CS 118 Spring 2018

Computer Network Fundamentals

UCLA

Project 1: Web Server Implementation using BSD Sockets

Ram Yadav (UID: 604841556)

Matthew Sanchez (UID: 204650577)

1 Web Server Design

The basic functionality of our web server involves several core steps. First, we implement a socket by writing a socket system call and set it so that it uses a TCP connection. The system call returns a file descriptor that the web server will use to refer to the socket. Then we obtain the port number from user input (via the command line). We use a special network byte order function called htons() in order to change the port number so that it uses network byte order, allowing the binding of the socket to the IP address and port number. We use the file descriptor for the socket that we created to bind the socket to the address.

Once the web server has established a connection, the server uses the listen() function to await up to a maximum of five connections from web browsers (the clients) that will request the server for a web page. Once a web browser has connected to the server, the web server will cease to block and accept the connection using the accept() function, creating a new file descriptor for the current connection between the server and the web browser client. Finally, the server will read in data sent from the client to the server via the new file descriptor we have created and store the message from the client into a buffer.

Since we are using a web browser as our client, the information sent from the browser and stored in the buffer in the web server will be the HTTP request created by the browser. The first thing the web server will do is print out the HTTP request message. Then, the web server will respond to the browser's HTTP request by parsing through the request to determine which file the browser wants to display as its web page. The server achieves this by examining the first line of the message: the request line. The server will parse through the request line to find the requested file/object, which is placed after the method field (in this project, the message field will be GET).

Our web server will allow requested filenames to contain spaces. Initially, the server will interpret a space character to be a three line string of the form "%20". To allow the correct functionality, the web server will parse through the filename to search for all instances of "%20" and change them to a single space character. In addition, our web server will be case-insensitive, meaning that filenames with the same sequence of letters but with different cases will refer to the same file regardless of the cases within the file names. To achieve this, the web server parses through the filename again, changing all alphabet characters to lowercase characters. Finally, the web server will parse through the filename one last time in order to obtain the correct file extension type. To achieve this, the server will create a new buffer that will store the file extension type. It will parse through the filename searching for a period character (.) and store all characters after the period into the buffer. If no period character is found, it is assumed that the file is a binary file type.

Once the requested filename has been found and parsed for correct functionality, the web server will begin the process of constructing the HTTP response message to the browser. First, the server will attempt to open the file with the specified filename in the server's current directory.

If the server can not open the file, it means that the requested file does not exist in the server's current directory. As a result, the HTTP response message will simply consist of a "404 Not Found" status line, a default content length and HTML type, and a HTML page consisting of a "404 Not Found" message. The server will write the response message line by line, finishing with a empty line (a carriage return and a line feed), then it will close the client socket, ending its connection with the browser and begin awaiting new connections.

If the server can open the file, it means the requested file does indeed exist in the server's current directory. The status line of the HTTP response will be a "200 OK" message. The field for the Content-Length header will be determined by the size of the file. In order to determine the size of the file, the server will use the C API fseek and ftell to parse through the file and count the number of bytes in the file. It will then write the Content-Length header to the browser. Then, looking at the filename extension type the server stored into a buffer, the web server will determine the field for the Content-Type header and write the header to the browser. To complete the HTTP response message, the server writes an empty line to the browser.

Finally, the web server will write the actual contents of the file requested by the browser to the browser via the socket file descriptor that represents the connection between the browser and the web server. The server will read the contents of the file into a file buffer, using the previously obtained file size to know when to stop reading the file. Then, using the socket file descriptor, the server writes the contents in the file buffer to the browser, using the file size to know how many bytes to write to the browser. Once this is finished, the web server will close its connection with the browser and the browser will display the content of the file it has requested.

2 Difficulties

In the process of creating our web server, we encountered two major difficulties and sought various methods to solve them.

The first major difficulty we encountered was determining how to properly display files with filenames that contain spaces. When the browser sends the HTTP request message to the web server, the filename within the request message will contain "%20" in place of space characters. When we initially attempted to convert the three character string "%20" to a single space character, it would do so successfully, but only if the filename had a single space character between other types of characters. If the filename contained 2 or more spaces consecutively, it would only convert the first instance of "%20" to a space character and leave the rest of the "%20" substrings untouched. After experimenting and debugging the code, we figured out that we had to only increment the counter in our for loop that parses through the filename if the current character in the loop was not converted to a single space character. By not incrementing the counter, the parser will not skip over any substrings of "%20" that were moved to the left when the size of the string decreased by 2 (since "%20" gets converted to a single space character).

The second prominent difficulty we faced was figuring how how to read and write back to the browser efficiently. When attempting to read in the data bytes from the requested file, the server originally read in and wrote to the browser one byte at a time. When the browser would attempt to display large image files, it would take a significant amount of time for the web page to complete loading. In the meantime, the server would not be able to accept connections from other client browsers, so this was clearly an issue. To fix this, we modified our server so that it read in the entire file at once before sending any data/bytes back to the browser. Then, we would write the entire file to the browser with a single write command executed once. This would allow the web page to load instantly, even in the case of large image files.

3. Compile and Run the Source Code

We have only one source code file called webserver.c written in c programming language for the server programming. Client will be the browser. In our case, we have performed all the test cases on the “Mozilla Firefox”. We have tested on the other browser as well, but we encountered problem running our program there. So please use “Mozilla Firefox” while testing to perform accordingly as spec suggested.

We have put the connection ‘accept socket connection’ in a while loop. Therefore, once you run the server you don’t have to run again to perform other test cases. To run the server, you need to specify the port number, otherwise program wouldn’t run. But before run the server, please make an executable file and run using the format ./[Executable webserver.c file] [Port number]. Doing it will enable you to run the server. Once you started the server, please run the client using the same port number you used for the server in the format localhost:[port number that you use for webserver]/[file name]. Please use Firefox Mozilla as a client.

When you request a file, if the file exists on the current dictionary to the webserver.c, it would display the output to the browser. For example, if the file ‘html, jpeg, txt etc’ requested using the browser and the file is in the current dictionary to webserver, it would display the result of html page, jpeg picture and texts to the browser respectively. If the file is not presented in the dictionary, it would give an error of “404 Not Found”. For the binary file, the browser would prompt to download, and once you download the file, you will able to see its content. We came to the conclusion that, getting pop up for the download box is one of the firefox functionality, if you don’t want to get the box popping up for the download when get the binary file, you can change it from the browser menu. Our program support the file (i.e. *.html, *.htm, and *.txt), JPEG and GIF images (i.e. *.jpg, *.jpeg, and *.gif) as spec mentioned. Beside this file, any other file’s output will be shown as “404 Not Found”.

Our program is also a space sensitive. Hence, a file name must match exactly with the file(including space of the file) that is presented in the dictionary to the webserver.c. If file doesn't match, the client "Mozilla Firefox" will shows "404 Not Found" as an output.

4. Output

As we have mentioned in the part 3, when a client (Mozilla Firefox) request a file to the server, client would able to see the file's output in the browser. If a file has not been presented in the current dictionary to the webserver.c, it would give an output of "404 Not Found". We have commented our source code throughout the file as well that may help you to easily understand the code.

For the part A, the output will display in the browser and connection details will be shown in the console. As we mentioned in the part 3, when client request file HTML, jpeg or txt, they will able to see the html doc, jpeg photo, and text files in the browser. If file happens to be not presented in the current dictionary to the webserver.c, browser will display "404 Not Found" as an output error.