CS 118 PROJECT 1 REPORT

Sahil Gandhi and
Arpit Jasapara

CS 118 / WINTER 2018 UIDs: 704-598-105 and 504-742-401

1) High-level Description of Server Design:

Our server uses sockets to bind a connection with incoming client requests. When a client requests to connect with the server, it accepts the connection and spawns a child process for the client. The child process reads in the GET request, outputs it to console, parses the file name and file type, opens and reads in the file's data and metadata using fstream and fstat, and appropriately formats it into an HTTP response. The HTTP response is then output to console and sent back to the client to be downloaded or displayed appropriately within the client's browser. As noted, we output both the received GET request, and the created HTTP response to the console so that the server user can verify that they are receiving/sending the correct data, and can see exactly how the two objects (server, browser) are communicating with each other.

We also perform some sanitation function in our program like being able to parse out %20 to handle files that contain spaces (the browser encodes spaces as %20), added some extra file types to support, and made sure that everything that is not readily showable by the browser is downloaded!

We are using port 5000 for the server, so the browser must connect to this port on the localhost in order to send requests and receive files.

2) Difficulties that we faced and how we solved them:

One major difficulty that we faced was how to open the file and get its data. We could use the C open method or use C++ fstream, since both have their advantages and disadvantages. We ultimately ended up using both, where we used the C open method in combination with fstat to quickly obtain the file's metadata, and fstream to actually read in the file's data efficiently.

Another difficulty that we faced was parsing the GET request so that we could only get the file name and not extraneous data. We were going to initially make a regex query to grep the file name, but instead ended up relying on the built in find command to chop up the string into certain pieces and get the substring that corresponded to the file name.

The final main difficulty that we had was sending the response. We often were missing particular line endings (like the \r\n) or had mistyped names or other formatting errors, and thus used the console heavily to debug our program. We also relied greatly on the book and the PowerPoints to make sure that our responses contained the appropriate fields that allowed the browser to handle the sent file properly.

3) Manual to compile and run the source code:

Run “make” to compile the program. Then run “./p1_server” to have the server up and running. Attached are screenshots of how to do this:

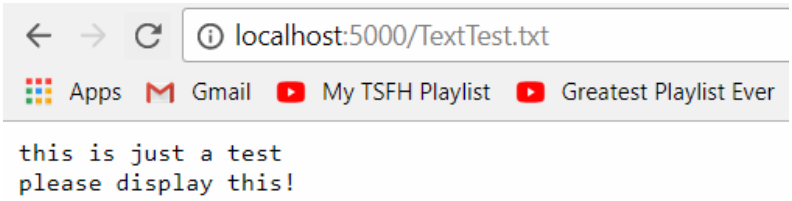
Making program:

```
sahilmgandhi:CS118_Project1$ make
g++ -Wall -Wextra -g -o p1_server -I. server.cpp
```

Running program:

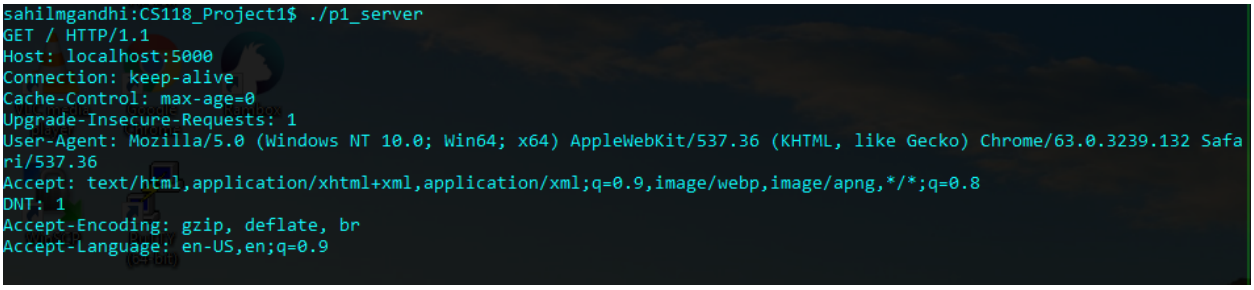
```
sahilmgandhi:CS118_Project1$ ./p1_server
```

Once the program is running, we can then connect to it by using a browser (like Google Chrome, or Firefox), and connecting to localhost:5000. By default it will show a blank page (since it is not requesting a file), and so to request a file, we merely have to do localhost:5000/${file_Name}, such as localhost:5000/TestText.txt



4) Sample outputs of client-server (Part A and Part B)

A) Part A sample response(s)



As we can see from this image, the server outputs the GET request from the browser successfully. The different fields that we see mean:

Line 1: The type of request, in this case GET, followed by a path, and then that it is HTTP protocol 1.1

Line 2: The host (server) being contacted. Localhost:5000 in our case

Line 3: Whether to keep the connection alive (default is alive in modern browsers since they use persistent channels).

Line 4: Cache-control max-age makes sure that caches do not interfere with the request or response. Max-age=0 means only get fresh copies, no stale ones.

Line 5: Upgrade-insecure-requests means server should send an encrypted, authenticated response and to be able to upgrade insecure requests if needed.

Line 6: The user agent line tells us that it is Mozilla compatible, the platform you are on and that we are not on a phone, but a laptop. Then it also says the gecko version and the chrome version that we are using.

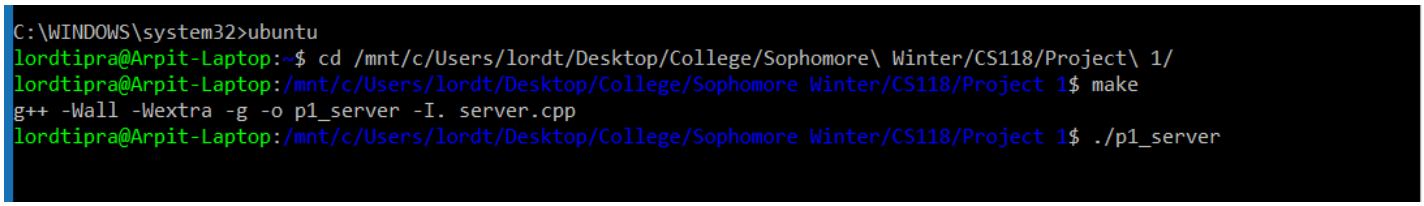
Line 7: Tells us what kinds of things our browser can accept.

Line 8: DNT 1 is a message saying to not track on the current site

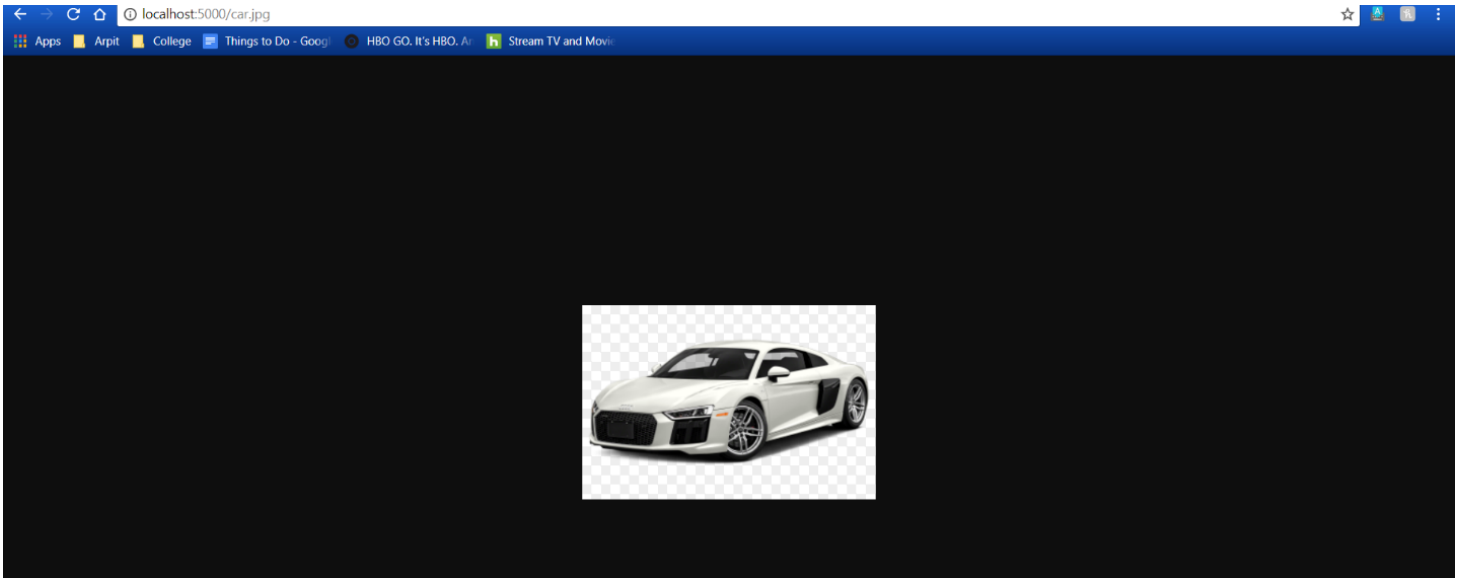
Line 9: Accept-encoding tells what kind of encoding the content can be in for the browser to understand

Line 10: Accept-language tells what kind of language the client can understand.

B) Part B sample response(s)



As we can see from this image, we made our project and ran the server.



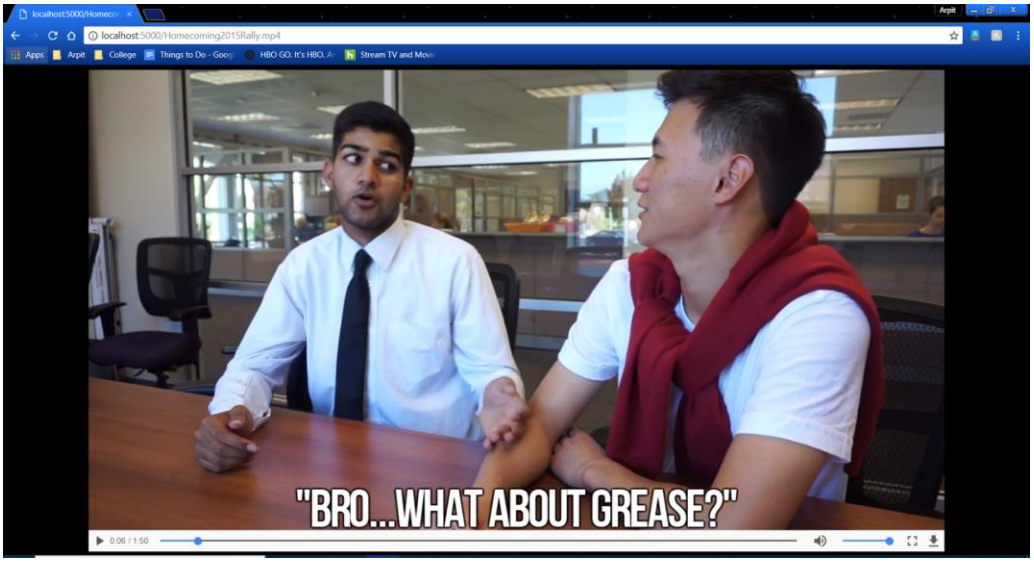
As we can see from this image, the GET request for a .jpg file (car.jpg) successfully shows up in the browser due to the HTTP response.

```
C:\WINDOWS\system32>ubuntu
lordtipra@Arpit-Laptop:~$ cd /mnt/c/Users/lordt/Desktop/College/Sophomore\ Winter/CS118/Project\ 1/
lordtipra@Arpit-Laptop:/mnt/c/Users/lordt/Desktop/College/Sophomore Winter/CS118/Project 1$ make
g++ -Wall -Wextra -g -o p1_server -I. server.cpp
lordtipra@Arpit-Laptop:/mnt/c/Users/lordt/Desktop/College/Sophomore Winter/CS118/Project 1$ ./p1_server
GET /car.jpg HTTP/1.1
Host: localhost:5000
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,es;q=0.8,lv;q=0.7
If-Modified-Since: Fri, 02 Feb 2018 00:28:55 GMT


HTTP/1.1 200 OK
Date: Fri, 02 Feb 2018 04:54:36 GMT
Server: Gandhi-Jasapara Server
Last-Modified: Fri, 02 Feb 2018 00:28:55 GMT
Content-length: 60681
Content-Type: image/jpeg
Content-Disposition: inline
Connection: close


Child exited successfully with code 17. Reaped child process.
```

As we can see from this image, we successfully received the GET request for the car.jpg, processed it, and sent back an appropriate HTTP response.



As you can see from these images, we added functionality for .mp4 and several other file types (.pdf, .wmv, etc.) to make our HTTP web server more robust. We successfully received the GET request and were able to display the video in-browser. Moreover, we sent back the appropriate HTTP response, and reaped the child process thereafter.

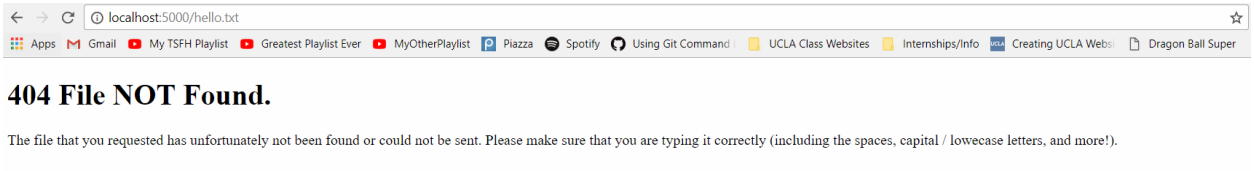
```
lordtipra@Arpit-Laptop: /mnt/c/Users/lordt/Desktop/College/Sophomore Winter/CS118/Project 1
Connection: close


Child exited successfully with code 17. Reaped child process.
GET /Homecoming2015Rally.mp4 HTTP/1.1
Host: localhost:5000
Connection: keep-alive
Accept-Encoding: identity;q=1, *;q=0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36
Accept: */*
Referer: http://localhost:5000/Homecoming2015Rally.mp4
Accept-Language: en-US,en;q=0.9,es;q=0.8,lv;q=0.7
Range: bytes=0-


HTTP/1.1 200 OK
Date: Fri, 02 Feb 2018 04:57:18 GMT
Server: Gandhi-Jasapara Server
Last-Modified: Fri, 16 Oct 2015 00:25:20 GMT
Content-length: 84142658
Content-Type: video/mp4
Content-Disposition: inline
Connection: close


Child exited successfully with code 17. Reaped child process.
```

Furthermore, upon a failed request, we have also implemented the 404 page.



Here are some other things that we can handle (gifs, html, text):