Raymond Lin, Frank Ren
304937942, 904826309
CS118 - Computer Network Fundamentals
Songwu Lu
26 April, 2019

Project 1: Web Server Implementation and BSD Sockets - Report

1. High Level Description

   This project can broken down into 3 parts. The first part is setting up socket and binding the socket to a specific part, in this case we bind the socket to port 6555. The second part of the server is to be constantly accepting new connections. We do this by running accept() in a while loop that keeps looping. We then read the message from the new connection into a buffer. We then print out the buffer to the terminal, which is the html request message. The third part, and the most complicated part is to analyze the html request message and send a proper response message.

   We first process the request message, turning %20 into spaces and turning Upper Case letters to lowercase and etc. We then get the file name and the file type while we are processing the request message. After getting the file name and file type, we then open the file that's indicated by the file name, and we load the file into a buffer. We then get all the characteristics of the file such as last modified time, the content size and etc. After we got all the information we need, we wrote to the client a response message as well as the requested file, and we then closed the connection.

2. Difficulties and Solutions

   First, we had some issues with sending the correct response header to the client browser. On Firefox in Ubuntu, we couldn't figure out why some of the response header entries were incorrect, or why the response headers were not being shown in the response headers section in the web developer console. In order to solve the problem, we did some string manipulations, such as correcting formatting and fixing line endings. Then the response headers showed up in the correct area in the browser.

   In particular, we had the most issues with the 'content-length' header line. At first, whenever we tried to download the large binary file, it would only download a small portion of it, and in GDB, it showed that the 'read' was returning 0. We thought that there was something wrong with the socket and TCP connection. However, we later realized that the header line would not show the correct file size. This was because we didn't assign the correct number of characters to the C-string that would store the digits of the long value that contained the file size in bytes. As a result, the program was truncating the file size and only returning 1MB out of the 100MB of the large binary file.

   We occasionally ran into a number of other smaller typical C programming issues such as memory management and segmentation faults. We solved these issues by carefully examining the code and running debugging software, such as 'valgrind' to

identify segmentation faults and ‘gdb’ for general debugging, to identify the bugs.

3. Manual
    a. $make
       This compiles the C program into ‘webserver’
    b. $./webserver
       This runs the server
    c. Open any browser
    d. In the address bar, type:
       ‘localhost:6555/<filename>’
       Our specified port number is 6555

4. Sample Outputs and Explanation
   -Below are the different test cases

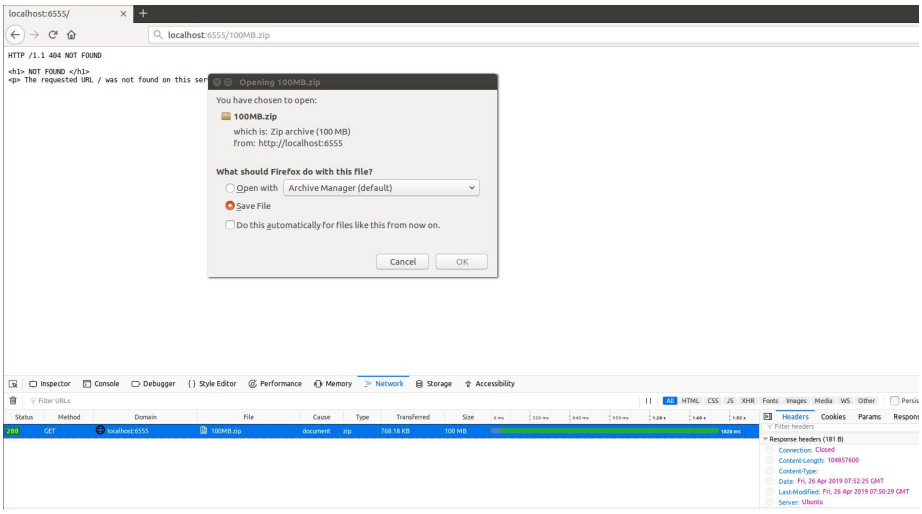


Figure 1: Test case for large binary file. After client asks for large binary file, the server responds with the corresponding binary file. The response message can be seen from the bottom right corner. The response status is 200, meaning the the request is successful.We then were asked to download the file. In this case, we downloaded and saved the file as 100MB_test.zip. We then ran diff command on the received binary file and the original binary as shown below to prove the validity of the html response. Noticed that the client’s GET message is printed in the terminal below!

```
ubuntu@ubuntu:~/CS118-Project1$ ./webserver
GET /100MB.zip HTTP/1.1
Host: localhost:6555
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:65.0) Gecko/20100101 Fire
fox/65.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0
.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1

^C
ubuntu@ubuntu:~/CS118-Project1$ ls
100MB_test.zip       file.txt       README.md     ucla.jpg
100MB.zip            giphy (1).gif  small_binary  Vagrantfile
canvas.png           giphy.gif      test.html     webserver
file_downloaded.txt  Makefile       ucla.jpeg     webserver.c
ubuntu@ubuntu:~/CS118-Project1$ diff 100MB.zip 100MB_test.zip
ubuntu@ubuntu:~/CS118-Project1$
```

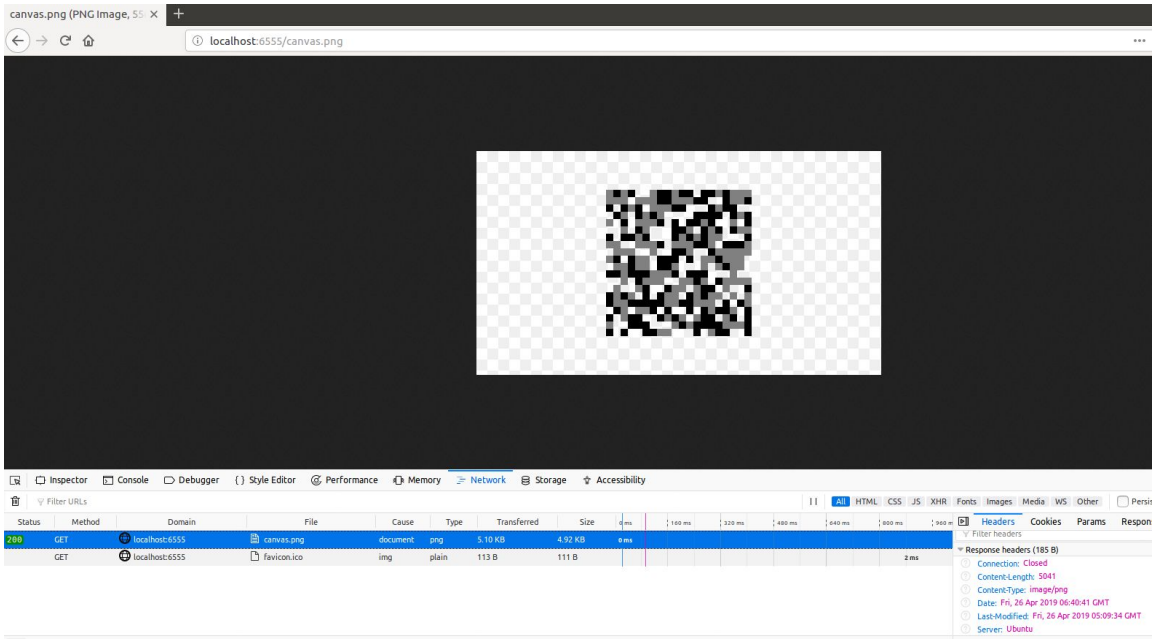


Figure 2: test case for png files. The response message can be seen from the bottom right corner. The response status is 200, meaning the the request is successful.

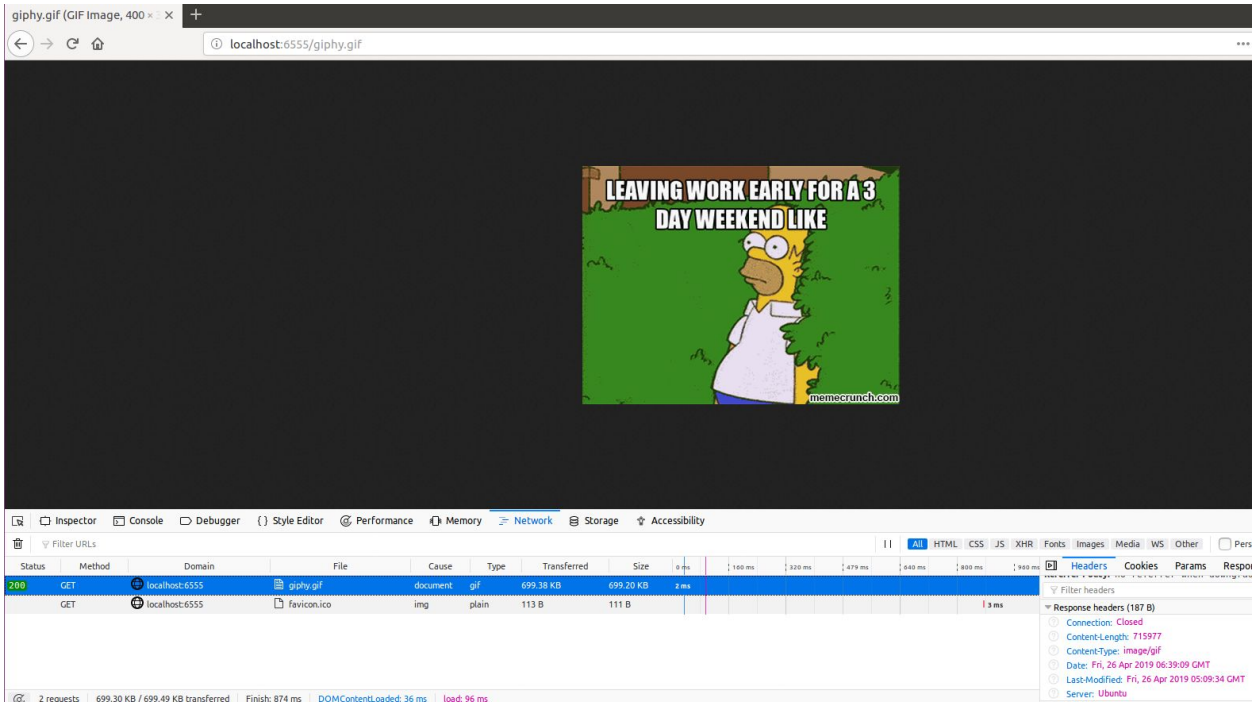


Figure 3: Test case for gif file. The response message can be seen from the bottom right corner. The response status is 200, meaning the the request is successful.

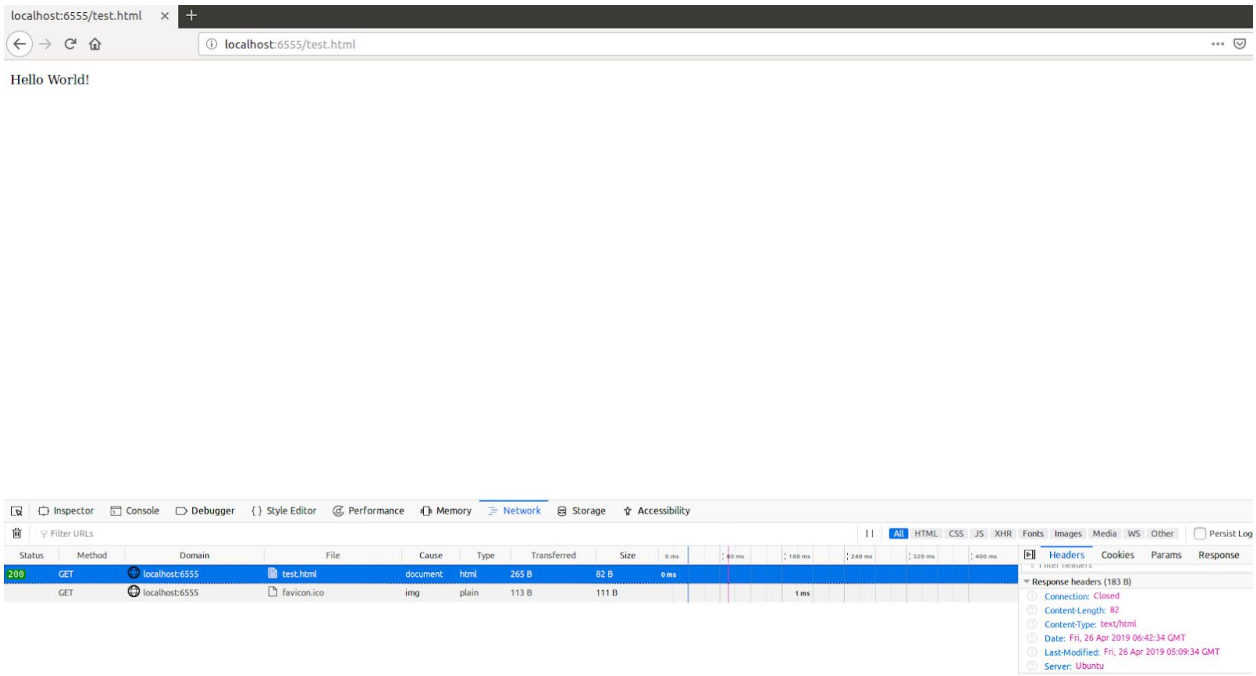


Figure 4: Test case for a custom made html file. The response message can be seen from the bottom right corner. The response status is 200, meaning the the request is successful.

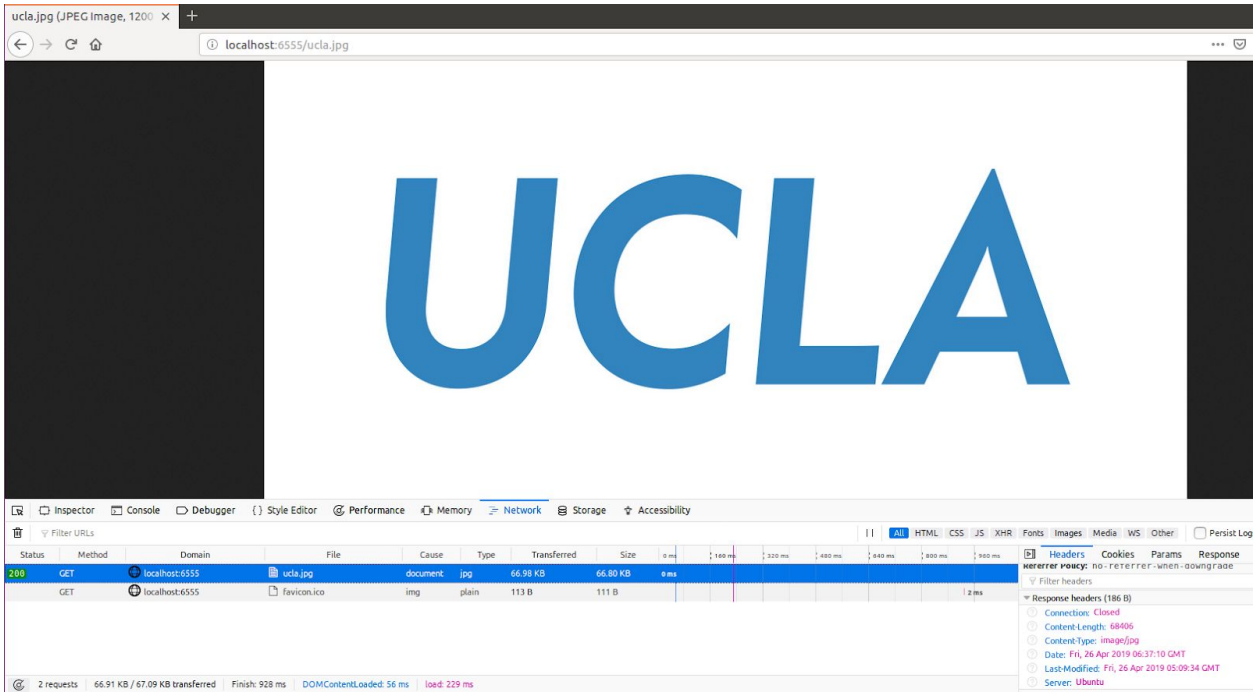


Figure 5: Test case for jpg file, which is very similar to jpeg file. The response message can be seen from the bottom right corner. The response status is 200 -> the the request is successful.

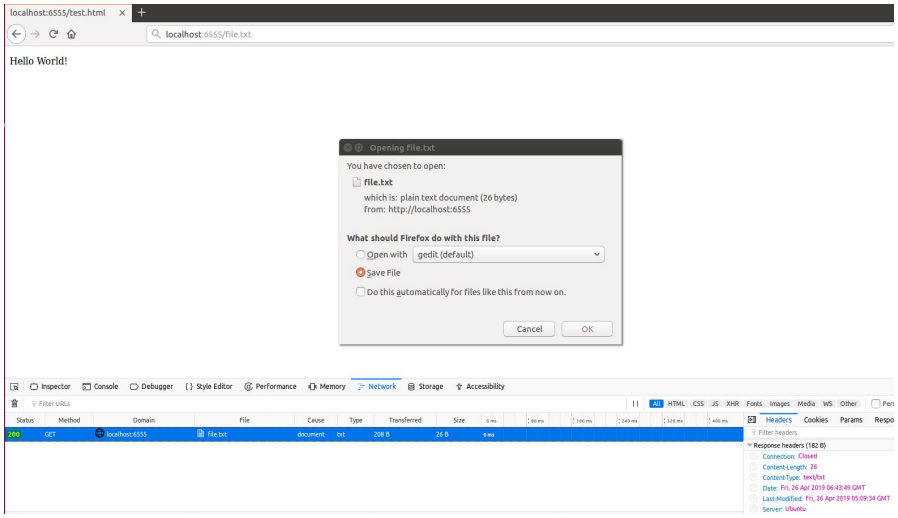
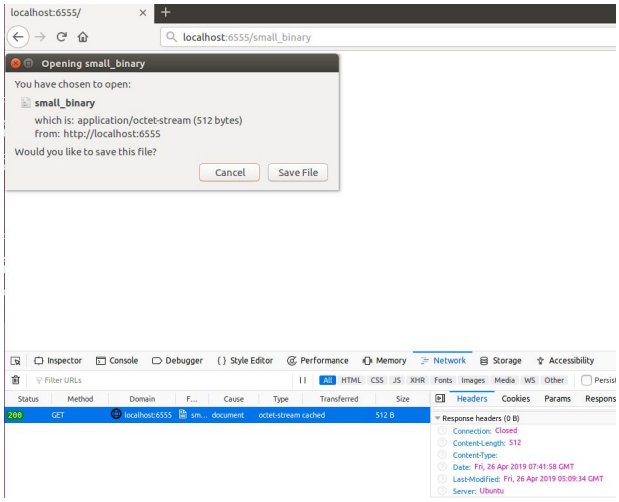


Figure 6: Test case for text file. After client asks for txt file, the server responds with the corresponding text file. We then were asked to download the file. In this case, we downloaded the file and saved as file_test.txt. We then ran diff command on the received text file and the original text as shown below to prove the validity of the html response.

```
ubuntu@ubuntu:~/CS118-Project1$ ls
canvas.png      giphy (1).gif  README.md      ucla.jpeg     webserver
file_test.txt   giphy.gif      small_binary   ucla.jpg      webserver.c
file.txt        Makefile       test.html      Vagrantfile
ubuntu@ubuntu:~/CS118-Project1$ diff file.txt file_test.txt
ubuntu@ubuntu:~/CS118-Project1$
```



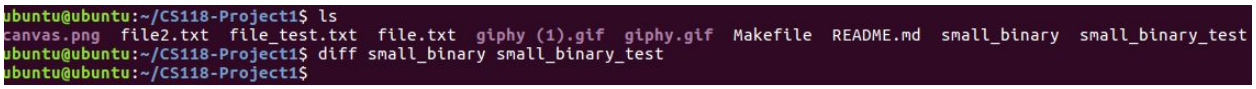


Figure 7: Test case for small binary file. After client asks for the binary file, the server responds with the corresponding binary file. We then were asked to download the file. In this case, we downloaded the file and saved as small_binary_test. We then ran diff command on the received binary file and the original binary file as shown above to prove the validity of the html response.