Purpose of the Assignment

The general purpose of this assignment is to explore network programming by building a simplified web client and web server. This assignment is designed to give you experience in:

- writing networked applications
- the socket API in Python
- writing software supporting an Internet protocol

Assigned

Wednesday, September 30, 2020 (please check the main course website regularly for any updates or revisions)

Due

The assignment is due Wednesday, October 21st, 2020 by 11:55pm (midnight-ish) through an electronic submission through the OWL site. If you require assistance, help is available online through OWL.

Late Penalty

Late assignments will be accepted for up to two days after the due date, with weekends counting as a single day; the late penalty is 20% of the available marks per day. Lateness is based on the time the assignment is submitted.

Individual Effort

Your assignment is expected to be an individual effort. Feel free to discuss ideas with others in the class; however, your assignment submission must be your own work. If it is determined that you are guilty of cheating on the assignment, you could receive a grade of zero with a notice of this offence submitted to the Dean of your home faculty for inclusion in your academic record.

What to Hand in

Your assignment submission, as noted above, will be electronically through OWL. You are to submit all Python files required for your assignment (for both your client and your server). If any special instructions are required to run your submission, be sure to include a README file documenting details. (Keep in mind that if the TA cannot run your assignment, it becomes much harder to assign it a grade.)

Assignment Task

You are required to implement in Python a stripped down and simplified web server and web downloader client. Before starting to panic, keep in mind that a web server is in essence a very simple program. In response to a request, it simply reads the contents of a named file and pushes it back over the same connection. That's it! (Also, keep in mind that we've looked at HTTP in our lessons, and could download things over a telnet connection ... and we know from Assignment #1 how simple telnet is!)

For this assignment, you only need to worry about implementing the GET request of HTTP/1.1. In other words, your downloader will open a TCP connection to your server and issue a request like: `GET /index.html HTTP/1.1` with an appropriate `Host:` header. Your server will open this file relative to its current directory, read the contents, and send back the results. The downloader will save the file to disk relative to its own current directory, and close the connection.

When you are done, in theory, you should be able to use a web browser to talk to your server, or use your web downloader client to retrieve documents off of another web site, provided that the web site doesn't require HTTPS for security. (You might run into version compatibility issues since your client and server are trying to force HTTP/1.1, but it should otherwise work ...) Pretty neat!

Some Particulars

Here are some specific requirements and other important notes:

- Your web downloader client and web server must communicate to each other using TCP/IP. Your server only needs to support one connection at a time, and it does need to worry about persistent connections.
- The web downloader client only needs to download a single file and then quit. Your server must support many consecutive connections and not terminate unless the user interrupts (such as using a *Ctrl-C).*
- The server may listen on a fixed port or may use one selected by the operating system (which would have to be printed out so you could instruct the client on where to find the server). If you choose to use a fixed port, you should define it as a constant in the server and it should be greater than 1024. If something else is already using the port you select, you will receive errors when trying to bind to it.
- The server will not require any command line options. The client will require the host name the server is executing on, the port number it is listening to, and the name of file to download as command line options. These can be provided as separate command line options, or you can choose to combine them together as a standard URL passed in as a single command line option, which will have to be split apart by your client.
- All files requested will be stored relative to the current directory of the server when it was executed. So, if a requested file begins with a /, you must add a . to the beginning of the file name to ensure it is accessed in a relative fashion. When the client goes to save the file downloaded, it should strip off the path name and save it with the remaining file name in the current directory of the client. For simplicity, you can assume that there will be no white space or unprintable characters in the file names.
- You only need to support the GET command of HTTP/1.1 in both your client and your server. Your client should send at least a `Host:` option to the request, as that is required by the standard, but your server is free to just ignore such things for now. (If you want to experiment with using some of them, that's up to you.)

- In response to a GET command, your server should respond with a proper HTTP response header when the file exists and can be sent. The first line should be a valid status line like HTTP/1.1 200 OK. You should also include a header line specifying the content length; i.e., the size of the file you are sending back. You should also provide a content type header line for *GIF* and *JPEG* files, to allow browsers to view embedded images and so on. Essentially, you just need to look for a *.gif* or *.jpg* or *.jpeg* file extension at the end of a file name, and add the content type line if the extension matches. Otherwise, do nothing, and the browser will assume it is an HTML file. You are welcome to add content type lines for other kinds of files, if you so choose. You may also add additional header values to return, such as dates, and so on, but they are not required.
- If the file requested in a GET request does not exist, you should return a 404 message to the client. The first line should be a status line like HTTP/1.1 404 Not Found. As a response body, it should include a simple HTML message describing the problem. A sample on can be found here. This can either be hard coded into the server to send, or stored in a file called 404.html and sent using the usual file transfer mechanism, when the original file cannot be found.
- If the client uses a version in its request other than HTTP/1.1, you should return a 505 message to the client. The first line should be a status line like HTTP/1.1 505 Version Not Supported. As a response body, it should include a simple HTML message describing the problem. A sample on can be found here. This can either be hard coded into the server to send, or stored in a file called 505.html and sent using the usual file transfer mechanism.
- If the client issues something other than a GET request, but still claims to be using HTTP/1.1, you should return a 501 message to the client. The first line should be a status line like HTTP/1.1 501 Method Not Implemented. As a response body, it should include a simple HTML message describing the problem. A sample on can be found here. This can either be hard coded into the server to send, or stored in a file called 501.html and sent using the usual file transfer mechanism.
- If the web downloader client received a reply from the server other than a 200 message indicating everything is fine, the client should print the entire message to the screen, and quit without saving anything. If successful, the file should be saved, and the client should exit normally.
- Remember to follow the HTTP formatting conventions as discussed in the course text book and lessons. You must have a blank line between response headers and the file being sent. You must also use *"\r\n"* instead of just *"\n"* to terminate lines in your HTTP messages, to ensure both a carriage return and a line feed.

You are to provide all of the Python code for this assignment yourself. You are **not** allowed to use Python functions to execute other programs for you! All server code files must begin with *server*, and all client files must begin with *client*. All of these files must be submitted with your assignment.

As an important note, marks will be allocated for code style. This includes appropriate use of comments and indentation for readability, plus good naming conventions for variables, constants, and functions. Your code should also be well structured (i.e. not all in the main function).

Please remember to test your program as well, since marks will obviously be given for correctness! You should transfer several different HTML documents, as well as images or other binary files. You can then use diff to compare the original files and the downloaded files to ensure the correct operation of your client and server. You must capture a session of downloading a file from your server to show it in action. (This can be done using the script command or by taking a screen shot or picture with your phone.) For additional testing, you can try to use a standard browser to retrieve documents from your server, and use your client to download documents stored off of other servers.