CS 118: Computer Network Fundamentals

Professor Lu

Project 1

Roger Chen, 504043927

Kailin Chang, 503999157

High-Level Description:

To create a web server that dumped request messages to the console for Part A, a socket is initialized and the port number is taken as a parameter. The port number is bound and the main server waits for a request to come in. When the request comes in, the server forks a child process to serve the request.

For Part B, after the child process has been forked, it parses the request header in order to determine what file to serve. It then creates a response message to serve to the client by filling out the header fields and opening up the file and writing the contents to the socket. To format the response message properly, we referenced the example response message on page 106 in the textbook.

```
Date: Tue, 09 Aug 2011 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html

(data data data data data ...)
```

We implemented various helper functions to get the data required for the header fields. To deal with the body of the request, we made a function to extract the file extension and used that to determine how to handle the file. If it was not a .html, .jpeg, .jpg, or .gif, we simply returned a 404: Not found error to the client.

Difficulties Faced and Solutions:

The very first difficulties we encountered for this project were simply understanding the specifications and determining what we needed to add to the skeleton code. Before any actual

coding, we had to decide if we were going to use C or C++. We ultimately decided on using C++ to leverage the advantages of using I/O streams and C++ strings over parsing one character at a time and reading it into a C-style string. An example of this was that determining the content length of the file became trivial using `ifstream` and `tellg()`. This also required us to modify the Makefile's chosen C compiler from `gcc` to `g++`.

At first, we thought the skeleton code already implemented all of Part A for us, but upon closer inspection, we realized that the request message dumped to the console was only partially complete. This turned out to be an easy fix, as the original skeleton code only read the first 256 bytes of the request message, so we added a `while` loop to continue reading until it reached the `<cr><lf><cr><lf>` which signifies the end of the header lines.

Getting the information required for some of the header lines was challenging without searching on Google for libraries that could do the functionality we wanted. For example, we learned that we could use the `time()` function in `time.h` to determine the current time in order to populate the `Date:` header field. To fill out the `Last Modified:` header field, we had to research about the `stat()` function located in `sys/stat.h`. In addition, using the above functions to retrieve the data we needed occasionally resulted in extra `\n` bytes being added at the end. This problem strongly affected the format of the response message we generated using those functions, but fortunately, the problem was easy to once we realized where the issues lie. We simply used the `substr()` function to remove the extra last byte from each of the affected responses.

One of the most difficult challenges we encountered was file input/output. The biggest issue was that we did not properly detect the end of file using `!eof()`, because we always had an extra byte of random character added at the end. After looking at what `ifstream` provides more in depth, we resolved the issue with reading in data.

The biggest difficulty we encountered for this project was working together. Issues such as figuring out how to share work and dividing up the work came up. We decided on setting up a git repository to share our code. We made our functions as modularized and descriptive as possible, so it was easy to assign tasks. Furthermore, we employed a bit of TDD to keep the functions we implemented correct.

Compiling and Running The Code:

1. `make`
2. `./serverFork sample_port_number`
3. Access `localhost:sample_port_number/file`

Results:

Here is a list of files in the same directory as `serverFork.c`:

```
-rwxrwxr-x 1 cs118 cs118   29104 2014-10-31 20:12 serverFork
-rw-rw-r-- 1 cs118 cs118   21080 2014-10-31 20:12 serverFork.o
-rw-r--r-- 1 root  root    10080 2014-10-31 20:12 serverFork.c
-rw-rw-r-- 1 cs118 cs118 2001029 2014-10-30 17:44 output.txt
drwxrwxr-x 2 cs118 cs118    4096 2014-10-30 17:40 test
-rw-r--r-- 1 cs118 cs118 2001029 2014-10-29 21:15 samoyed.gif
-rw-r--r-- 1 cs118 cs118    5340 2014-10-29 21:15 samoyed.jpeg
-rw-r--r-- 1 cs118 cs118   61166 2014-10-29 21:15 samoyed.jpg
-rw-r--r-- 1 cs118 cs118      43 2014-10-28 16:12 README.md
-rw-rw-r-- 1 cs118 cs118     158 2014-10-28 16:12 README
-rw-r--r-- 1 cs118 cs118    1075 2014-10-28 16:12 LICENSE
-rw-r--r-- 1 root  root      154 2014-10-28 16:12 Makefile
-rw-rw-r-- 1 cs118 cs118    2105 2014-10-28 16:12 client.c
-rw-rw-r-- 1 cs118 cs118      42 2014-10-28 16:12 hello.html
-rwxrwxr-x 1 cs118 cs118    7865 2014-10-25 20:58 client
```

We open up the server at port 12345 and we request a file called `samoyed.jpg`. The request message is shown below:

```
cs118@ubuntu:~/Desktop/workspace/ClientServer_Example$ ./serverFork 12345
Here is the message: GET /samoyed.jpg HTTP/1.1
Host: localhost:12345
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
If-Modified-Since: Wed Oct 29 21:15:48 2014
```

The client would see a picture of a Samoyed in their browser, shown below: