CS 118 Project 1 Report

Web Server Implementation using BSD Sockets

Yukai Tu 204761085
Xufan Wang 204477479

February 3, 2017

Introduction

For this project, we implemented a web server in C using the BSD socket programming library on a 32-bit Ubuntu Virtual machine. The server implementation include two parts. For part A, the server can receive HTTP requests and then dump request messages to the console. For part B, one more functionality is added to the server, so that it can further parse the requests, generate HTTP response messages consisting of the requested files preceded by header lines, and send the responses back to the client.

Compilation and Usage

Tester can use the following steps to compile the server.

1. Navigate to the directory containing the source files.
2. Run the command "make".
2. Run the command "chmod +x serverFork".

After compilation, use the command:

```
./serverFork <portnum>
```

to start the server. In the above command, <portnum> is a port number that the server will bind to, so the same port should be used for the client to send request.
After the server starts, the client can connect to the server by accessing following URL using a web browser.

```
http://<machine name>:<portnum>/<html file name>
```

In the above command, <machine name> should be the name of the server machine, and <html file name> should be the name of a file that the client would like to receive.

High-Level Description

For both Part A and Part B, the server builds a communication point through a port by using functions in the libraries:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

which define different data structures and socket operations.

To be more specific, the server first use the `socket` function to create a socket. It then uses the function `bind` to bind the new socket to a port on the local machine. The port number is specified by the user. After doing those, the server starts listen to the socket for any client requests, and this is done by using `listen` function. By now, the socket is ready and requests

can be sent through it by the client.

The server accepts requests with the function `accept`. To keep the server ready to accept new requests, once the server receive a request, it will fork a child process to serve the request, so that the parent process can continue to run `accept` and wait for the next request. When the child exits, the parent handles the signal `SIGCHILD` to prevent zombie processes.

As the child process is forked, it starts to parse the request, send console outputs and responses, and exit when all the work is finished for that request. It mainly runs two functions, `parseRequest` and `serveRequest`.

The dumping functionality in part A is implemented in `parseRequest` function:

```
void parseRequest(int sock, RequestHeader* header);
```

It first read the request through socket file descriptor `sock` into a buffer, then print out the content of the buffer, which is a request line followed by several header lines. It also extract the header of the request and store it into `header`. This header is used for part B.

The functionality of sending response in part B is implemented in the `serveRequest` function:

```
void serveRequest(int sock, const RequestHeader header);
```

The function tries to read the file specified in `header`. If the file does not exist or it cannot be opened, it will send an error message to inform the client. If the file exists, the server will send an HTTP response message, with the file included, to the client. The server determines the MIME type of the requested file by looking at the URI in the header, and generate different response messages accordingly.

Difficulties

The main difficulty of this project lies in the formatting of response messages. Based on RFC 1945, the header can include many header lines. It took us some time to figure out which ones should be included in the header and which should not.

Also, we tried many ways to parse the header of request messages. We are currently using the function `strtok`, which splits the buffer by delimiter to accomplish this, since the function uses only a few lines. However, since the function will change the original buffer, it can only be used to parse the first header line. The method is sufficient but not elegant enough.

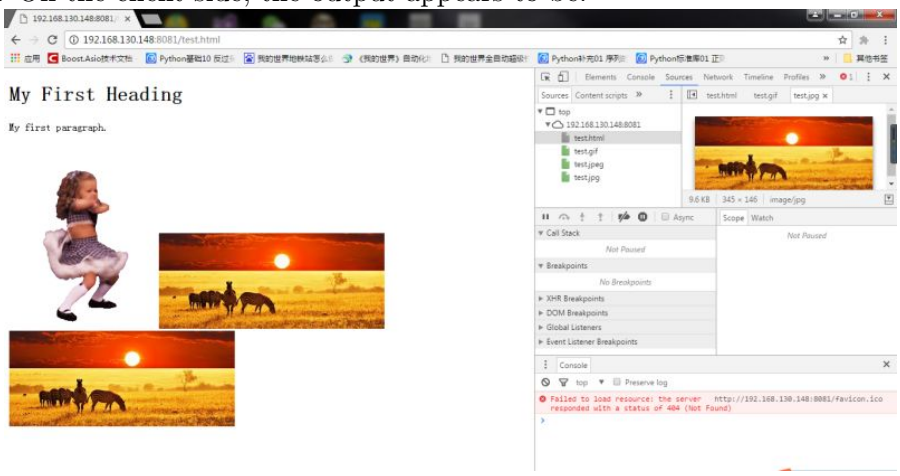
Another difficulty would be that when we tried to send images to the browser, we always get "CONTENT LENGTH ERR". We debugged the code which gets the length of the content, but we eventually found out that the bug is cause by the wrong buffer size of the response message.

Sample Output

After starting the server, we can send requests to the server by a browser. Here is the sample output from the request:

```
http://192.168.130.148:8082/test.html>
```

In which, "test.html" is an HTML file which refers to a JPEG file and a GIF file. On the client side, the output appears to be:



On the right side, we can see that the HTML file actually refers to three files, and they are all displayed correctly on the left side, in the HTML format.

At the server side, the console displays the message contents of four requests. The first one is for the HTML file.

```
request info:
 GET /test.html HTTP/1.1
Host: 192.168.130.149:8081
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.87 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0
.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8


resource fmat:0, length:171, fileLength:171
generate response
response content:
 <!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>
<img src='test.gif' />
<img src='test.jpg' />
<img src='test.jpeg' />

</body>
</html>
```

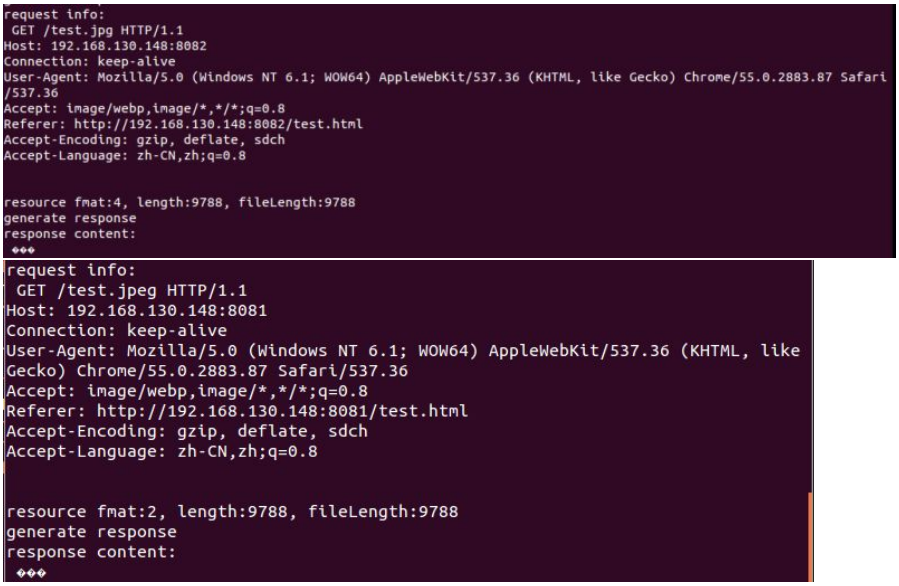
The request message is displayed in the beginning. The first line is the request line, which specifies the method "GET", the URI "/test.html", and the HTTP version. It is asking for a file with the URI specified. The second line is the address of server host. The third line specifies the current connection type. The fourth line suggests that the browser prefer to redirect to an HTTPS connection. The fifth line tells the information of the user. The sixth, seventh, and eighth line lists the acceptable types, encoding and language of the response respectively.

After the request message is printed, the server starts to output information about the response. For example, the first line in the graph above suggests that the response format is HTML (represented by 0) and the length of the file is 147. It also prints out the content of the file.

After the HTML file is sent, the browser parses the file, finds the references, and places three more requests. The console output for those three are:

```
request info:
 GET /test.gif HTTP/1.1
Host: 192.168.130.148:8082
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2883.87 Safari
/537.36
Accept: image/webp,image/*,*/*;q=0.8
Referer: http://192.168.130.148:8082/test.html
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8


resource fmat:3, length:57684, fileLength:57684
generate response
```



Those request lines are mostly in the same format as in the HTML case, besides that they include a new line to suggest that they are referred by the previous HTML file, and do not inlcude the line for requesting a redirection to HTTPS.

Conclusion

This project implementation meets all the requirements in part A and part B successfully. The web server gives us a better understanding on the client-server model and socket programming with the libraries. It also illustrates an example for HTTP request and response. Though the functionality of the server is simple, it does cover the important concepts, like port, socket, HTTP messages, and etc.