CS 118: Computer Network Fundamentals

Professor Lu

Project 1

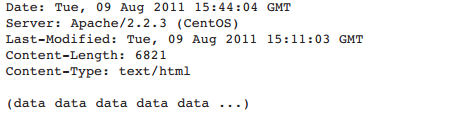
Roger Chen, 504043927

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.



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 and StackOverflow for ideas. References will be provided at the end of the report. 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 consulting numerous sources on Google and StackOverflow, we finally came across a solution that used ifstream that 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:**

The project contains a Makefile that builds serverFork.c and simply running make will compile the code and create an object file called serverFork. Then, run ./serverFork sample\_port\_number. For example, to run the server on port 10222, run the command ./serverFork 10222. To request files from the server, the files have to be contained in the same directory as serverFork.c or be in the subdirectories. For example, to access, if you want to request a file hello.html located in /subdir/hello.html you would point your browser to localhost:10222/subdir/hello.html if the server is initialized to port 10222.

To summarize this in an easy three-step process:

1. make
2. ./serverFork sample\_port\_number
3. Access localhost:sample\_port\_number/file

**Results:**