Practical Assignment-1: HTTP Client - Server

The goal of this assignment is to gain experience in application layer network programming through Python Sockets and get familiarized with the basics of distributed programming. Specifically, this assignment will help to understand the Hypertext Transfer Protocol (HTTP), which is one of the widely used protocols on the Internet, and the operation of webpage translators. This project illustrates that correctly following protocol standards allows your programs to interact with each other but also with other people's programs using application layer networking concepts.

Assignment Description

Part 1 - HTTP Client

In the first part, you will build an HTTP client that that can connect to HTTP servers using TCP sockets. Your HTTP client should support HTTP version 1.1. The program should accept **three input arguments**:
1. <u>HTTP Command:</u> HEAD, GET, PUT or POST.
2. <u>URI:</u> e.g. http://www.example.com or http://www.google.com
3. <u>Port:</u> the default port number of 80 should be used to connect with the server.

Given the above arguments, your HTTP client should construct a valid HTTP request, which it will send to the HTTP server.
For the first part, you can test your client implementation against public HTTP servers using any URI's, such as http://www.example.com or http://www.google.com.

After each request, your client should create and store **the body** of response it received from the server in an HTML file.
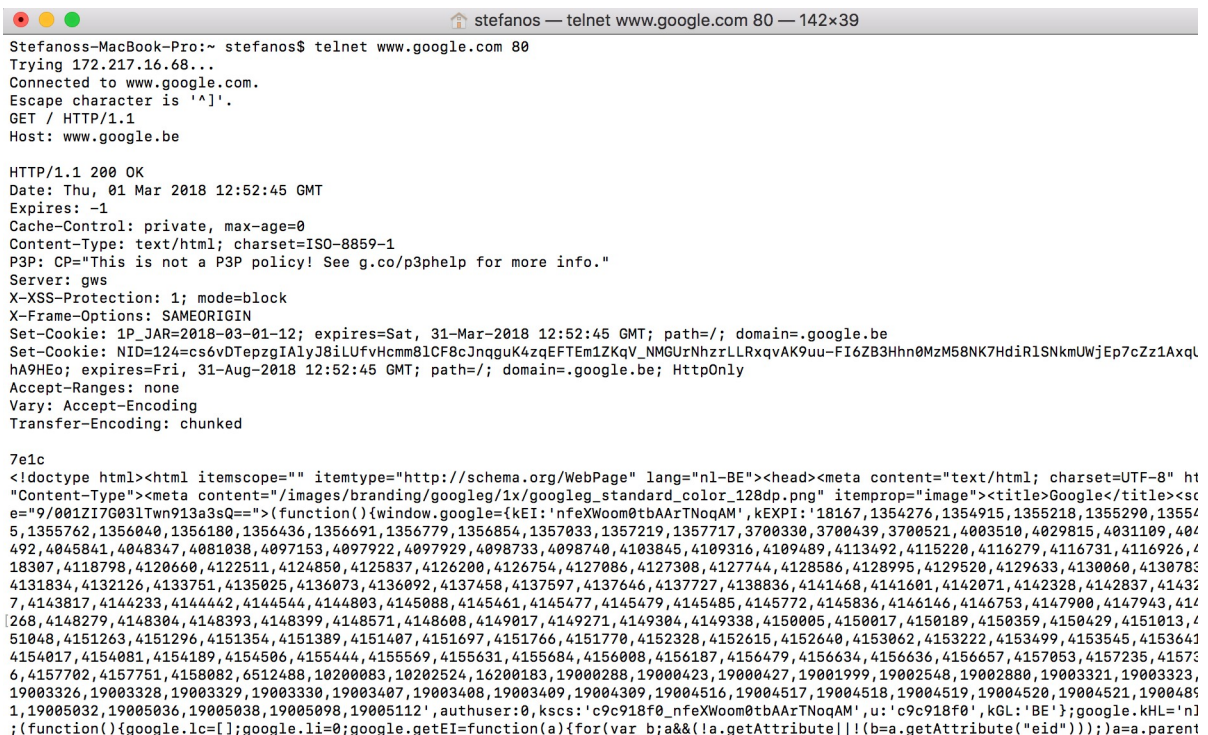You may also display the response on the terminal for debugging.
The size of the body is indicated by the HTTP server with one of two headers: 'Content-Length' or 'Transfer-Encoding: chunked', which determine how your client should read the body. Your client needs to support **both** of these two headers.

When you retrieve a webpage from the server, you should scan the HTML file and check for embedded objects, such as images. You can choose any parser to scan the HTML file, but you are not allowed to use any API's from the parser library to retrieve the images. If you find embedded images in the HTML file, you should use the GET command to retrieve those images as well. When these embedded images are also stored on the same server, you should use the existing socket to request these im- ages. Otherwise, you should connect with a new socket to the server(s) that have theseimages. The retrieved images should be stored locally.
Note#1: the HTML specifies the paths to these images, as stored on the server. You may alter these paths to point to the right directory where your client stores the retrieved images. This is the ONLY part of the HTML that you are allowed to modify.

For PUT and POST commands, your client should prompt the user for a string, read the given string and send that to the HTTP server. These two commands will be testedwith your HTTP server program (see Part 2).

For debugging purposes, we strongly recommend to use "telnet" [6]. Telnet enables you to connect to a remote server and issue HTTP commands through your terminal, where you can view the HTTP response of the server. An example is shown below:


```
                        🏠 stefanos — telnet www.google.com 80 — 142×39
Stefanoss-MacBook-Pro:~ stefanos$ telnet www.google.com 80
Trying 172.217.16.68...
Connected to www.google.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.google.be

HTTP/1.1 200 OK
Date: Thu, 01 Mar 2018 12:52:45 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2018-03-01-12; expires=Sat, 31-Mar-2018 12:52:45 GMT; path=/; domain=.google.be
Set-Cookie: NID=124=cs6vDTepzgIAlyJ8iLUfvHcmm8lCF8cJnqguK4zqEFTEm1ZKqV_NMGUrNhzrLLRxqvAK9uu-FI6ZB3Hhn0MzM58NK7HdiRlSNkmUWjEp7cZz1AxqU
hA9HEo; expires=Fri, 31-Aug-2018 12:52:45 GMT; path=/; domain=.google.be; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked

7e1c
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="nl-BE"><head><meta content="text/html; charset=UTF-8" ht
"Content-Type"><meta content="/images/branding/googleg/1x/googleg_standard_color_128dp.png" itemprop="image"><title>Google</title><sc
e="9/001ZI7G03lTwn913a3sQ==">(function(){window.google={kEI:'nfeXWoom0tbAArTNoqAM',kEXPI:'18167,1354276,1354915,1355218,1355290,13554
5,1355762,1356040,1356180,1356436,1356691,1356779,1356854,1357033,1357219,1357717,3700330,3700439,3700521,4003510,4029815,4031109,404
492,4045841,4048347,4081038,4097153,4097922,4097929,4098733,4098740,4103845,4109316,4109489,4113492,4115220,4116279,4116731,4116926,4
18307,4118798,4120660,4122511,4124850,4125837,4126200,4126754,4127308,4127744,4128586,4128995,4129520,4129633,4130060,413078
4131834,4132126,4133751,4135025,4136073,4136092,4137458,4137597,4137646,4137727,4138836,4141468,4141601,4142071,4142328,4142837,41432
7,4143817,4144233,4144442,4144544,4144803,4145088,4145461,4145477,4145479,4145485,4145772,4145836,4146146,4146753,4147900,4147943,414
[268,4148279,4148304,4148393,4148399,4148571,4148608,4149017,4149304,4149338,4150005,4150017,4150189,4150359,4150429,4151013,4
51048,4151263,4151296,4151354,4151389,4151407,4151697,4151766,4151770,4152328,4152615,4152640,4153062,4153222,4153499,4153545,415364
4154017,4154081,4154189,4154506,4155444,4155569,4155631,4155684,4156008,4156187,4156479,4156634,4156636,4156657,4157053,4157235,41573
6,4157702,4157751,4158082,6512488,10200083,10202524,16200183,19000288,19000423,19000427,19001999,19002548,19002880,19003321,19003323,
19003326,19003328,19003329,19003330,19003407,19003408,19003409,19004309,19004516,19004517,19004518,19004519,19004520,19004521,1900489
1,19005032,19005036,19005038,19005098,19005112',authuser:0,kscs:'c9c918f0_nfeXWoom0tbAArTNoqAM',u:'c9c918f0',kGL:'BE'};google.kHL='nl
;(function(){google.lc=[];google.li=0;google.getEI=function(a){for(var b;a&&(!a.getAttribute||!(b=a.getAttribute("eid")));)a=a.parent
```

To summarize, your HTTP client is successfully implemented when it can correctly send HEAD, GET, PUT, POST HTTP commands. GET involves retrieving any web-page from the Internet, along with its embedded images. You can test if GET works correctly by double-clicking the html file in which you stored the body of the response. If your client retrieves and stores all contents correctly, the retrieved webpagewill appear identical to what you would see if you used your web-browser to retrieve it.

Note#2: In principle, your HTTP client should be able to retrieve any webpage from the Internet. In the demo, at least one of the webpages we will ask you to retrieve will be from the list given below. Naturally, you should use the following URI's to test your client program:

- o   www.example.com
- o   www.google.com
- o   www.tcpipguide.com

- ○ www.tinyos.net
- ○ www.linux-ip.net

Part 2 - HTTP Server

In the second part, you will implement your own HTTP server, which should host a simple web page on your local machine. You can make a simple .html file, or use one of the web pages by your client. In any case, the stored webpage should also contain **at least one** embedded image, which is stored in the same directory.

Your HTTP server should be **multi-threaded** to support multiple clients at the same time. You can create multiple tabs in your web browser to test this with your HTTP server. The server should support the following client operations: HEAD, GET, PUT and POST. Your HTTP Server should handle multiple clients at the same time. Create multiple tabs in your web browser to test this with your HTTP server.

A client should be able to retrieve the web page hosted on your server, along with the embedded image(s). The client can either be your own HTTP Client from Part 1, or any third party client such as Firefox or Chrome. If you have followed the protocol standards correctly, any client will be able to interact with your server.

In the case of PUT and POST commands, your HTTP server should store the data received from clients in a text file, stored in the same directory.
For the PUT command, the user input should be stored in a new text file on the server. For the POST command, the user input should be appended to an existing file on the server. If the file does not exist, then the file should be created. The name of the file is specified in the HTTP command as part of the path.

As your server needs to support HTTP version 1.1, it should use persistent connections and support the *if-modified-since* HTTP header. Note that the host header field is mandatory for HTTP version 1.1.

Finally, you should at least support the following status codes on your server:
    200 OK
    404 Not Found
    400 Bad Request
    500 Server Error
    304 Not Modified

Along with the status codes, the server should send the date, content type and content length headers to the client.
The server should respond with the "400: Bad Request" status code when the HTTP Client does not include the host header in its request for HTTP version 1.1.

Practical Guidelines

The assignment can be done in groups of two, however, you will be individually graded. Your grade will be determined based on the demonstration of your submitted project and a Q&A session, whose details will follow later. The submission deadline **TBA**, any changes you make beyond the deadline will not be considered. We expect every student to be able to defend the code that they submitted.

You will implement this assignment using sockets programming in **Python v3.x**. You should use the "socket" to implement the communication between client/server. You should use the "threading" library to implement your multi-threaded HTTP server. **You are NOT allowed to use libraries that implement the HTTP Client / Server functionality for you. This includes (but is not limited to) the following libraries:**
• http.server,
• http.client,
• SimpleHTTPServer,
• requests,
• httplib,
• urllib.

If you are uncertain about whether or not you are allowed to use a certain library, please ask on the "Discussion Board" on Toledo, or during one of the workshop sessions.

Finally, you should document your code, and aim for a good design. You should be able to motivate your decisions, and explain what your code does, e.g. how a certain method works in your code or to demonstrate certain functions. Keep that in mind when copy pasting code bits that you may find online, e.g. Stack Overflow.
We will test your insight and understanding of HTTP during the demonstration by asking you additional questions besides the code.

Marking Specifications
The following specifications will be used during the marking session.

HTTP Client Marking (worth 10 out of 20 marks)

Mark	Expected Functionality
Below 2	Client is not functional or sufficiently demonstrated.
2-4	Client failed to work with all web pages.

4-6	Client works with all the web pages supporting HTTP 1.1, but does not store the web page and images correctly.
7-9	Client works correctly with HTTP 1.1 and the student has understood the protocol clearly.
10	All of the previous, with well documented code and elegant design.

HTTP Server Marking (worth 10 out of 20 marks)

Mark	Expected Functionality
Below 2	Server is not functional or sufficiently demonstrated.
2-3	Server handles only one client.
4-6	Server supports HTTP 1.1, but has threading problems.
7	As above with the correct use of threading and proper support for status codes.
8-9	As above, server successfully serves the provided web page and embedded image(s), retrieved using your HTTP client program.
10	As above with documented code and elegant design.

References:

Python Sockets. https://www.geeksforgeeks.org/socket-programming-python/
Very short HTTP intro: https://learn.onemonth.com/understanding-http-basics/
More in-depth: https://www.tutorialspoint.com/http/http_quick_guide.htm
From an implementer point-of-view: http://www.jmarshall.com/easy/http/
The HTTP Specification: HTTP 1.0 (RFC 1945).
The HTTP Specification: HTTP 1.1 (RFC 2616).
https://en.wikipedia.org/wiki/Telnet