CS 118 Computer Network Fundamentals

Project 1: Concurrent Web Server using BSD Sockets

Name:	Haosheng Zou	Mingrui Shi
ID:	804518325	104517541
SEASnet login:	haosheng	mingrui

2014/10/30

Sever Design Description

Our team designed a simple web server, which has basic functions to parse HTTP request from a browser and make a response back to the browser with the information requested.

To communicate between a web browser (client) and our server, we established a connection with socket programming, in addition, we used the select function to concurrently serve multiple connections from different clients.

After a connection was set up, the server could receive request message from a client and dumped the message to the console, at the same time, request message was parsed such that the server could extract the file's URL requested by the client. We also extracted IP address and port number of the client for further use.

The server will first check the header of request message, if the header is invalid then will directly response 400 Bad Request. After that, The server could check whether the requested file existed in the server system or not and built up different HTTP response message. A HTTP response message is consist of a header and the data body; we constructed the header first and appended the data to it, then sent the whole response to the client. For the header part, we referenced the example request message from the textbook, wrote functions for each header line and constructed them together into an integrated header. For the data part, we read the data form requested file and appended it to the header, the server supported various types of data like html, jpg, mp4, pdf, etc. If the file could not be found in the server, the response message would be 404 not found.

Difficulties and solutions

1. We have used 'while' loop to continuously read and write through the socket, but when the file is larger than the buffer size, we found it didn't work. So we checked the program and found out that the number of chars read from the file should be equal to the buffer size, otherwise it would be a '\0' blended in the data, made the binary file unable to display.
2. When we finished the select function, we didn't know how to test it. We could not set up multiply connection manually. TA Yuanjie told us to try to use a script to run multiple 'wget' functions. We eventually wrote a shell script program which called the 'wget' command several times to test

whether it works or not, and fortunately it works.

3. When trying to get Last-Modified time of the requested file, things got really weird at first. We called *stat()* function to get the last modified time of the file, however the return value of the function was either 1969 or 2107 for Year, which was ridiculous. At last we found out that the FilePath begins with a "\" was invalid for the function *stat()*.
4. At first when we implemented select function to the server, the server did not work. It seemed the program went into a dead loop of the main *while(1)*. The reason was that we used file descriptor *n=0* to signal whether a connection is done. It turned out that *n* would not go to 0 unless the request was just the size of buffer size. We fixed the fault and *select()* function did worked on the server.

Manual on how to compile and run

The project consists of a make file that builds *serverFork.c*. Compile the makefile should create an object file called *serverFork.o.*

$ make

Run the file with a port number (recommended not to use ports 0 to 1024). Take port number 12345 for example.

$ ./serverFork 12345

To setup connection, open a browser using server's IP address and port number to request a file from server. For instance, using localhost and port number 12345 and the request file is *test.html*.

Type in your browser:

http://127.0.0.1:12345/test.html

The request message should be dumped in to the console and the server will look in the directory to find out whether the requested file exists or not. If the request header invalid, a 400 bad request message will be directly send back. If the requested file exists in your server system, it constructs a response message with the requested file and sends it back to the browser, which means the content of the file

will appear on the browser. Otherwise, the server will send a 404 not found message to the browser.

Results and Analysis

A request for *test.html* from browser

To test the server performance I have used three different browsers (FireFox, Safari, and Chrome) to send request to the server. It turned out that there were some difference between browsers when dealing with video files.

In the screenshot below, we can observe the response message from the browser (request for *test.html*). In the test.html, some text ("ni hao"), a picture (Lucy.jpg), a gif (test.gif), and a video (test.mp4) were included. The browser firstly sent a HTTP GET for test.html to the server.

```
^C
cs118@ubuntu:~$ gcc server9.c -o server9
cs118@ubuntu:~$ ./server9 6666
Here is the message:
GET /test.html HTTP/1.1
Host: 127.0.0.1:6666
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
```

Right after the first request above, the browser sent three requests for the picture, gif and video respectively. The three requests are shown as following screenshots.

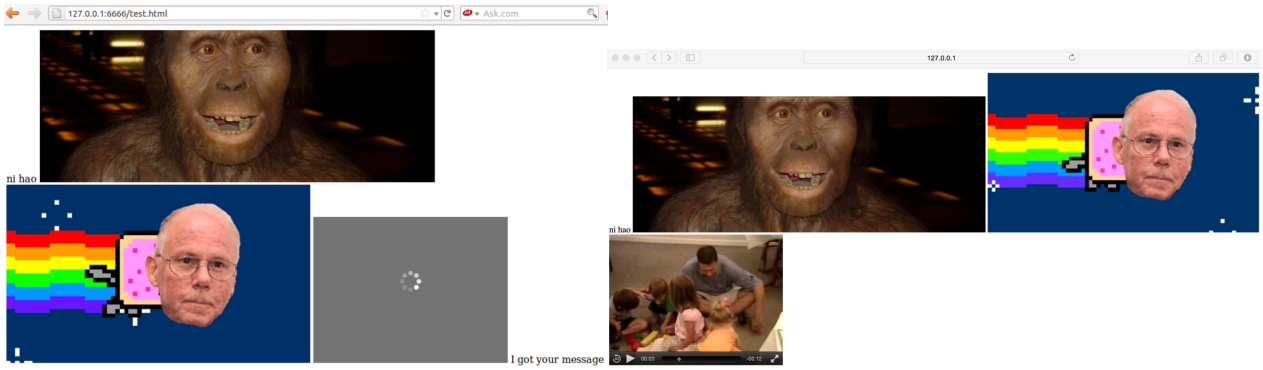
```
Here is the message:
GET /Lucy.jpg HTTP/1.1
Host: 127.0.0.1:6666
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:7.0.1) Gecko/20100101 Firefox/7.0.1
Accept: image/png,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://127.0.0.1:6666/test.html
```

```
Here is the message:
GET /test.gif HTTP/1.1
Host: 127.0.0.1:6666
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:7.0.1) Gecko/20100101 Firefox/7.0.1
Accept: image/png,image/*;q=0.8,*/*;q=0.5
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
Referer: http://127.0.0.1:6666/test.html
```

```
Here is the message:
GET /test.mp4 HTTP/1.1
Host: 127.0.0.1:6666
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
Referer: http://127.0.0.1:6666/test.html
```

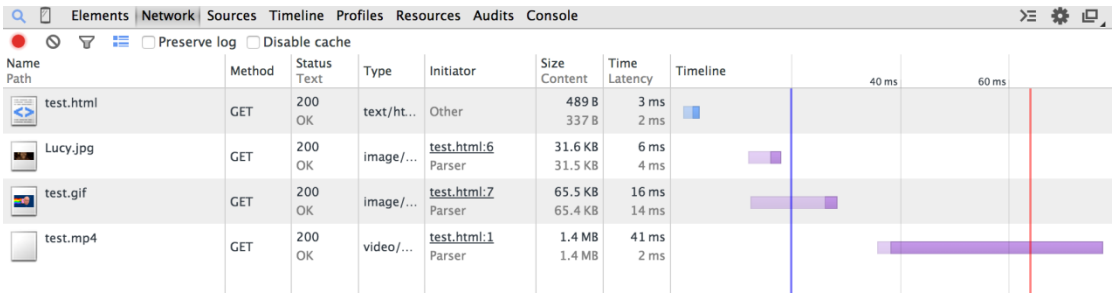
As for the response to the browser, all three browsers could successfully show the text, picture and gif. However, FireFox and Chrome weren't able to load the video, only Safari could play the video.

The following screenshot shows the result on FireFox (left), the video part failed. Safari (right) some how could successfully load and play the video.

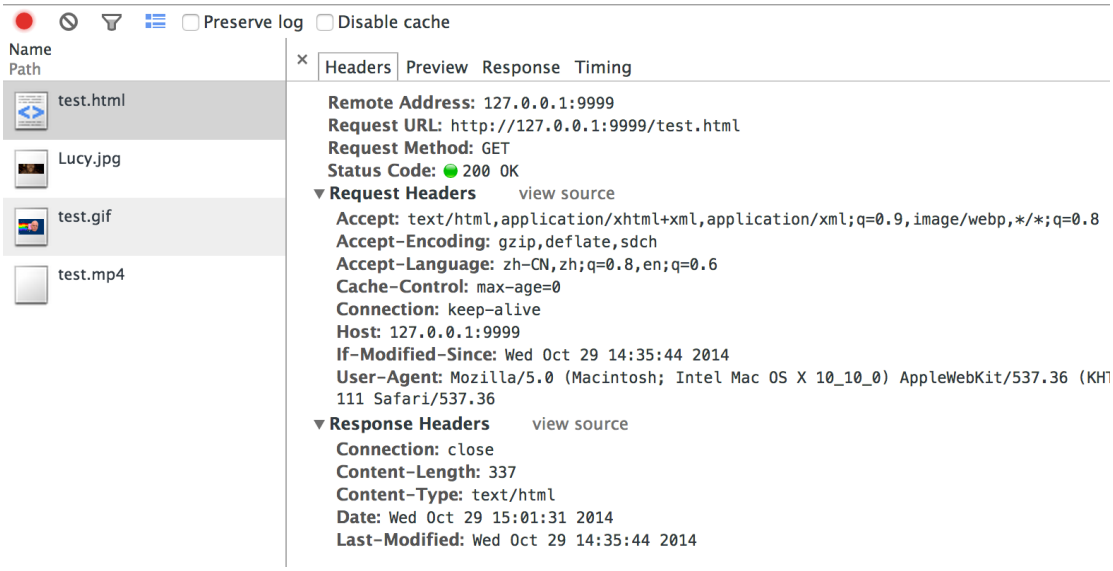


Here we used chrome tool to analyze the response message from the server to the browser.

In the screenshot below we can see clearly 4 responses to each HTTP GET request.



Furthermore, I checked the header of one response message. As shown in the screenshot below, the status code was **200**, and Header lines of **Date, Last Modified, Content-Length, and Content-Type** were all included.



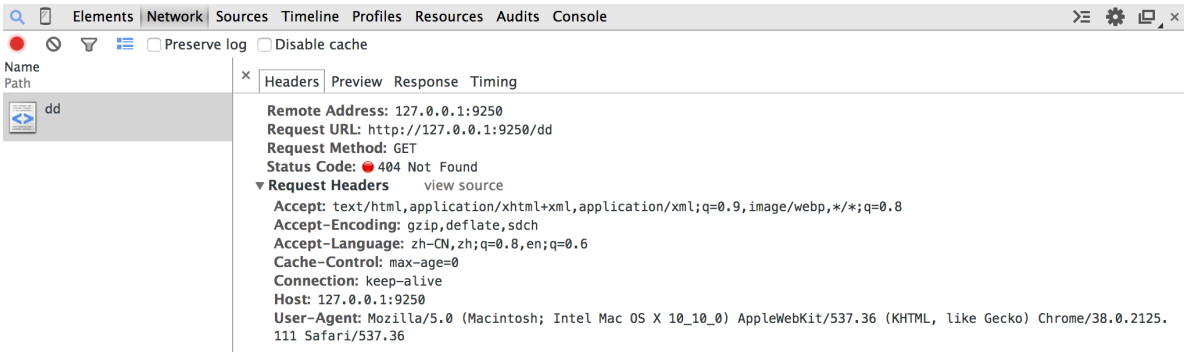
The following chart shows the performance of *serverFork* (block mode)

	Chrome (Mac OS)	Safari (Mac OS)	FireFox (ubuntu)
JPEG	normal	normal	normal
GIF	normal	normal	normal
MP4	*Only audio*	normal	*blackbox*
HTML	normal	normal	normal
PDF	normal	normal	*download*
Wrong filename	normal	normal	normal
Wrong header	normal	normal	normal

* Abnormal performances are marked in *italic*.

A request for invalid filename

Here is the result of wrong filename in the request. Server will check the filename, if there has no such file at local, an HTTP 404 response will be send back. We can see the header information of response in the screenshot below.



Serve Multiple TCP Connections Simultaneously

In addition to the server under block-mode, we have implemented select function to the server in order to enable the server deal with multiple sockets concurrently. To test the performance of the blocked-mode server and non-blocking mode server, we wrote a shell script to "wget" the *test.html* file on the server 20 times. The difference of performance was quite significant: concurrent server could handle all 20 requests at the same time, on the other hand, the block-mode server could only deal with one socket at a time so the rest request are queuing to get accepted.

To be honest, the concurrent server we designed is not yet perfect. Some bugs do exist and will terminate the server. We have observed bugs for video file, the browser could not play and sometimes server would fail. Our guess is the connection may be abnormally shut down or reset and different browsers may deal with video file through various ways. Due to time limits, we didn't fix such bug yet. Here is a diagram that shows the performance of our concurrent server under different tests.

	Chrome (Mac OS)	**Safari (Mac OS)**	**FireFox (ubuntu)**
JPEG	normal	*download*	normal
GIF	normal	normal	normal
MP4	*Only audio*	*Server crash*	*download*
HTML	normal	normal	normal
PDF	*download*	*download*	*download*
Wrong filename	normal	normal	normal
Wrong header	normal	normal	normal

* Abnormal performances are marked in *italic*.