CS118 Report

1. Description of the design

This web server is built based on the "Unix Socket Programming" on the tutorialspoint website in addition to TA's server.c sample code. For convenience, user could assign a port number each time we run our server program by passing the port number as an argument.

Basically the server program will build a socket connection according to the port number it received, and listen to the client message. Upon accepting the client request, the server will fork a process, and in the new child process, the server reads the HTTP request from the socket, dumps it on the console (for the visualization of the server function) and then processes the HTTP request.

In order to process the HTTP request, the server will try to parse the HTTP request to obtain the HTTP request method. If the method is "GET", the server will attempt to retrieve the URL in the request message and save it in a variable named "path". **All the files associated with the server should be stored in a folder named "www".** Therefore, the server will insert "www/" as a prefix at the beginning of the URL. If the file exist, the server will generate a 200 OK HTTP response header, and attach the requested file afterwards. If the file doesn't exist, the server will generate a 404 Not Found response message, and print "404 Not Found" in the browser window. Beside the requirement in the project description, when the user does not type the detailed URL of the file, the server will try to redirect to www/index.html. For example, http://127.0.0.1:*port_number/* or http://127.0.0.1:*port_number* the server will redirect to www/index.html.

If the method is some other methods, such as "POST" or "DELETE", our server will generate a 400 Bad Request message, and dump "400 Bad Request" on the browser since our server cannot support the POST method at this moment.

2. Troubleshooting

1) Generating the response message

The first information we need is to extract the method in the request message. At this moment, our server does not support the "POST" method. For convenience, we wrote a read_line function to read the message from the socket line by line, then we use a get_method function to extract the method. If it is "GET", we will call readURL function to read the address of the requested file. (We store all the files related to the server in a "www" folder, so we automatically add "www/" prefix to the URL) . If it is "POST", we generate a "400 Bad Request" message.

2) Passing the file through the socket

The biggest challenge we ran into is how to have the requested file correctly displayed in the browser. Initially what we found is that, as long as we load the file into a string buffer and write the string buffer to the socket, the Firefox browser was able to detect the type of the file and interpret with the right format.
However, later after we composed the complete http response message, we found that Firefox always showed a garbled web page. The reason for that is we passed "text/html" as the content type to Firefox no matter what the file is and firefox will interpret the data based on the file type we provided. Therefore, we implement another function "readFileType" to extract the data type of the file from HTTP request. We changed the formatting of our header part of the message, and most importantly, wrote the content-type information into the socket, such as "Content-Type: xxxx" (text/html or image/jpg, etc).

3) Setting up 404 and 400 response

Another critical problem for us is how to generate the 404 and 400 response message. We have tried two mechanisms: The first one is that, we generate two html files called "400.html" and "404.html" containing the error message, in the case of error, the program will automatically open one of these two files, and send them as a regular html file. One downside is that if the html files are lost or broken, we will have a problem. The other one is to send the error message by "write" or "send" in the program. This approach has the downside that it is less convenient to change the error message.

3. Manual

1) To compile, go to the folder containing the makefile, and type "make" in the terminal. It will compile the source code and generate an executable called "server"
2) To run the server, type " ./server *port_number* " in the terminal. *Port_number* is a number that you can choose.
3) **All the Web page files should be stored in the www/ directory and the www/ directory should be in the same directory as the server.**
4) To send a HTTP request to the server, open the firefox, type" http://127.0.0.1:*port_number/*url of the file
5) The HTTP request and the HTTP response will be dumped in the console

4. Sample output analysis

Note: We only have index.html, picture.jpg, anime.gif in the www/ directory

Case 1:

Input: http://127.0.0.1:5100/anime.gif

Output on the console:

```
HTTP request accepted:

GET /anime.gif HTTP/1.1
Host: 127.0.0.1:5100
```

```
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


sending file...
Response Header:

HTTP/1.1 200 OK
Server: webserver
Content-Length: 863329
Content-Type: image/gif
Connection: keep-alive
```

Analysis:
The first part of the display is the HTTP request sent from the Firefox browser. In the first line, it shows that the browser is using GET method, requesting a file called anime.gif and following HTTP/1.1 protocol. In the second line, it lists the target host IP address and the port number. The third line identify the browser to the server. Then the Accept field indicates which types of the media is acceptable for the response. Then the Accept-Language indicates the preferable natural language the browser prefers. Accept-Encoding restricts the encoding set the server should use. Accept-Charset shows what character sets are acceptable and make some special purpose char more comprehensive. Then the keep-alive in the connection field shows a persistent connection should be established.
In the second part, HTTP/1.1 200 OK indicates the HTTP request was received, accepted, understood, and processed successfully. The Server field indicates the server type and we name the server we implemented as “webserver”. The Content-Length is the length of the gif file. The connection keep-alive indicated a persistent connection has been established.

Case 2:
Input: http://127.0.0.1:5100/s

```
HTTP request accepted:

GET /s HTTP/1.1
Host: 127.0.0.1:5200
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/1.1 404 Not Found
Server: webserver
Content-Length: 64
Content-Type: text/html
Connection: close
```

The first part is almost the same as Case 1. But the second part is different from the Case 1. In the first line of second part, it generates the error code 404, which means the server is unable to locate the file. Besides, the connection is closed, which means the server would terminate the connection when the error occurred.

Case 3:
In order to generate a 400 Bad Request message, we use telnet as the client program.
In the terminal, input: telnet localhost 5200 to open a telnet connection.
Then input:

```
POST /234 HTTP/1.1
```

Response from server:

```
HTTP/1.1 400 Bad Request
Server: webserver
Content-Length: 64
Content-Type: text/html
Connection: close
```

The first part is almost the same as Case 1. But the second part is different from the Case 1. In the first line of second part, it generates the error code 400, which means the server is unable to understand this request. Besides, the connection is closed, which means the server would terminate the connection when the error occurred, which is same as the Case 2.