Project 1: Proxy Web Server — Due: October 13, 2016

The goal of Project 1 is to give you experience working with TCP sockets and creating and parsing HTTP requests and responses. You will implement a simple web client and web proxy using Python. The client will request a webpage via the web proxy, and the web proxy will return the webpage to the client. Project 1 is broken into 4 parts, with each part adding further complexity.

Part 0: Python set-up and echo client/server

Python installation. If you do not already have Python on your machine, you can download it from `https://www.python.org/downloads/`. Please make sure you are using Python 3.x not Python 2.x. For further details about installing and using Python, please see the slides for lecture 7 and the references linked from the class webpage.

Echo client and echo server. To help you get started, you have been given code for a simple echo client and echo server, downloadable from `http://vumanfredi.web.wesleyan.edu/2016/COMP360/lectures/code/`. If you recall we went over this code in class. I recommend using the echo client as the basis for your web client and the echo server as the basis for your web proxy. You should run the code yourself and see what happens. To run the code, open two terminal windows. In one terminal, start the echo server with `python echo_server.py`. In the other window run the echo client with `python echo_client.py`. You should see that the "Hello world" string in `echo_client.py` code sent to the echo server is echoed back and appears in the terminal window where you ran the echo client.

Threading. The echo server is multi-threaded. This means that rather than blocking on a client while that client is being served, a thread is spawned off the main thread of execution of the echo server process to serve each client that connects to the server. This lets the main execution thread of the server continue to listen for new client connections, while clients are being served. If multiple clients connect to the server simultaneously, then each would be served without blocking the other. You should not need to modify the threading code. As an aside, Python does not have true multi-threading in that the use of threads will actually make the code run slower rather than faster (e.g., because separate threads cannot run in parallel on separate cores). The benefit of threading here is that it allows for a simpler server (or proxy) implementation.

Part 1: URLs and GET requests

You will create the web client and web proxy according to the setup shown in Figure 1. In Figure 1, the client and web proxy both run on your local machine using address `localhost`. The web server is any website on the Internet that accepts `HTTP` connections on port `80`. Note that many websites only accept `HTTPS` connections on port `443`: you will not want to use such websites for your testing. Your web proxy will listen on port `50008` for connections from a web client.

Web client. The goal of the web client is to connect to a URL, such as `www.nytimes.com` as in Figure 1 or, for instance, `www.wesleyan.edu/mathcs/index.html`. This URL should be passed to the client program via the command-line when the client starts up. For testing, you may want to use a hard-coded default URL in the client code. To connect to the web proxy, the client will connect to the TCP socket on which the web proxy is listening. The client will then send an HTTP GET request over this socket to the web proxy. Suppose the client wishes to connect to `http://www.wesleyan.edu/mathcs/index.html`. Then the client GET request will have the following format, where `\r\n` represents the carriage return and new line characters. The URL information contained in the host and path in the GET request indicates the desired object to download, as well as the host to which the web proxy will need to connect.

```
GET /mathcs/index.html HTTP/1.1\r\n
Host: www.wesleyan.edu\r\n\r\n
```

Web proxy. The web proxy, upon receipt of this GET request, will parse out the hostname and open a TCP connection to that host. The web proxy will then forward the GET request to the host over this

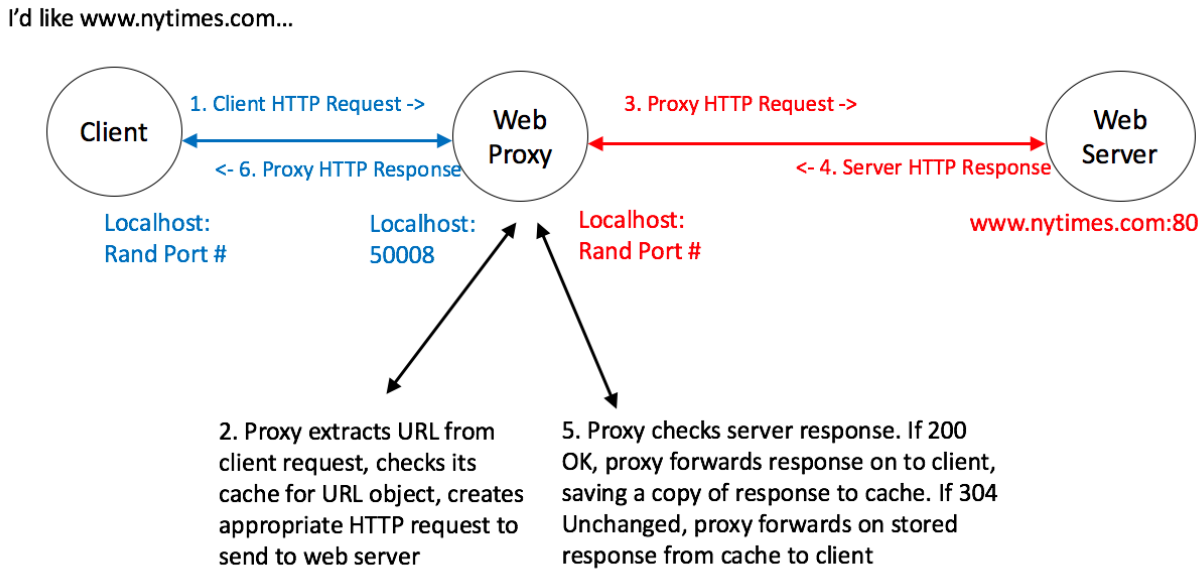

Figure 1: Architecture for Project 1.

TCP connection and receive back the host's HTTP response. The web proxy will then forward the received response back to the client, over the TCP connection that the web proxy has with the client.

Part 2: Proxy cache

Part 2 is an intermediate step on the way to making the web proxy more realistic. The purpose of Part 2 is simply to store the responses from the various web servers to which clients connect.

What to cache. An HTTP response message has the format below. You may additionally see the header line, `Last-Modified:<date>` in the HTTP response. In a real web proxy, you would store only the entity-body along with the URL and date. However, for simplicity, and so that you don't have to parse the HTTP responses, you will save the entire HTTP response in your cache. You will want to keep track of the HTTP response itself, the URL it is associated with, and the `Last-Modified` date returned in the response indicating when it was last updated (or simply the `Date` value if there is no `Last-Modified field`). This information will be used in Part 3 of the project.

```
HTTP/1.1 200 OK\r\n
Date: Tue, 20 Sep 2016 18:27:37 GMT \r\n
Connection: Close \r\n
Content-Type: text/plain \r\n
Content-Length: 5430 \r\n
Last-Modified: Sun, 18 Sep 2016 20:43:27 GMT\r\n\r\n
[ Entity-body of length 5430 bytes ]
```

Cache data structure. You are free to choose whatever approach you'd like to organize your cache, which might range from a simple dictionary data structure, to storing webpages in files. I suggest starting with something very simple like a dictionary, so that you can move on to Part 3.

Part 3: Conditional GET requests

Rather than have the web proxy automatically forward on the client's GET request to the web server, the web proxy will now forward on a modified version of the GET request. How the request is modified depends on whether the webpage object for the desired URL is stored in the proxy cache. If the object for the desired URL *is not* stored in the proxy cache, the web proxy forwards on the client's GET request unmodified. If the object for the desired URL *is* stored in the proxy cache, the web proxy turns the client's GET request into a conditional GET request, using the date information stored in the cache. A conditional HTTP GET request has the following format:

```
GET /mathcs/index.html HTTP/1.1\r\n
Host: www.wesleyan.edu\r\n
If-Modified-Since: Sun, 18 Sep 2016 20:43:27 GMT\r\n\r\n
```

In the response to the conditional GET request, one of two things may occur:

1. **The URL object has changed, or the `If-Modified-Since` is ignored.** In this case, the web server's response will be as in Part 1, and the web proxy forwards the web server's HTTP response directly to the client.
2. **The URL object has not changed.** The web server will return a `304 Not Modified` status code in its response rather than a `200 OK` status code. In this case what the web proxy has stored in its cache is what is returned to the client. The web proxy creates an HTTP response containing the information in its cache, and forwards this response to the client. If the web proxy is storing entire responses in its cache, rather than just the entity-body, as you will be doing, then the web proxy simply returns what is stored in its cache, which should be a validly formatted HTTP Response.

Several websites you might wish to use to test conditional requests include `www.mit.edu` and `www.cyberciti.biz`. I recommend also testing with a URL such as `www.wesleyan.edu/mathcs/index.html` which includes a pathname and object.

Part 4 (10 points): Testing with a web browser

In Part 4, you will test your web proxy using a web browser as a client instead of your web client. To do this, choose your favourite browser and change the proxy settings to use the web proxy listening on `localhost:50007`: i.e., to use your web proxy.

Pathname parsing. When a web browser uses a proxy, it includes the full URL ncluding the hostname as the path in the GET line of the request. This pathname, however, will not be parseable by web servers. Therefore, the web proxy must now check the pathname it is given, identify whether it is a complete URL, and if so create a new HTTP request that includes only the path and object portion of the URL in its GET line.

What to submit

- All of your python files. Your python code should include a header at the top of each file describing what the file does and your python code itself should be well-commented, check for exceptions and shut down nicely (e.g., clean up sockets and other state when the web client or web proxy terminates for some reason).
- Readme file indicating how to run your code and listing all of your python files and the purpose of each file.

- A write-up describing and motivating your program design, a list of web sites for which your program works, a list of example websites for which your program does not work (if any) along with an explanation why. For at least one test case for which your program works, include example screen shots of the output. I do not expect that your code will work with all web servers, since, for instance, some web servers do not use utf-8 encoding, and you are not fully parsing responses. However, in such cases you should catch whatever error is shown and continue cleanly or shutdown. Also include any test cases that cause the web client or web proxy to crash.

How to submit

On wesfiles, I have created directories for each student in the class, named according to email usernames, and have set permissions so that only the corresponding student can access the folder. You should upload your project code and write up to your folder. Since you will be reusing the folder for all project submissions, I suggest creating sub-folders for each project and puting your code and write-up for Project 1 in the Project 1 sub-folder. The path to your folder is as follows, substituting your email username for USERNAME.

`https://wesfiles.wesleyan.edu/home/vumanfredi/web/2016/COMP360/submissions-proj/USERNAME`

Grading policy

Code

(30 points) Successful fetch of a webpage using your client.

(20 points) Successful reception of 302 Not Modified response.

(10 points) Successful fetch of a webpage using a browser.

(10 points) Documentaion in code.

(10 points) Overall code design.

(5 points) Readme.

Design document

(5 points) Description of design.

(10 points) Description of test cases.

I recommend writing out pseudocode for what you need to do, then adding comments for the pieces to fill in. That way, if something isn't working or you don't have time to finish something, I can see what you were trying to do and possibly give you partial credit.

References

You may find the following helpful when looking at HTTP requests and responses and working with conditional HTTP requests.

RFC 7232: Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. `https://tools.ietf.org/html/rfc7232`

RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. `https://tools.ietf.org/html/rfc2616`