet Login: iskandar

Task Description

The purpose of this project is to utilize socket programming via TCP to create a server that can receive and process HTTP request messages from the client browser. The server then sends back an HTTP response message to the client browser with appropriate status codes and headers, along with the data of the file requested.

We are using Linux OS (Ubuntu) to develop the server. Firefox runs on the same local machine and acts as a client. Therefore, in order for our client to reach the server given some port number, we use the URL:

```
127.0.0.1:port
```

A file can be accessed from the server by requesting it at the end of the URL, as in:

```
127.0.0.1:port/file.html
```

High Level Design

The web server we constructed is based off of the skeleton code from Beej's Guide to Network Programming. The port number that our server uses is defined early on in the code; by default, our server uses a port number of 2080.

When the server is run, it listens for any clients that are attempting to connect to the socket with the correct port number. When the user successfully connects Firefox to the server, the server accepts the connection and uses another socket to facilitate TCP data transmission. This second socket contains the client's HTTP request. For our convenience, this HTTP request message is dumped into the console for analysis.

For each client's request, a child process is forked to handle the HTTP messages and serve the file to the client. In general, the child process does two things: it dumps the HTTP request message into the console (and grabs the name of the requested file in the meantime) and serves that file to the client browser. A summary snippet of the code is shown below.

```
if (!fork()) { // this is the child process
      close(sockfd); // child doesn't need the listener

      char *filename;
      filename = requestDump(new_fd); // dump request message, fetch filename
      serveFile(new_fd, filename); // serve requested file to client

      close(new_fd);
      exit(0);
}
```

Notice that `serveFile` is a very thorough method. First it checks to make sure that the filename is valid or that a file was even requested. If not, a 404 status header is sent to the client and a "File Not Found" page is displayed in the browser. Otherwise, it uses `fopen` and `fread` to convert the requested file's data into a c string buffer, `source`.

Before the file itself is transmitted, a method called `generateResponseMessage`, responsible for building a status-successful HTTP response message, analyzes the file, including last-modified date, content-length (or length of the `source` string buffer), and content-type. The content-type of the file is determined by checking for substrings of possible extensions. This web server accepts html, txt, jpeg, jpg, and gif files. Any other file is assumed to be some form of "txt"-like plaintext. Like the request message, this HTTP response message is transmitted to the client via the `send` method and optionally dumped into the console for the server to analysis.

Finally, the file's `source` buffer is also transmitted via `send` to the client. At this point, the requested file has been served to the client.

The server continues to accept more HTTP requests from multiple clients until forcefully closed. For more implementation details, read the code comments. For visual examples, read the following sections in this report.

Difficulties Faced

For the most part, coming up with the design for `webserver.c` was not especially difficult – the idea of using socket programming to allow client-server communication is relatively straightforward. However, the actual execution in terms of coding the server created some challenges. For example, using c string buffers to put together the HTTP headers proved to be somewhat challenging – even the smallest inconsistency or error in the line breaks (that is, `"\r\n"`) not only ruined the HTTP response message itself, but also corrupted the file's data so that nothing could be correctly transmitted.

Some other minor challenges involved getting used to C syntax or deciding which methods were best for manipulating c strings, and so on.

Compiling and Running the Server

The included `Makefile` means compiling the web server is very easy. One simply needs to type "`make`" (no quotes) into the console in the directory containing `webserver.c`. Otherwise, the console command "`gcc -o webserver.c webserver`" (no quotes) will also compile the program.

To run the server, enter the console command "`./webserver`" (without quotes)

The port number is 2080 by default (although it can be changed in the code). Therefore, to connect the client to the server, open up Firefox and enter the URL:

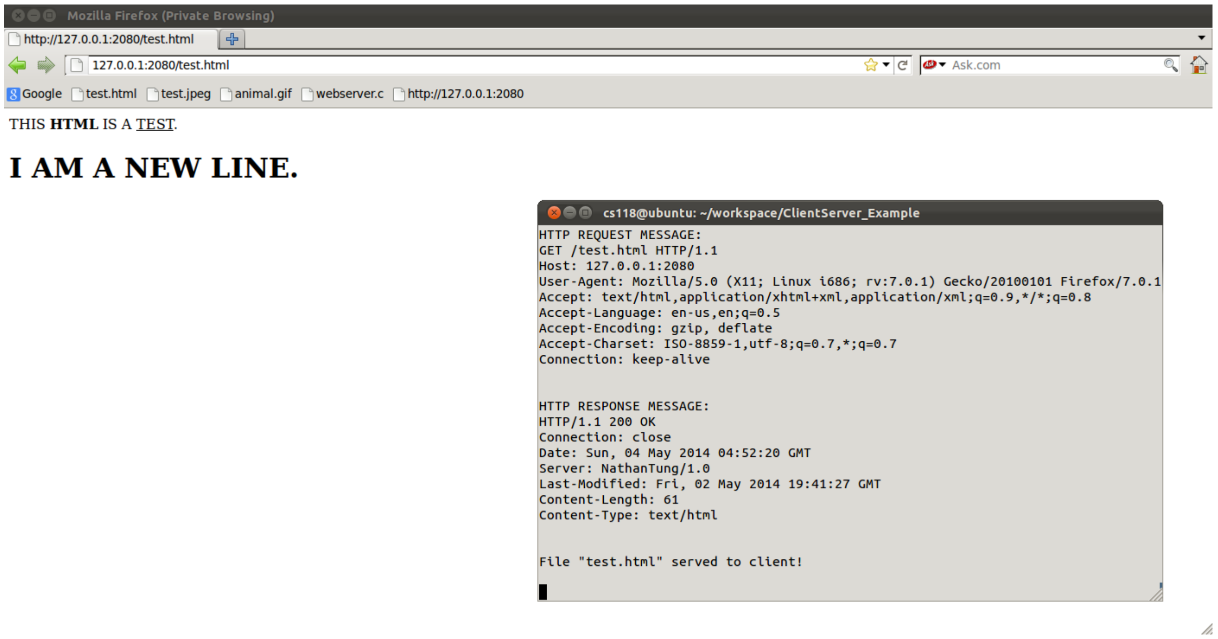
```
127.0.0.1:2080/file.html
```

The specified file in this case is `file.html`. Replace the filename after the slash in order to request other files. Notice the requested files must be in the same directory as the webserver itself.

Sample Outputs of Client-Server

After compiling and running the program with "`make`" and "`./webserver`", we access the localhost URL and attempt to grab the HTML file "`test.html`".

```
http://127.0.0.1:2080/test.html
```


Mozilla Firefox (Private Browsing)
http://127.0.0.1:2080/test.html
127.0.0.1:2080/test.html
Ask.com
Google
test.html
test.jpeg
animal.gif
webserver.c
http://127.0.0.1:2080
cs118@ubuntu: ~/workspace/ClientServer_Example

THIS **HTML** IS A TEST.

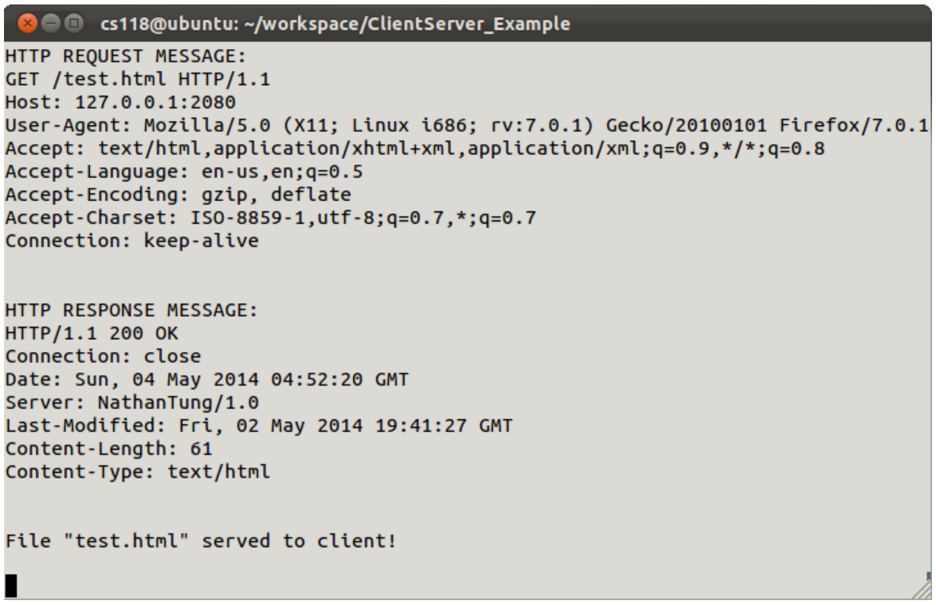
I AM A NEW LINE.

```
HTTP REQUEST MESSAGE:
GET /test.html HTTP/1.1
Host: 127.0.0.1:2080
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


HTTP RESPONSE MESSAGE:
HTTP/1.1 200 OK
Connection: close
Date: Sun, 04 May 2014 04:52:20 GMT
Server: NathanTung/1.0
Last-Modified: Fri, 02 May 2014 19:41:27 GMT
Content-Length: 61
Content-Type: text/html


File "test.html" served to client!
```

Success! The requested .html file was delivered to the browser. Let's take a closer look at the console output. Below, we can see precisely what the HTTP messages are. The request message is what the client sent to the server to request the file (Part A). The response message is the header that the server sent back to the client prefacing the data (and to console for debugging).



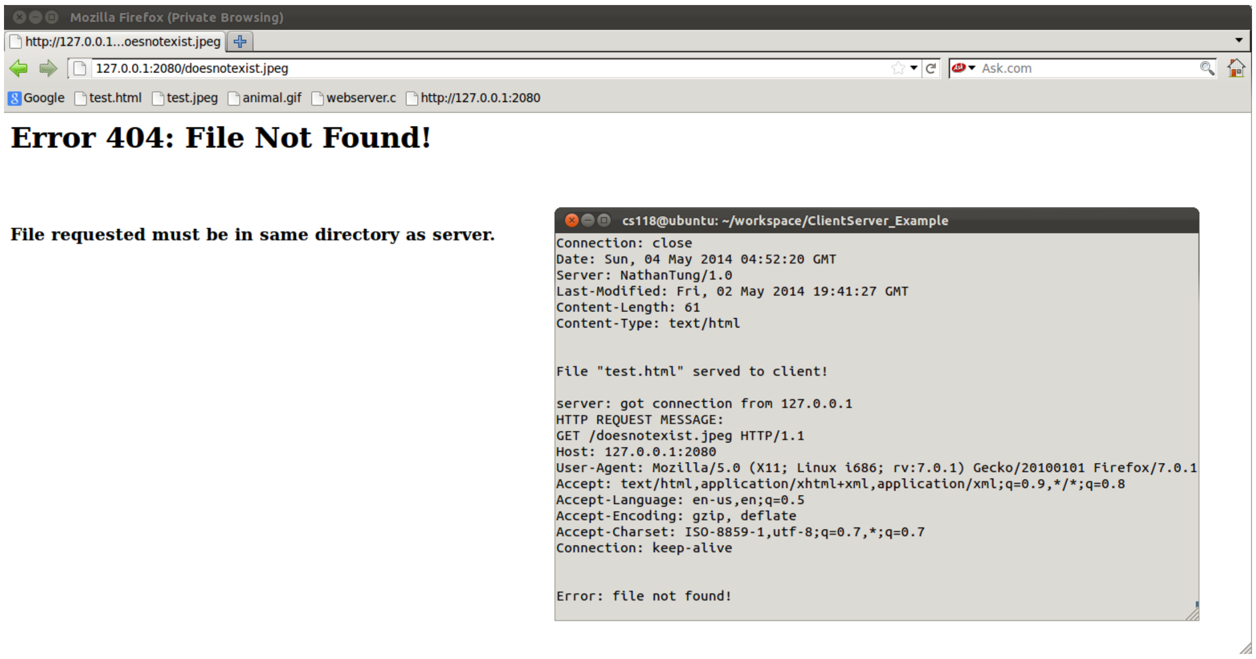
```
HTTP REQUEST MESSAGE:
GET /test.html HTTP/1.1
Host: 127.0.0.1:2080
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:7.0.1) Gecko/20100101 Firefox/7.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive


HTTP RESPONSE MESSAGE:
HTTP/1.1 200 OK
Connection: close
Date: Sun, 04 May 2014 04:52:20 GMT
Server: NathanTung/1.0
Last-Modified: Fri, 02 May 2014 19:41:27 GMT
Content-Length: 61
Content-Type: text/html


File "test.html" served to client!
```

On the other hand, if we request a file that does not exist in our server's directory, such as "`doesnotexist.jpeg`", we are presented with a 404 error message. The response message, although not shown in the console, is "`HTTP/1.1 404 Not Found`".

`http://127.0.0.1:2080/doesnotexist.jpeg`


Mozilla Firefox (Private Browsing)
http://127.0.0.1...oesnotexist.jpeg
127.0.0.1:2080/doesnotexist.jpeg
Ask.com
Google
test.html
test.jpeg
animal.gif
webserver.c
http://127.0.0.1:2080
cs118@ubuntu: ~/workspace/ClientServer_Example

Error 404: File Not Found!

File requested must be in same directory as server.

```
Connection: close
Date: Sun, 04 May 2014 04:52:20 GMT
Server: NathanTung/1.0
Last-Modified: Fri, 02 May 2014 19:41:27 GMT
Content-Length: 61
Content-Type: text/html


File "test.html" served to client!

server: got connection from 127.0.0.1
HTTP REQUEST MESSAGE:
GET /doesnotexist.jpeg HTTP/1.1
Host: 127.0.0.1:2080
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


Error: file not found!
```

Conclusion

Overall, building a concurrent web server via socket programming for Project 1 provided a valuable experience to learn how clients and servers communicate with each other and how data is transmitted between them via TCP. In addition, we got a glimpse of what HTTP messages look like and how they function as requests and responses in the big picture.