A Simple HTTP Server

1 Background

HTTP GET requests are used to request data from the specified source. For example, if we want to access a webpage at http://www.foo.com/bar.html, our browser will send something similar to the following HTTP GET request.

```
GET /bar.html HTTP/1.1
Host: www.foo.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:28.0) Gecko/20100101 Firefox/28.0
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
Connection: keep-alive
```

In the above example, our browser is the client, and Host: www.foo.com is the server host name. The client is requesting server resource located at /bar.html. Usually, HTTP servers run on port 80. So, by default, the server port number is 80. If, for example, the HTTP server runs on port 8080, then the URL would be http://www.foo.com:8080/bar.html, and the Host field in the HTTP request will also contain the port number: www.foo.com:8080.

The HTTP server replies with HTTP response. A valid HTTP response includes: (1) the status line, (2) the response header, and (3) the message body. For example, the text below shows an example HTTP response. Note that there is an empty line between the response header and the message body. This empty line indicates the end of header fields[1].

```
HTTP/1.1 200 OK
Date: Thu, 02 Apr 2015 01:51:49 GMT
Server: Apache/2.2.16 (Debian)
Last-Modified: Tue, 10 Feb 2015 17:56:15 GMT
Accept-Ranges: bytes
Content-Length: 2693
Content-Type: text/html

... ... <content of the bar.html>
```

2 Multi-threaded HTTP Server

Your HTTP server should support a subset of the HTTP standard. It only needs to serve HTTP GET requests and should only support basic portions of the standard. That is, you do not need to support persistent connections, request pipelining or other, advanced portions of the standard. Your server must only correctly handle HTTP requests where a single request/response pair is sent over a single TCP connection. Specifically, your server should operate as follows:

- Your HTTP server should not take any command line arguments. It should create a TCP server socket to listen for incoming TCP connections on an unused port, and output the host name and port number the

[1] https://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html

HTTP server is running on. Note: You must use sockets in this project. Any implementation without the use of sockets will receive zero points.

- It should look for a directory called www located in the same directory as your HTTP server executable. The www directory should contain resources you want your HTTP server to serve. If this directory does not exist, the HTTP server should output an error message and quit.
- When a new request comes in, the HTTP server accepts the connection, establishing a TCP connection with the client.
- The HTTP server parses the HTTP request from the client and prepares an HTTP response.
- The HTTP server looks up the requested resource from its local www directory.
  - If the requested resource is located by the HTTP server, prepare a response with status code "200 OK", followed by response header and the content of the requested resource. The response header should include the following fields:
    - Date The date and time the response is originated in the format defined by RFC 7231 Date/Time Formats [2].
    - Server A name for your HTTP server. You can be very creative here.
    - Last-Modified Last modified time and date of the requested resource, also in RFC 7231 Date/Time Formats.
    - Content-Type The MIME type of this content. You can use the mime.types file located at /etc/mime.types to determine the correct MIME type given a filename extension. If a filename extension is not found in the mime.types file, you can use application/octet-stream.
    - Content-Length The length of the requested resource in bytes.
  - If the requested resource does not exist, reply with status code "404 Not Found". You can be creative and write a fancy custom error page to return to the client.
- After finishing sending the response, the HTTP server closes the connection socket to the client.

To maintain a high level of throughput even under multiple simultaneous client requests, different requests should be handled in different threads, with a new thread created for each client request.

2.1 How to test your implementation

You must test your HTTP server on the remote nodes of CS department computers: remote.cs.binghamton.edu. When accessing remote.cs.binghamton.edu, you are actually redirected to one of the 8 REMOTE machines ({remote00, remote01, ..., remote06, remote07}.cs.binghamton.edu) using DNS redirection. So to test your implementation, you need to run your HTTP server on one of the REMOTE nodes and set your web client, e.g., wget[3], to use the correct server host name: "remoteXX.cs.binghamton.edu", instead of "remote.cs.binghamton.edu".

[2] https://tools.ietf.org/html/rfc7231#section-7.1.1.1
[3] http://linux.die.net/man/1/wget

For example, suppose your HTTP server is started on remote02.cs.binghamton.edu port 47590. You can run the following command in an EMPTY directory on a different REMOTE computer to download the resource:

```
wget http://remote02.cs.binghamton.edu:47590/bar.html
```

Note:

- You MUST replace bar.html with a valid resource in your HTTP server directory!
- Since this is only an HTTP server, not an HTTPS server, your URL MUST use HTTP, not HTTPS.
- You may also run your HTTP server on your own machine and use Wireshark[4] for debugging. However, you must test your implementation on REMOTE computers before submission.
- Your web browser has a cache. A second request for the same URL may be directly served by your browser cache without going through your server. Therefore, you are recommended to use wget for debugging and testing.

After wget successfully downloads the requested resource, use the diff[5] command to check if the downloaded resource matches the resource located in the www directory. wget has a few options that can help you debug your implementation. For example, the --limit-rate option allows you limit the download speed to a pre-specified amount. This can be helpful if you want to debug your multi-threaded implementation. If you need to inspect if your HTTP server constructs the HTTP response header correctly, you can take advantage of the --save-headers option. More information can be found at the manual page of wget.

Upon successfully serving a request, your HTTP server must write to standard output (stdout) the following items:

requested resource client IP client’s IP address in dotted decimal representation access times the

number of times this resource has been requested since the start of the HTTP server

These three items of a same request should be printed to a same line and seperated by the “|” character. Below is an example:

```
/bar.html|128.226.118.20|1
/pics/foo.jpg|128.226.118.20|1
/bar.html|128.226.118.26|2
/bar.html|128.226.118.21|3
/pics/foo.jpg|128.226.118.21|2 /pics/foo.jpg|128.226.118.26|3
```

[4] https://www.wireshark.org/
[5] http://linux.die.net/man/1/diff