Homework #2: HTTP Web Server and Client

1 Objectives

In this homework, you will gain experience with writing TCP server/client and specifically with the HyperText Transfer Protocol (HTTP). Specifically, during this project you will implement a simple HTTP server that can serve up web pages stored on your hard drive, when requested by a web browser; you also implement a simple HTTP client that can request HTML page on a given URL.

Implementation 1:
Your HTTP server, *webserver*, should be written in C or C++, compiled using a Makefile (which you will provide), and run from the command-line using the syntax described in Table 1. For example, if you run the following command:

```
./webserver -p 10000 ./docs/webpages
```

Your server should start up, and listen on port 10000 for TCP connections. When a client connects and sends an HTTP GET request for mypage.html, your server will look for *./docs/webpages/mypage.html*. If it exists, it will send an appropriate **HTTP OK 200** response containing the contents of *mypage.html*. If the page can't be found, then the server should respond with an **HTTP 404 ERROR**.

For this project, you will want to keep the HTTP documentation handy (see the following URL).

http://tools.ietf.org/html/rfc7231 (but also consider RFC7230-7237 and RFC3986)

Your server will need to implement a subset of the HTTP/1.1 spec. It needs to support the **GET** and **HEAD** methods. Your server should appropriately provide the following responses: **200** for successful requests, **400** for malformed HTTP requests, **403** for requests for pages outside of the pages directory or for which the server doesn't have read access, **404** for requests for pages that can't be found, **405** for requests using any method other than GET or HEAD, and **414** for any requests with URI (absolute-URI) which are longer than 30 characters

Table 1: HTTP server syntax

```
webserver [-t port] [directory]

port (Optional): Your web server's port. Default value: 8080

directory (Optional): The directory on the server's file system, where
the web pages are stored. Default value: ./ (the same directory as
./webserver hosted)
```

You do not need to support persistent connections (just send a Connection: close header with every response, so the client knows that the connection is going down.)

Implementation 2:
Your HTTP client, *getfile*, should run from the command-line using the syntax described in Table 2. For example, if you run the following command:

```
> ./getfile http://www.cs.clemson.edu/help/index.html -p 8088 -f index.html
```

Your client should be able to send request to HTTP server www.cs.clemson.edu on port 8088 for getting /help/index.html document with TCP connection. When all data has been received, we should get a textual file index.html at current directory.

If there is not a specified filename on your HTTP client, your program should print the received web page to stdout directly. It should have a very similar behavior and result as the following wget command:

```
> wget http://www.cs.clemson.edu/help/index.html -O -
```

Table 2: HTTP client syntax

```
getfile URL [-t port] [-f filename]

URL (Required): The requested URL.

port (Optional): The web server's port. Default value: 8080

filename (Optional): The document (web page) will be saved as a textual
file instead of printing at stdout.
```

2 Headers

Your server should tolerate all HTTP headers (they shouldn't crash your server), but many of them can safely be ignored. Here are the headers that you should not ignore.

Requests

- All requests should include a Host header. Your server should return a **400** (bad request) response to any request that does not include this header.

Responses

- You should always include the Connection: **close** header, to tell the client that the connection is not persistent. If you don't, you might confuse some clients that try to use persistent connections.
- All **405** responses should include an appropriate **Allow** header.
- All responses should include a **Date** header with the server's date and time.
- All responses should include a **Last-Modified** header with the date and time that the file was last modified (from the file system). Use the **stat** function to get this information.
- All responses that contain a message body (like a requested web page) should include a correct **Content-Length** header.
- Your responses should also include a **Content-Type** header when returning a requested web page, image, or file. Your server should return the following MIME types: **text/css** for files ending in ".css", **text/html** for web pages(files ending in ".html" or ".htm"),

application/javascript for ".js" files, **text/plain** for ".txt" files, **image/jpeg** for ".jpg" files, and **application/pdf** for ".pdf" files. All other files should get the type **application/octet-stream**, which represents arbitrary binary data.

- All responses should include a **Server** header that identifies the server (and version number) in use (like "MyLittleHTTPD/1.2").

HTTP server output format:

```
<METHOD><tab><URL><tab><DATE><tab><RESPONSE>
```

Example output:

```
GET  index.html     2 Dec 2014 19:43    200
HEAD cpsc3600/schedule.html   3 Dec 2014 5:43     200
GET  ../../etc/password.txt   3 Dec 2014 5:46     403
POST cpsc3600/schedule.html   3 Dec 2014 5:48     405
```

Note: Your server should write a single line to STDOUT for each received HTTP request, including requests that result in errors.

Example request:

```
GET / HTTP/1.1\r\n
User-Agent: Wget/1.14 (darwin12.2.1)\r\n
Accept: */*\r\n
Host: www.cs.clemson.edu\r\n
Connection: Keep-Alive\r\n
\r\n
```

Example response:

```
HTTP/1.1 200 OK\r\n
Date: Thu, 20 Feb 2014 22:07:47\r\n
Last-Modified: Thu, 20 Feb 2014 22:07:47\r\n
Content -Type: text/html\r\n
Content -Length:    34\r\n
Server: MyLittleHTTPD/1.2\r\n
\r\n
<html><body>Hi World</body></html>
```

After each response is sent back to the client, your server should close the current connection, and then accept the next connection.

You should not use any HTTP libraries or system calls that send, construct, or interpret HTTP requests or responses. You should write code from scratch to interpret all HTTP requests, and create all HTTP responses from scratch.

As with all programming assignments in this course, all of your code must be your own.

3 Output

Your program should log the requests it receives to STDOUT, as shown in Listing 2. This should be the only thing printed to STDOUT. Any debug information should be written to STDERR.

For simplicity, your server does not need to be multithreaded, and can simply handle connections sequentially, in the order they arrive.

4 Submission Instructions

When your project is complete, archive your source materials, using the following command:

```
tar cvzf youruserid-homework2.tar.gz README Makefile {source
files}
```

Note: Do not include any subdirectory.

The Makefile should build your program (by running make).

The README file should include your name, a short description of your project, and any other comments you think are relevant to the grading. It should have five clearly labeled sections titled with KNOWN PROBLEMS, and DESIGN. The KNOWN PROBLEMS section should include a specific description of all known problems or program limitations. This will not hurt your grade, and may improve it. I am more sympathetic when students have thoroughly tested their code and clearly understand their shortcomings, and less so when I find bugs that students are either ignorant of or have tried to hide. The DESIGN section should include a short description of how you designed your layers (especially anything you thought was interesting or clever about your solution). You should also include references to any materials that you referred to while working on the project. Please do not include special instructions for compilation. The compilation and running of the tests will be automated (see details below) and your instructions will not be followed.

Please make sure you include only your source files, not the object files.

Before you submit this single file, please use the following command to check and make sure you have everything in the youruserid-homework2.tar.gz file, using the following script:

```
> mkdir test
> cd test
> tar xvzf ../ youruserid-homework2.tar.gz
> make
> ./webserver -t 8090 ../testwww
```

These commands should put all of your source files in the current directory, compile your webserver program, and start your HTTP server on port 8090 serving up the pages stored in the "../testwww" directory.

Submit your youruserid-homework2.tar.gz file via handin.cs.clemson.edu. You must name your archive youruserid-homework2.tar.gz, and it must compile and run. We will compile your code using your Makefile, and run it using the syntax described in Table 1 and 2, but using our own directory full of pages and files.

We will test on the SoC Lab machines. Make sure it runs there!

5 Grading

Your project will be graded based on the results of functional testing and the design of your code. We will run several tests to make sure it works properly and correctly handles various error conditions. We will test your HTTP server against real HTTP clients (e.g. Firefox, Chrome, lynx, wget, nc) as well as our own custom HTTP client that sometimes misbehaves (mishandling connections and sending malformed HTTP requests, for example). Similarly, we will test your HTTP client against real HTTP server, as well as your own server.

You will receive 10% credit if your code successfully compiles, and 10% for code style and readability. The rest of your score will be determined by the number of tests that your client passes. These tests will evaluate both your server's output (to stdout) and its responses to the test client. Your source materials should be readable, properly documented and use good coding practices (avoid magic numbers, use appropriately-named variables, etc). You code should be -Wall clean (you should not have any warnings during compilation). Our testing of your code will be thorough. Be sure you test your application thoroughly.

6 Collaboration

You will work independently on this homework. You must not discuss the problem or the solution with classmates, and all of your code must be your own code.

You may, of course, discuss the homework with our grader, but you may only discuss conceptual issues related to the homework that we have already discussed in lecture. (**Warning: you should not have any algorithm and source code level discussion with our GTA**). **Collaborating with peers (or strangers on the Internet) on this homework, in any other manner will be treated as academic misconduct.**