Project 1 – CS 356 – Computer Networks – David A. Bryan
HTTP Server

In this project you will be constructing a (very) basic web (HTTP) server using Python 3. The server will serve up basic text, HTML, and JPEG files, and must produce error messages (in the responses sent back!) for a number of cases we will test. You will also write (but not turn in) some short clients (based on an example I will give) to test your server.

You should look at the slides and book (section 2.2) for the basics of HTTP/1.1. You can also find very detailed specifications in the HTTP/1.1 specification ("the RFC"):

https://tools.ietf.org/html/rfc2616

When in doubt, check the specification! It may be cryptic, but it has the right answer! You should use the specification to get the right error codes for some errors you need to generate.

Python Version, Using Modules/Libraries, Logistics/Rules:

Your code will be tested using the `python3` command on the CS Linux machines. (currently version 3.4.3) ***Saying "it ran on my machine" is NOT reason for regrade!*** We may also run the client and server on the same machine and on two different machines. Make sure you test both! ***Make sure you use version 3 of python! (`python3`, not `python`!)***

You will need to write most of this in Python yourself. You can ***ONLY*** use the few libraries I list below. ***YOU CANNOT USE ANY OTHER LIBRARIES OR UTILITIES UNLESS YOU GET APPROVAL FROM ME!*** Most obviously, you can't use the built in HTTP server in Python! You can use the following modules:

- socket, of course
- sys, the system utilities. Among other things you will need this to get command line options.
- time and datetime to get and modify dates and times.
- re (the regular expression module)
- os.path and/or os.stat (to get file information)
- signal, if you want to get fancy and catch ^C to clean up the socket on close (not required)
- ***DON'T USE ANY OTHERS WITHOUT PERMISSION***

You will be turning in ONLY your server, using Canvas. Make sure it is readable, well commented, doesn't contain offensive variable names, etc. Name your server server-***uteid***.py, so if your UT EID is abc123, your file should be called server-abc124.py. It's ok if canvas appends a number.

Your server **MUST** take exactly one command line argument, which is the port number to be used, to allow for ease of testing. Trust me, you will want this, since if you crash, the socket will be "in use" for a few seconds, and this lets you switch easily. Information about how to handle command line arguments in Python is widely available online. ***If you run on some port you hardcode and not the command line argument passed, it won't run, and you will likely get a zero!***

Basic Functions, Testing and Grading:

Files requested (supported) only need to be text (.txt extension), HTML (.html or .htm extensions), or JPEG (.jpeg or .jpg). HTML files can include references to JPEG files (i.e., IMG tags) Files will all be in the current working directory where the server is run. Requests will be things like /test.html or /file.jpeg – you can ignore subdirectories. Your server will only support the HTTP `GET` method, no other request methods. Note that even though HTTP requests start with a /, that doesn't mean the root of the filesystem (/), but the directory where the program is running (working directory).

Given the above restriction, your server needs to be able to receive requests and respond properly to (simple) requests from any major browser (Chrome, Mozilla, etc.) and test scripts. It must generate (appropriate) errors for things it doesn't understand, files it can't find, etc.

Make sure you test with a browser, sending requests for both valid and invalid object types, files that exist (text, html, jpeg, and html with jpeg), and files that are missing, etc. Return correct error codes (see the specification!) for missing files, requests for unsupported objects, etc. The browser testing will be roughly half of your grade,

You also will be provided two very simple test scripts, one making a valid request, and one with a syntax error in it. You should hack together a number of other basic test client scripts based on that to send errors you can't otherwise get the browser to generate (for example, syntax errors in messages) to make sure your server supports them. Your test clients can be as ugly as you want. They won't be turned in or graded – you will only submit your server.

We will test your server will be tested with several browsers. Additionally, test scripts will send requests with certain errors (see below), which your server must handle properly.

Handling Requests:

Again, as stated above, your server will have to accept only HTTP `GET` method requests -- no other type of request is required (or allowed) by your server. Files requested will only be HTML (.html or .htm), text (.txt), or JPEG (.jpg or .jpeg).

To handle JPEG you will need to read and send a binary file...how to do this is left as an exercise for the reader.

All files requested will be in the same directory (working directory) we run your server in. So if we request /test.html, expect it to be in the same directory as your code. We won't ask for things in subdirectories in our testing.

You need to support the following features. In many cases, we will send you errors (malformed messages, ask for things that don't exist, etc.) from our automated scripts, and you should handle it properly, returning the correct (error) status code and phrase in your response. You will need to look in the book, or better yet, the specification to determine what the correct error status code and phrase are. You need to handle the following features/errors/scenarios – to get full credit make sure you have created a test script to check each:

- You **MUST** properly deliver a requested .html, .htm, .txt, .jpg, and .jpeg file when present, both to the script and to the browser. Make sure you actually see the JPEG in the browser.
- You **MUST** return the proper error when a file is not present.
- You **MUST** handle conditional GET properly. (See 2.2.5 in the book.) We will test this having a file with a known modification date and sending requests for it with `If-modified-since` header set both before and after the modification date and checking that it is properly handled.
- You **MUST** make sure the request line sent to your server has exactly three parts -- method, URL, and Version. Missing or too many parts should generate an error (which error? Look in the RFC!)
- You **MUST** make sure the version in the request line HTTP/1.1. Your servers must ONLY accept version 1.1, and will have to generate an appropriate error if it is something else. (again, which error? Look in the RFC!)
- Your server only supports the `GET` method, and **MUST** respond with proper error handling for other requests sent to it.
- While you can ignore many (in fact most) of the headers you will get from the test scripts of browser, HTTP/1.1 has ***required*** header(s) for requests, and you **MUST** make sure these are present. (see book or the specification to see what is required)
- Your server **MUST** make sure that basic syntax rules about where spaces, cr/lf (including the final line), blank lines, etc. are followed. You should reject malformed requests with the proper response. Again, look at Section 2.2 of the book, or better yet, the specification.
- You **MUST** support a valid request for an existent file, in the form of html/htm, txt, or jpeg/jpg.
- You **MUST** properly handle requests for non-existent files.
- Finally, you **MUST** handle conditional `GET` requests – that is, respond properly to requests with "`If-modified-since`" headers, taking into account the time the requested file was last modified. See 2.2.5.
    - That also means your server **MUST** include "`Last-modified`" headers for all files it sends – see below in generating responses section.

Generating Responses:

You will need to be able to generate a number of responses – again, for the scenarios listed above several different responses (status codes and phrases) will be required. For each, you need to detect if things are correct and generate a response, or deliver the (correct) error. Response status code (200, 404, etc.) and phrase (OK, Not Found, etc.) need to be set appropriately.

Your response **MUST** have to have the right format -- see figure 2.9 in the book and the specification. You also must get the following correct:

- Responses **MUST** have correct syntax (Status line is correct and follows the right rules, spaces, blank lines, cr/lf (\r\n) in the right places, etc.)
- You **MUST** include the "`Server`" header. You can make up any server name for your server you want (in the example in the book it is an Apache server...call yours anything! Just make sure the header is present.)
- You **MUST** include the "`Date`" header. You will need to look up how to get the date in Python (it's easy)
- You **MUST** include a (correct) "`Content-Length`" header for your responses.
- You **MUST** include a (correct) "`Content-Type`" header based on the file type requested. Again, only html, txt, and JPEG files are supported – anything else requested should be considered an error and handled accordingly.
- You **MUST** include a correct "`Last-Modified`" header to support conditional GET, as described above.

A Few Thoughts:

- You can assume no file that is requested is ever over 8192 characters/bytes if you would like (although you can just as easily handle larger things)
- If you can, install wireshark on your own PC (wireshark.org) and use your PC to send the requests to your server. Then you can also run wireshark on your PC and capture the traces! (you won't be able to do this on the CS machines) Just make sure your final testing is on the CS machines.
- Pick a unique port -- anything between 12000 and 64000 -- for your server (passed as a command line!). When you point your browser at the server, you will have to type "hostname:12000", so if you choose 12000 and run your server on mentos, and ask for test.txt, what you type in the browser will be "mentos.cs.utexas.edu:12000/test.txt" Make sure to try with server and browser on two different machines, too!
- You are welcome to discuss ideas, Python syntax, HTTP specification questions, etc. with other students, but **MUST NOT** share code or show your code to other students. We will compare files to test for similarity and look for collusion!

Testing Self-Checklist

Use this list to make sure you have a properly working server. We might test a few other things (see the list above), but if you have these, and think you meet all the bullets in Handling Requests and Generating Responses above, you should be in reasonable shape:

☐ HTML file (as .htm or .html) loads from browser and from script.

☐ JPEG file (as .jpg or .jpeg) loads from browser and from script.

☐ An HTML file that refers to (includes an `img` tag) JPEG file works in browser.

☐ You return the right status code for a valid request.

☐ Your response is formatted correctly. Check spaces, cr/lf, etc. See 2.2.3.

☐ You return the correct "`Content-type`" header in responses.

☐ You return the correct "`Content-length`" header in responses.

☐ You return a valid "`Server`" header in responses.

☐ You return the correct (current) `date` in the correct header in responses.

☐ You return the correct modification time for files in responses.

☐ You correctly handle requests with "`If-modified-since`" headers asking for a file using times both before and after the file was last modified and behave correctly for each case.

You catch/return the (correct! Check the spec!) error code for at least the following errors: (there will be a few more, so be clever in coming up with ideas!)

> ☐ Missing file requested
>
> ☐ Method other than `GET` (both unsupported and non-sense) sent to your browser
>
> ☐ Wrong HTTP version sent to your browser
>
> ☐ Requests missing a required header is sent to your browser
>
> ☐ Cr/lf wrong or missing in request sent to your
>
> ☐ Request line has too many parts (hint: this is the SyntaxError1.py script I gave you), has too few parts, one or more of the parts is wrong or malformed, strange separators used, etc. In otherwords, make sure the request line is valid!