Module 2: Lab00 Implementing a Basic Web Server

Grading Criteria

Total Points: 70 + 30 extra credit points available

You may only use your own original Python libraries and the `thread` ***and*** `socket` ***Python libraries.***

☐ **[2.5 points]** Your submission is labeled as "cs460_module02_lab00_[nau_id]_[lastname]_[firstname].zip." For example, if I were to submit a file, it would be labeled as cs460_module02_lab00_mv668_vigil-hayes_morgan.zip. **FAILURE TO COMPLY WITH THIS STEP CAN LEAD TO A ZERO GRADE.**

☐ **[7.5 points]** Your submission files are in the correct format. **FAILURE TO COMPLY WILL RESULT IN A ZERO GRADE.**

- ☐ [2.5 points] All files must be submitted as a single ZIP file
- ☐ [2.5 points] Server code is all contained in a file called [nau_id]_web_server.py
- ☐ [2.5 points] This lab manual is contained in a file called [nau_id]_lab_manual.pdf

☐ **[20 points]** Code runs without errors in Unix environment and is highly readable. I will be testing code on Mac OS X and Ubuntu environments.

- ☐ [5 points] XX_web_server.py starts without error and is able to start up a main server thread that handles incoming client requests.
- ☐ [5 points] XX_web_server.py runs using the following command at the terminal:
  `python3 XX_web_server.py [socket]`
- ☐ [5 points] Code is well organized into subroutines (helper functions) that
- ☐ [5 points] I am able to use a simple telnet client to test the server

☐ **[20 points]** Your server correctly complies with RFC 2616

- ☐ [5 points] When the server receives a correctly formatted GET request for the file at the root directory, the server is able to provide a correctly formatted response that results in sending a small bit of HTML with an original message embedded (e.g., "Computer Networks are the best!")
- ☐ [5 points] When the server receives correctly formatted HTTP requests that are NOT GET requests, the server follows RFC 2616 and sends the proper response indicating a lack of server implementation support.

- ☐ [5 points] When the server receives correctly formatted HTTP requests that is a GET request for a file that does not exist in the server's directory, the server needs to send the proper response according to RFC 2616.
- ☐ [5 points] When the server receives an incorrectly formatted HTTP request, it needs to respond according to RFC 2616.

☐ **[20 points]** Part 1 of the Lab Manual (i.e., this document) has been completed correctly and completely

- ☐ [2.5 points] Identify where in RFC 2616 you found the rules for correctly responding to a correctly formatted GET request for the root directory
- ☐ [2.5 points] Show a screenshot of a Wireshark capture of the GET request packet sent to your server and the server response
- ☐ [2.5 points] Identify where in RFC 2616 you found the rules for correctly responding to a correctly formatted GET request for a nonexistent file
- ☐ [2.5 points] Show a screenshot of a Wireshark capture of the GET request for nonexistent file packet sent to your server and the server response
- ☐ [2.5 points] Identify where in RFC 2616 you found the rules for correctly responding to correctly formatted non-GET requests (e.g., POST) for a nonexistent file
- ☐ [2.5 points] Show a screenshot of a Wireshark capture of the correctly formatted non-GET request packet sent to your server and the server response
- ☐ [2.5 points] Identify where in RFC 2616 you found the rules for correctly responding to incorrectly formatted requests
- ☐ [2.5 points] Show a screenshot of a Wireshark capture of the incorrectly formatted request packet sent to your server and the server response

Extra Credit Opportunities

These opportunities are only available if you have completed Part 1 of the Lab Manual. Please check the boxes on this section if you have completed any of the extra credit components; otherwise, I will not grade them and you will not receive extra credit. You may do one, two, or all three options for extra credit.

☐ **[5 points] EC Option 1:** Serve a **file** called index.html and demonstrate that a browser-based client (e.g., Firefox or Chrome) can load it correctly.

- ☐ [1.25 points] Include a file called index.html in your ZIP file
- ☐ [1.25 points] Include well-organized and commented code for your extra credit web server as a file called [nau_id]_web_server_ec1.py. It should be able to run using the following command:

```
python3 XX_web_server_ec1.py  [socket]
```

- ☐ [1.25 points] In Part EC of the lab manual (e.g., this document), provide screenshot of Firefox or Chrome displaying your index.html file as a web page in the browser
- ☐ [1.25 points] In Part EC of the lab manual, provide a screenshot of the GET request that the browser sent to your server for index.html and the server response.

☐ **[5 points] EC Option 2:** Serve many different clients simultaneously using threading. A multithreaded server would allow up to 3 clients (or more!) connect to the server simultaneously and allow them to make HTTP requests simulateously.

- ☐ [2.5 points] Include well-organized and commented code for your extra credit web server as a file called [nau_id]_web_server_ec2.py. It should be able to run using the following command:
  `python3 XX_web_server_ec2.py [socket]`
- ☐ [2.5 points] The server successfully handles at least 3 simultaneous telnet client connections

☐ **[20 points] EC Option 3:** Create a web server that acts as a proxy server* according to RFC 2616. The proxy server should be able to request files from other HTTP-only origin servers on behalf of the client and then forward the responses from the origin server back to the client. The proxy should also cache files that have been requested so that when another client requests the same file, the proxy is able to provide it directly.

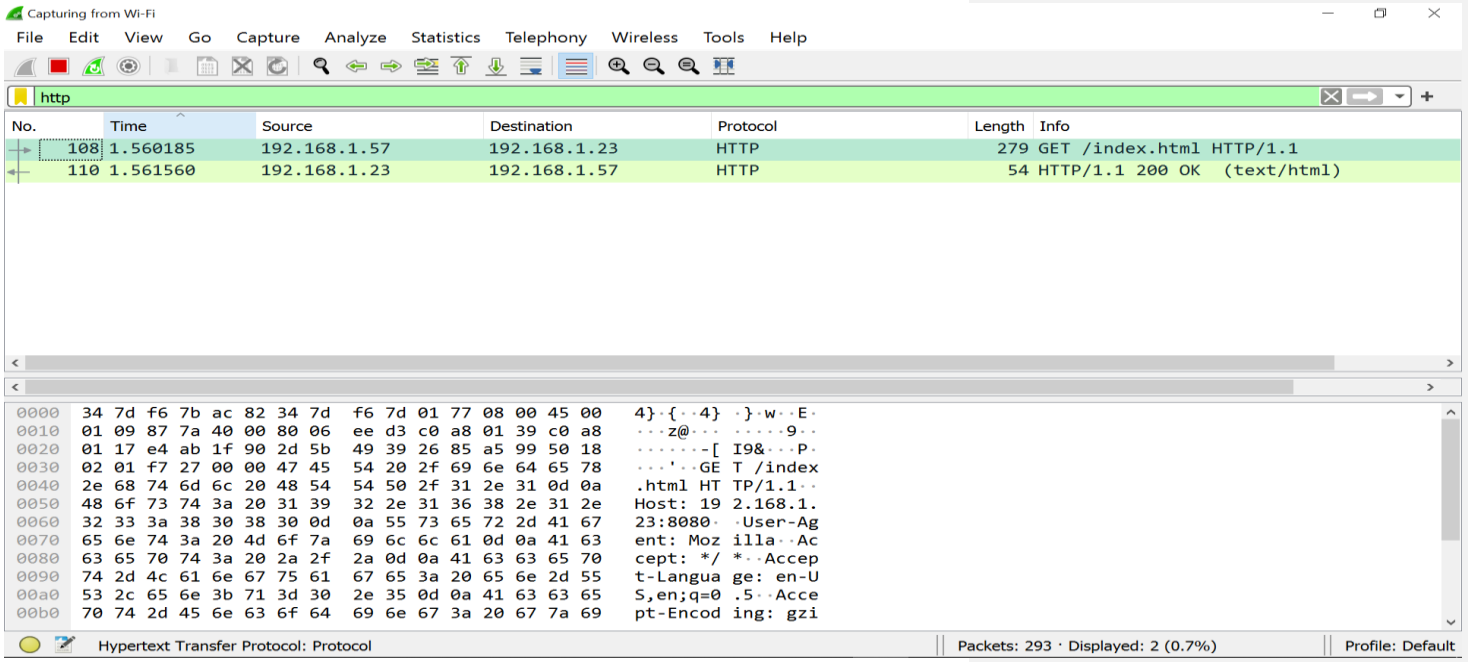
- ☐ [2.5 points] Include a file called [nau_id]_web_server_ec3.py in your ZIP file. It should run using the command:
  `python3 XX_web_server_ec3.py [socket]`
- ☐ [5 points] I should be able to point a Firefox browser to your proxy server and then make a request in my browser to some HTTP-only origin server, and your proxy server should be able to request and deliver the file to my client successfully.
- ☐ [5 points] If I request the same file again, your proxy server should be able to provide the file *without* having to first request it from the origin server.
- ☐ [2.5 points] In Part EC of the lab manual, copy the lines (or screenshot and highlight parts) of RFC 2616 that you used to help you determine how your proxy server should work with clients and origin servers.
- ☐ [5 points] In Part EC of the lab manual (e.g., this document), paste a screenshot of a Wireshark capture that shows the browser requesting a file from the proxy and the proxy requesting the file from the origin server and then forwarding it on to the client according to RFC 2616.

Part 1: Simple Web Server

[2.5 points] Identify where in RFC 2616 you found the rules for correctly responding to a correctly formatted GET request for the root directory

```
10.2.1 200 OK

   The request has succeeded. The information returned with the response
   is dependent on the method used in the request, for example:

   GET    an entity corresponding to the requested resource is sent in
          the response;

   HEAD   the entity-header fields corresponding to the requested
          resource are sent in the response without any message-body;

   POST   an entity describing or containing the result of the action;
```

[2.5 points] Show a screenshot of a Wireshark capture of the GET request packet sent to your server and the server response

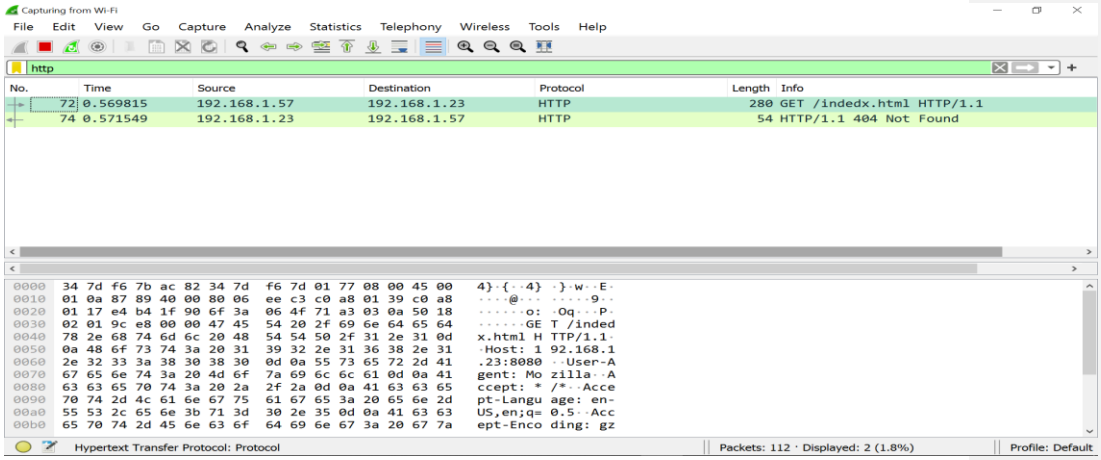


[2.5 points] Identify where in RFC 2616 you found the rules for correctly responding to a correctly formatted GET request for a nonexistent file

10.4.5 404 Not Found

The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent. The 410 (Gone) status code SHOULD be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

[2.5 points] Show a screenshot of a Wireshark capture of the GET request for nonexistent file packet sent to your server and the server response

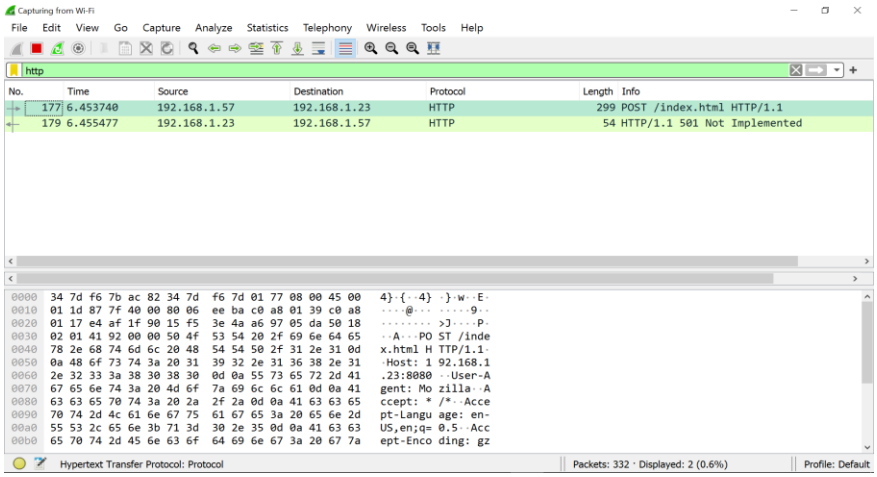


[2.5 points] Identify where in RFC 2616 you found the rules for correctly responding to correctly formatted non-GET requests (e.g., POST) for a nonexistent file

10.5.2 501 Not Implemented

The server does not support the functionality required to fulfill the request. This is the appropriate response when the server does not recognize the request method and is not capable of supporting it for any resource.

[2.5 points] Show a screenshot of a Wireshark capture of the correctly formatted non-GET request packet sent to your server and the server response

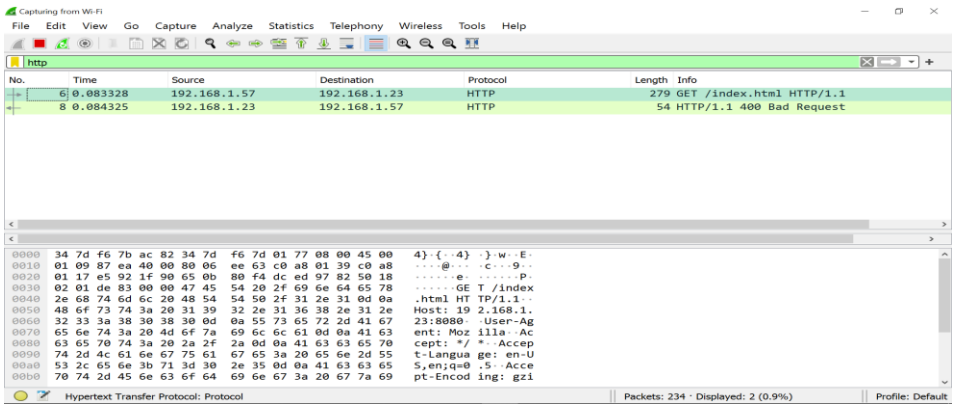


[2.5 points] Identify where in RFC 2616 you found the rules for correctly responding to incorrectly formatted requests

10.4.1 400 Bad Request

The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modifications.

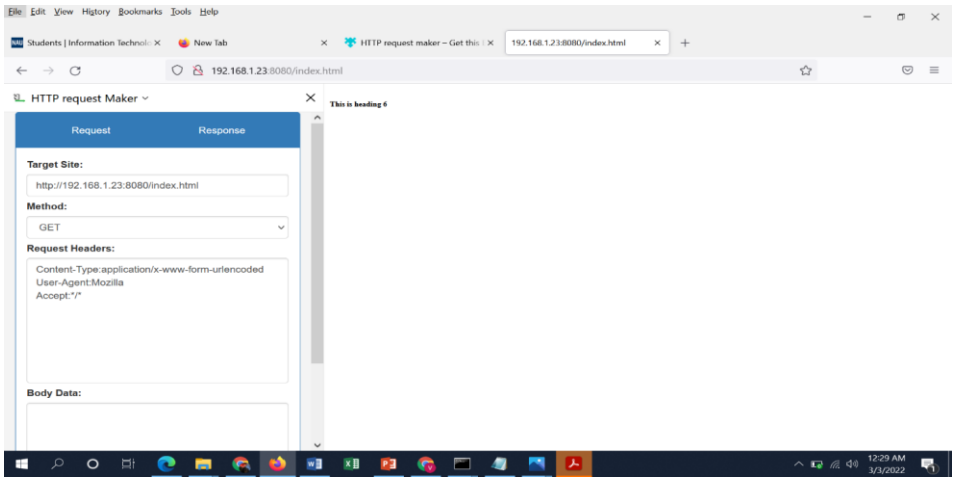
[2.5 points] Show a screenshot of a Wireshark capture of the incorrectly formatted request packet sent to your server and the server response



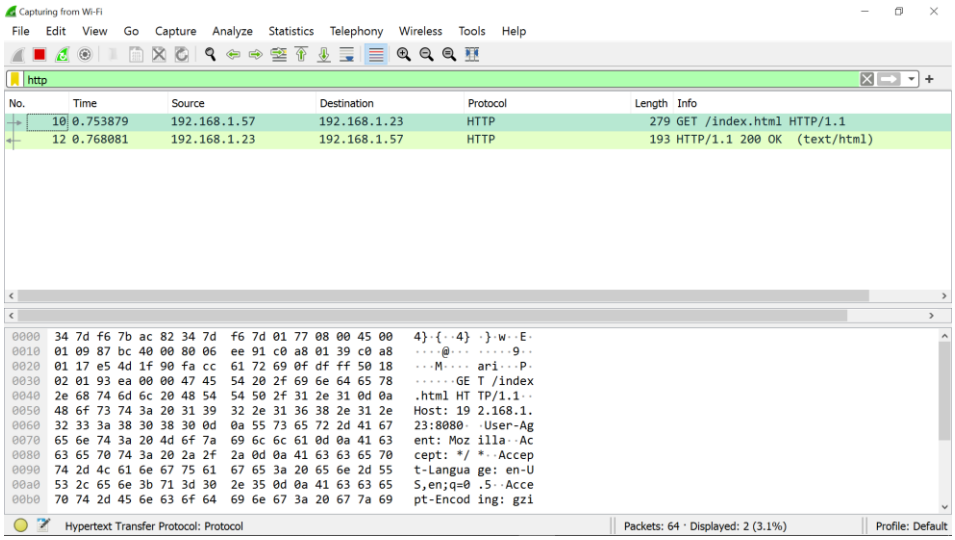
Part EC: Extra Credit Options

EC Option 1

[1.25 points] Provide a screenshot of Firefox or Chrome displaying your index.html file as a web page in the browser.



[1.25 points] Provide a screenshot of the GET request that the browser sent to your server for index.html and the server response.



EC Option 3

[2.5 points] Copy the lines (or screenshot and highlight parts) of RFC 2616 that you used to help you determine how your proxy server should work with clients and origin servers.

[LINES OR SCREENSHOT GOES HERE]

[5 points] In Part EC of the lab manual (e.g., this document), paste a screenshot of a Wireshark capture that shows the browser requesting a file from the proxy and the proxy requesting the file from the origin server and then forwarding it on to the client according to RFC 2616.

[SCREENSHOT GOES HERE]

Appendix A: Hints & Tips

Some background on HTTP

The Hypertext Transfer Protocol (HTTP) is the protocol used for communication on this web: it defines how your web browser requests resources from a web server and how the server responds. For simplicity, in this assignment, we will be dealing only with version 1.1 of the HTTP protocol, defined in detail in RFC 2616. You may refer to that RFC while completing this assignment, but our instructions should be self-contained.

HTTP communications happen in the form of transactions; a transaction consists of a client sending a request to a server and then reading the response. Request and response messages share a common basic format:

An initial line (a request or response line, as defined below)

Zero or more header lines

A blank line (CRLF)

An optional message body.

The initial line and header lines are each followed by a "carriage-return line-feed" (\r\n) signifying the end-of-line.

For most common HTTP transactions, the protocol boils down to a relatively simple series of steps:

1. A client creates a connection to the server.
2. The client issues a request by sending a line of text to the server. This **request line** consists of a HTTP *method* (most often GET, but POST, PUT, and others are possible), a *request URI* (like a URL), and the protocol version that the client wants to use (HTTP/1.1). The request line is followed by one or more header lines. The message body of the initial request is typically empty.

3. The server sends a response message, with its initial line consisting of a **status line**, indicating if the request was successful. The status line consists of the HTTP version (HTTP/1.1), a *response status code* (a numerical value that indicates whether or not the request was completed successfully), and a *reason phrase*, an English-language message providing description of the status code. Just as with the the request message, there can be as many or as few header fields in the response as the server wants to return. Following the CRLF field separator, the message body contains the data requested by the client in the event of a successful request.

4. Once the server has returned the response to the client, it closes the connection.

It's fairly easy to see this process in action without using telnet. From the terminal, type:

```
telnet www.yahoo.com 80
```

This opens a TCP connection to the server at www.yahoo.com listening on port 80 (the default HTTP port). You should see something like this:

```
HTTP/1.1 200 OK

Date: Tue, 16 Feb 2010 19:21:24 GMT

(More HTTP headers...)
```

```
Content-Type: text/html; charset=utf-8


<html><head>

<title>Yahoo!</title>

(More HTML follows)
```

There may be some additional pieces of header information as well- setting cookies, instructions to the browser or proxy on caching behavior, etc. What you are seeing is exactly what your web browser sees when it goes to the Yahoo home page: the HTTP status line, the header fields, and finally the HTTP message body- consisting of the HTML that your browser interprets to create a web page.

Detailed Tips on Implementing a Simple Web Server

Your task is to build a Web server capable of accepting HTTP requests and returning response data to a client. To simplify some of the more complex binary translation aspects of returning data, you will only be asked to return a simple, text-only HTML string. You will only be responsible for implementing the HTTP GET method. All other request methods received by the proxy should elicit an error code based on RFC 2616.

For ease of grading, the assignment will be implemented in Python using Python3.6. It should run without errors on Ubuntu 16.04. The program should take as its first argument a port to listen from. **Don't use a hard-coded port number.**

You shouldn't assume that your server will be running on a particular IP address, or that clients will be coming from a pre-determined IP.

Listening

When your server starts, the first thing that it will need to do is establish a socket connection that it can use to listen for incoming connections. Your proxy should listen on the port specified from the command line and wait for incoming client connections.

Once a client has connected, the server should read data from the client and then check for a properly-formatted HTTP request. You will need to ensure that the server receives a request that contains a valid request line:

```
<METHOD> <URL> <HTTP VERSION>
```

All other headers just need to be properly formatted according to RFC 2616.

An invalid or inseverable request from the client should be answered with an appropriate error code. Similarly, if headers are not properly formatted for parsing, your proxy should also generate a type-400 message.

Parsing

You will need to create a function (or even a small library if you want) that parses the HTTP Request to ensure they contain a valid request line. An invalid request from the client should be answered with an appropriate error code. Similarly, if headers are not properly formatted for parsing, your proxy should also generate a type-400 message.

Testing correctness

To test the correctness of your program, start by running your Web server with the following command (assuming a Pythonic implementation):

```
python [YOUR_NAU_ID]_web_server.py port
```

As a basic test of functionality, try requesting a page using telnet:

```
telnet localhost <port>
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
GET http://127.0.0.1/path_to_html_file HTTP/1.1
```

Commented [MOU1]: My program file would be mv668_webserver.py

Commented [MOU2]: This is a port number that you enter. I suggest something like 8080 or 1010

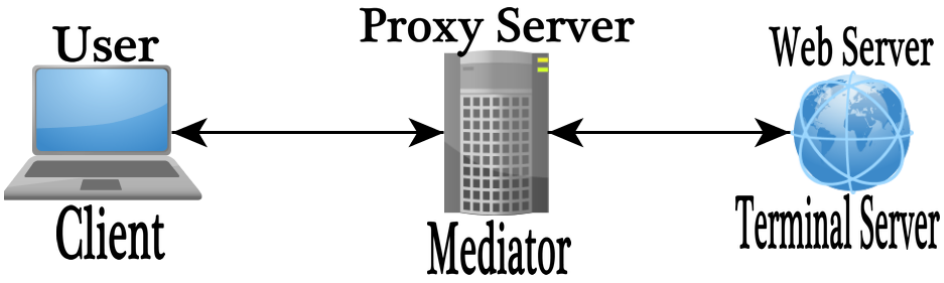
Commented [MOU3]: You can install telent on Ubuntu 16.04 using: sudo apt-get install telnet

Commented [MOU4]: Use the same port number as what you ran as an argument to your Web server.

Commented [MOU5]: This should be the Unix formatted path to where you stored the HTML file you downloaded from BbLearn.

About HTTP Proxies

Ordinarily, HTTP is a client-server protocol. The client (usually your web browser) communicates directly with the server (the web server software). However, in some circumstances it may be useful to introduce an intermediate entity called a proxy. Conceptually, the proxy sits between the client and the server. In the simplest case, instead of sending requests directly to the server the client sends all its requests to the proxy. The proxy then opens a connection to the server and passes on the client's request. The proxy receives the reply from the server, and then sends that reply back to the client. Notice that the proxy is essentially acting like both a HTTP client (to the remote server) and a HTTP server (to the initial client).



Why use a proxy? There are a few possible reasons:

- **Performance:** By saving a copy of the pages that it fetches, a proxy can reduce the need to create connections to remote servers. This can reduce the overall delay involved in retrieving a page, particularly if a server is remote or under heavy load.
- **Content Filtering and Transformation:** While in the simplest case the proxy merely fetches a resource without inspecting it, there is nothing that says that a proxy is limited to blindly fetching and serving files. The proxy can inspect the requested URL and selectively block access to certain domains, reformat web pages (for instances, by stripping out images to make a page easier to display on a handheld or other limited-resource client), or perform other transformations and filtering.
- **Privacy:** Normally, web servers log all incoming requests for resources. This information typically includes at least the IP address of the client, the browser or other client program that they are using (called the User-Agent), the date and time, and the requested file. If a client does not wish to have this personally identifiable information recorded, routing HTTP requests through a proxy is one solution. All requests coming from clients using the same proxy appear to come from the IP address and User-Agent of the proxy itself, rather than the individual clients. If a number of clients use the same proxy (say, an entire business or university), it becomes much harder to link a particular HTTP transaction to a single computer or individual.

Tips for Proxy Server

Once the proxy receives a valid HTTP request, it will need to parse the requested URL. The proxy needs at least three pieces of information: the requested host, port, and path. You will need to parse the absolute URL specified in the given request line. You will need to use the path to identify the data that needs to be sent back to the client. If the file does not exist, you should answer the request with an appropriate error code.

Since many origin servers only use HTTPS these days, it is useful to focus on testing your proxy server's abilities to provide access to older servers, such as these:

https://www.spacejam.com/

https://whynohttps.com/ (see individual sites listed here)

As I find other appropriate servers, I will post them to Slack.

Listening

When your proxy starts, the first thing that it will need to do is establish a socket connection that it can use to listen for incoming connections. Your proxy should listen on the port specified from the command line and wait for incoming client connections.

Once a client has connected, the proxy should read data from the client and then check for a properly-formatted HTTP request -- but don't worry, we have provided you with libraries that parse the HTTP request lines and headers. Specifically, you will need to ensure that the server receives a request that contains a valid request line:

```
<METHOD> <URL> <HTTP VERSION>
```

All other headers just need to be properly formatted:

```
<HEADER NAME>: <HEADER VALUE>
```

In this assignment, client requests to the proxy must be in their absolute URI form.

```
GET http://www.cs.princeton.edu/index.html HTTP/1.1
```

Your browser will send absolute URI if properly configured to explicitly use a proxy (as opposed to a transparent on-path proxies that some ISPs deploy, unbeknownst to their users). On the other form, your proxy should issue requests to the webserver properly specifying *relative* URLs, e.g.,

```
GET /index.html HTTP/1.1
```

Parsing

You will need to create a function (or even a small library if you want) that parses the HTTP Request to ensure they contain a valid request line. An invalid request from the client should be answered with an appropriate error code, i.e. "Bad Request" (400) or "Not Implemented" (501) for valid HTTP methods other than GET. Similarly, if headers are not properly formatted for parsing, your proxy should also generate a type-400 message.

Once the proxy receives a valid HTTP request, it will need to parse the requested URL. The proxy needs at least three pieces of information: the requested host, port, and path. You will need to parse the absolute URL specified in the given request line. If the hostname indicated in the absolute URL does not have a port specified, you should use the default HTTP port 80.

Returning Data to the Client

After the response from the remote server is received, the proxy should send the response message as-is to the client via the appropriate socket. To be strict, the proxy would be required to ensure a `Connection: close` is present in the server's response to let the client decide if it should close it's end of the connection after receiving the response. However, checking this is not required in this assignment for the following reasons. First, a well-behaving server would respond with a `Connection: close` anyway given that we ensure that we sent the server a close token. Second, we configure Firefox to always send a `Connection: close` by setting keepalive to false. Finally, we wanted to simplify the assignment so you wouldn't have to parse the server response.

Testing Proxy Correctness

To test the correctness of your program, start by running your Web server with the following command (assuming a Pythonic implementation):

```
python [YOUR_NAU_ID]_web_server_ec3.py port
```

As a basic test of functionality, try requesting a page using telnet:

```
telnet localhost <port>
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
GET http://www.google.com/ HTTP/1.1
```

If your proxy is working correctly, the headers and HTML of the Google homepage should be displayed on your terminal screen. Notice here that we request the absolute URL (`http://www.google.com/`) instead of just the relative URL (`/`). A good sanity check of proxy behavior would be to compare the HTTP response (headers and body) obtained via your proxy with the response from a direct telnet connection to the remote server. Additionally, try requesting a page using telnet concurrently from two different shells to test simultaneous service provided to multiple clients going to multiple servers.

Commented [MOU6]: For example, my program script would read: mv668_proxy.py

Commented [MOU7]: This is a port number that you enter. I suggest something like 8080 or 1010

Commented [MOU8]: You can install telent on Ubuntu 16.04 using: sudo apt-get install telnet

For a slightly more complex test, you can configure Firefox to use your proxy server as its web proxy as follows:

1. Go to the 'Edit' menu.
2. Select 'Preferences'. Select 'Advanced' and then select 'Network'.
3. Under 'Connection', select 'Settings...'.
4. Select 'Manual Proxy Configuration'. If you are using localhost, remove the default 'No Proxy for: localhost 127.0.0.1". Enter the hostname and port where your proxy program is running.
5. Save your changes by selecting 'OK' in the connection tab and then select 'Close' in the preferences tab.