Lab 03: Untangling the Web

Due date: **September 29, 2019**
Points: **200**
Questions: **12**
Programs: **2**

Checklist Feel free to make use of this checklist as your own sanity check.	
Complete [30 pts]	All questions are answered. Each complete question is worth 0.83 points [10 pts] ☐ Source code for simple Web server. This includes a Makefile if compilation is required [5 pts] ☐ Source code for multithreaded Web proxy. This includes a Makefule if compilation is required [5 pts] ☐ README for the simple Web server that describes the function of the programs, how to compile, how to run, and expected outputs [5 pts] ☐ README for the multithreaded Web server that describes the function of the programs, how to compile, how to run, and expected outputs [5 pts] ☐
Correct [150 pts]	Responses demonstrate that you have thought about your answer and that your answer is unique to you as an individual. Each correct question is worth 2.5 points. [30 pts] ☐ Source code for simple Web server compiles and runs without errors. It processes simple HTTP GET requests appropriately and responds with the correct error codes when it receives unexpected output [50 pts] ☐ Source code for multithreaded Web proxy compiles and runs without errors. It processes simple HTTP GET requests appropriately and responds with the correct error codes when it receives unexpected output. It is able to handle at least 2 simultaneous requests [70 pts] ☐
Name [20 pts]	Your name and NAU email are on the assignment [5 pts] ☐ The submitted **question file (this file)** has your first and last name in the correct format: lastname_firstname_cs460_lab03.pdf [5 pts] ☐ The submitted **basic web server** program files are submitted as a ZIP file that is labelled as: lastname_firstname_cs460_lab03_web_server.zip [5 pts] ☐ The submitted **web proxy server** program files are submitted as a ZIP file that is labelled as: lastname_firstname_cs460_lab03_web_proxy_server.zip [5 pts] ☐ *****Failure to comply with these formatting requirements will result in a zero grade for your submission*****

The world wide web (or just the Web) is one of the most well-known and most-used Internet applications—and arguably one of the 20th century's most important advancements. In this lab, we will begin untangling the Web by having you write a small Web server and then a more complex multithreaded Web proxy server and examine and compare what they look like under the hood using Wireshark.

For the coding part of this assignment, you are only allowed to use standard packages in addition to the following unless you have received explicit documentation from Dr. Vigil-Hayes.

Python	Your own original libraries, thread, socket
C/C++	Your own original libraries, pthread.h, socket.h

Part 0: Identification

Name:	Stavros Triantis
NAU Email:	sst46@ nau.edu

Part 1: Write a simple Web server

In this part of the assignment, you will implement a simple Web server that listens for HTTP GET requests, processes them, and returns the requested resources. This will give you a chance to get to know one of the most popular application protocols on the Internet -- the Hypertext Transfer Protocol (HTTP) -- and give you an introduction to the Berkeley sockets API.

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 1.0 of the HTTP protocol, defined in detail in RFC 1945. 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 (important sections of RFC 1945 are in parenthesis):

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/1.0). The request line is followed by one or more header lines. The message body of the initial request is typically empty. (5.1-5.2, 8.1-8.3, 10, D.1)

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/1.0), 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. (6.1-6.2, 9.1-9.5, 10)
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 a web browser. 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/1.0 200 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. You may notice here that the server responds with HTTP 1.1 even though you requested 1.0. Some web servers refuse to serve HTTP 1.0 content.

Assignment Details

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 page (provided in the BbLearn folder for this lab). You will only be responsible for implementing the HTTP GET method. All other request methods received by the proxy should elicit a "Not Implemented" (501) error (see RFC 1945 section 9.5 - Server Error).
This assignment can be completed in either Python, C, or C++. It should compile and run without errors on Ubuntu 16.04. If you are using C or C++ you MUST submit a Make file that compiles your code and creates a binary titled [YOUR_NAU_ID]_webserver. If you are using Python, your script MUST be titled [YOUR_NAU_ID]_webserver.py. The program should take as its first argument a port to listen from. **Don't use a hard-coded port number.** All submissions must include a README with a description of what the program does, how to compile (if necessary), how to run, and what to expect as output.
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:

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

In this assignment, client requests to the proxy must be in their absolute URI form (see RFC 1945, Section 5.1.2), e.g.,

```
GET http://www.cs.princeton.edu/index.html HTTP/1.0
```

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.

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. See the `URL (7)` manual page for more info. 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, i.e., “Not Found” (404).

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]_webserver.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/1.0
```

On your screen, the headers and the text associated with the HTML page you downloaded from BbLearn.

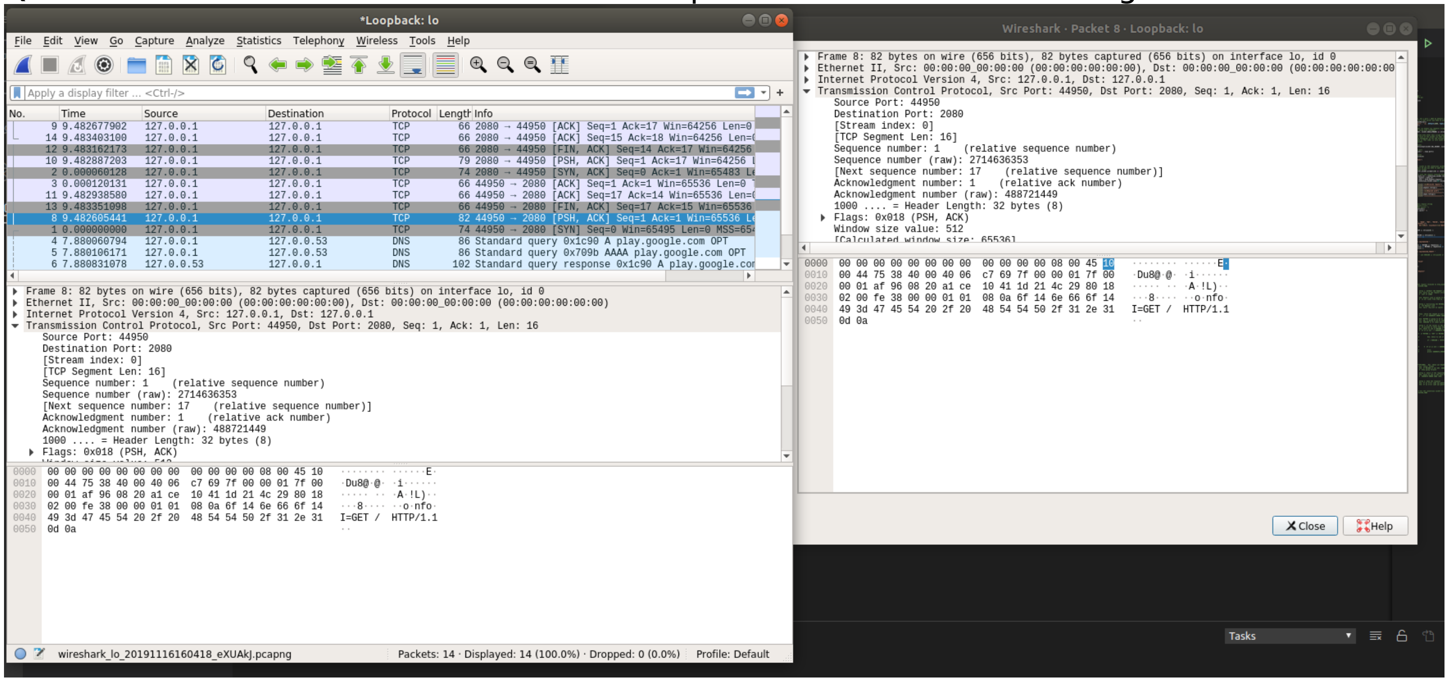
Part 2: Evaluate the Web server with Wireshark

In this section, we will look under the hood to understand at a packet level what it looks like to use a Web server.

1. Start a Wireshark capture on l0 (localhost).
2. Start your Web server.
3. Run the telnet test from Part 1: Testing correctness.
4. Stop the Wireshark capture.

Now, examine the packets you captured as part of running your evaluation.

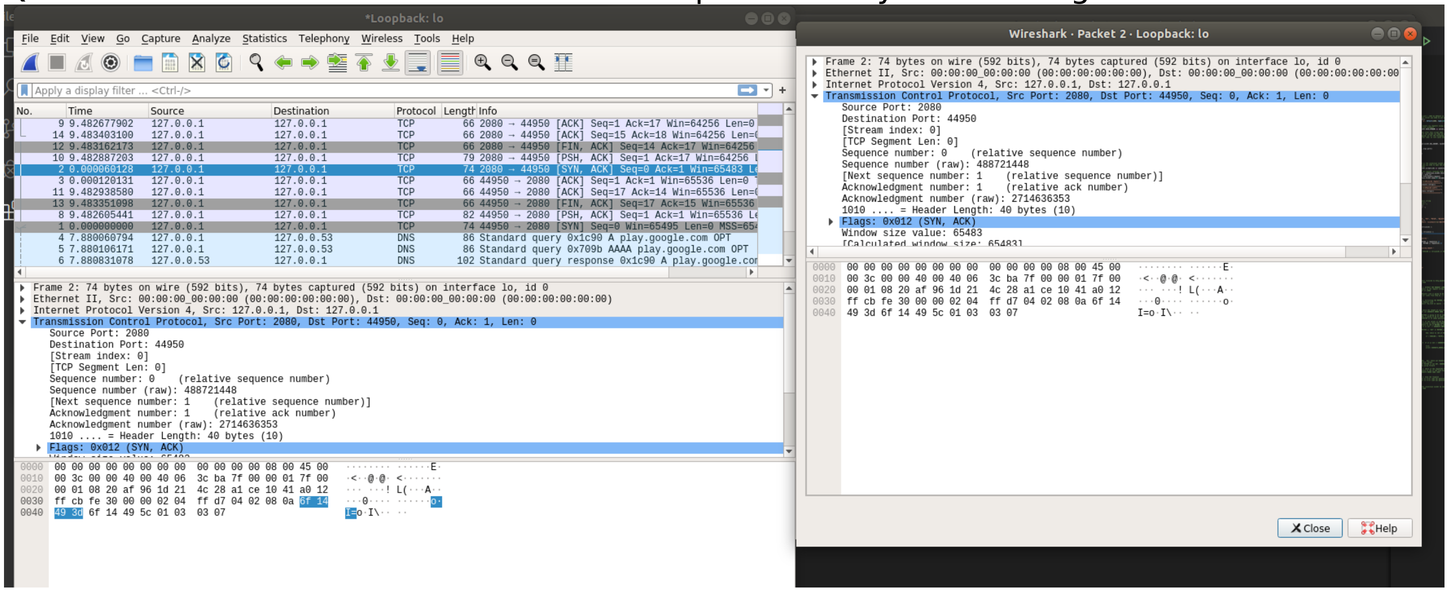
Q0. Post a screen shot of the HTTP GET request that the telnet client generated.


*Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Wireshark · Packet 8 · Loopback: lo
Close
Help
Tasks

```
No.  Time         Source      Destination  Protocol Length Info
   9 9.482677902  127.0.0.1   127.0.0.1    TCP       66 2080 → 44950 [ACK] Seq=1 Ack=17 Win=64256 Len=0
  14 9.483403100  127.0.0.1   127.0.0.1    TCP       66 2080 → 44950 [ACK] Seq=15 Ack=18 Win=64256 Len=(
  12 9.483162173  127.0.0.1   127.0.0.1    TCP       66 2080 → 44950 [FIN, ACK] Seq=14 Ack=17 Win=64256
  10 9.482887203  127.0.0.1   127.0.0.1    TCP       79 2080 → 44950 [PSH, ACK] Seq=1 Ack=17 Win=64256 L
   2 0.000060128  127.0.0.1   127.0.0.1    TCP       74 2080 → 44950 [SYN, ACK] Seq=0 Ack=1 Win=65483 Le
   3 0.000120131  127.0.0.1   127.0.0.1    TCP       66 44950 → 2080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 T
  11 9.482938580  127.0.0.1   127.0.0.1    TCP       66 44950 → 2080 [ACK] Seq=17 Ack=14 Win=65536 Len=(
  13 9.483351098  127.0.0.1   127.0.0.1    TCP       66 44950 → 2080 [FIN, ACK] Seq=17 Ack=15 Win=65536
   8 9.482605441  127.0.0.1   127.0.0.1    TCP       82 44950 → 2080 [PSH, ACK] Seq=1 Ack=1 Win=65536 Le
   1 0.000000000  127.0.0.1   127.0.0.1    TCP       74 44950 → 2080 [SYN] Seq=0 Win=65495 Len=0 MSS=654
   4 7.880060794  127.0.0.1   127.0.0.53   DNS       86 Standard query 0x1c90 A play.google.com OPT
   5 7.880106171  127.0.0.1   127.0.0.53   DNS       86 Standard query 0x709b AAAA play.google.com OPT
   6 7.880831078  127.0.0.53  127.0.0.1    DNS      102 Standard query response 0x1c90 A play.google.con

▸ Frame 8: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▾ Transmission Control Protocol, Src Port: 44950, Dst Port: 2080, Seq: 1, Ack: 1, Len: 16
    Source Port: 44950
    Destination Port: 2080
    [Stream index: 0]
    [TCP Segment Len: 16]
    Sequence number: 1    (relative sequence number)
    Sequence number (raw): 2714636353
    [Next sequence number: 17    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    Acknowledgment number (raw): 488721449
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x018 (PSH, ACK)

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 10   ········ ······E·
0010  00 44 75 38 40 00 40 06  c7 69 7f 00 00 01 7f 00   ·Du8@·@· ·i······
0020  00 01 af 96 08 20 a1 ce  10 41 1d 21 4c 29 80 18   ····· ·· ·A·!L)··
0030  02 00 fe 38 00 00 01 01  08 0a 6f 14 6e 66 6f 14   ···8···· ··o·nfo·
0040  49 3d 47 45 54 20 2f 20  48 54 54 50 2f 31 2e 31   I=GET /  HTTP/1.1
0050  0d 0a                                              ··

wireshark_lo_20191116160418_eXUAkJ.pcapng    Packets: 14 · Displayed: 14 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

```
▸ Frame 8: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▾ Transmission Control Protocol, Src Port: 44950, Dst Port: 2080, Seq: 1, Ack: 1, Len: 16
    Source Port: 44950
    Destination Port: 2080
    [Stream index: 0]
    [TCP Segment Len: 16]
    Sequence number: 1    (relative sequence number)
    Sequence number (raw): 2714636353
    [Next sequence number: 17    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    Acknowledgment number (raw): 488721449
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x018 (PSH, ACK)
    Window size value: 512
    [Calculated window size: 65536]

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 10   ········ ······E·
0010  00 44 75 38 40 00 40 06  c7 69 7f 00 00 01 7f 00   ·Du8@·@· ·i······
0020  00 01 af 96 08 20 a1 ce  10 41 1d 21 4c 29 80 18   ····· ·· ·A·!L)··
0030  02 00 fe 38 00 00 01 01  08 0a 6f 14 6e 66 6f 14   ···8···· ··o·nfo·
0040  49 3d 47 45 54 20 2f 20  48 54 54 50 2f 31 2e 31   I=GET /  HTTP/1.1
0050  0d 0a                                              ··
```

Q1. Post a screen shot of the HTTP GET response that your server generated.


*Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Wireshark · Packet 2 · Loopback: lo
Close
Help

```
No.   Time         Source       Destination  Protocol Length Info
    9 9.482677902  127.0.0.1    127.0.0.1    TCP       66 2080 → 44950 [ACK] Seq=1 Ack=17 Win=64256 Len=0
   14 9.483403100  127.0.0.1    127.0.0.1    TCP       66 2080 → 44950 [ACK] Seq=15 Ack=18 Win=64256 Len=(
   12 9.483162173  127.0.0.1    127.0.0.1    TCP       66 2080 → 44950 [FIN, ACK] Seq=14 Ack=17 Win=64256
   10 9.482887203  127.0.0.1    127.0.0.1    TCP       79 2080 → 44950 [PSH, ACK] Seq=1 Ack=17 Win=64256 L
    2 0.000060128  127.0.0.1    127.0.0.1    TCP       74 2080 → 44950 [SYN, ACK] Seq=0 Ack=1 Win=65483 Le
    3 0.000120131  127.0.0.1    127.0.0.1    TCP       66 44950 → 2080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 T
   11 9.482938580  127.0.0.1    127.0.0.1    TCP       66 44950 → 2080 [ACK] Seq=17 Ack=14 Win=65536 Len=(
   13 9.483351098  127.0.0.1    127.0.0.1    TCP       66 44950 → 2080 [FIN, ACK] Seq=17 Ack=15 Win=65536
    8 9.482605441  127.0.0.1    127.0.0.1    TCP       82 44950 → 2080 [PSH, ACK] Seq=1 Ack=1 Win=65536 Le
    1 0.000000000  127.0.0.1    127.0.0.1    TCP       74 44950 → 2080 [SYN] Seq=0 Win=65495 Len=0 MSS=654
    4 7.880060794  127.0.0.1    127.0.0.53   DNS       86 Standard query 0x1c90 A play.google.com OPT
    5 7.880106171  127.0.0.1    127.0.0.53   DNS       86 Standard query 0x709b AAAA play.google.com OPT
    6 7.880831078  127.0.0.53   127.0.0.1    DNS      102 Standard query response 0x1c90 A play.google.cor

▸ Frame 2: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▾ Transmission Control Protocol, Src Port: 2080, Dst Port: 44950, Seq: 0, Ack: 1, Len: 0
    Source Port: 2080
    Destination Port: 44950
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Sequence number (raw): 488721448
    [Next sequence number: 1    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    Acknowledgment number (raw): 2714636353
    1010 .... = Header Length: 40 bytes (10)
  ▸ Flags: 0x012 (SYN, ACK)

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 3c 00 00 40 00 40 06  3c ba 7f 00 00 01 7f 00   ·<··@·@· <·······
0020  00 01 08 20 af 96 1d 21  4c 28 a1 ce 10 41 a0 12   ··· ···! L(···A··
0030  ff cb fe 30 00 00 02 04  ff d7 04 02 08 0a 6f 14   ···0···· ······o·
0040  49 3d 6f 14 49 5c 01 03  03 07                     I=o·I\·· ··
```

```
▸ Frame 2: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▾ Transmission Control Protocol, Src Port: 2080, Dst Port: 44950, Seq: 0, Ack: 1, Len: 0
    Source Port: 2080
    Destination Port: 44950
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Sequence number (raw): 488721448
    [Next sequence number: 1    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    Acknowledgment number (raw): 2714636353
    1010 .... = Header Length: 40 bytes (10)
  ▸ Flags: 0x012 (SYN, ACK)
    Window size value: 65483
    [Calculated window size: 65483]

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 3c 00 00 40 00 40 06  3c ba 7f 00 00 01 7f 00   ·<··@·@· <·······
0020  00 01 08 20 af 96 1d 21  4c 28 a1 ce 10 41 a0 12   ··· ···! L(···A··
0030  ff cb fe 30 00 00 02 04  ff d7 04 02 08 0a 6f 14   ···0···· ······o·
0040  49 3d 6f 14 49 5c 01 03  03 07                     I=o·I\·· ··
```

Part 3: Write a multi-threaded Web proxy[1]

In this part of the assignment, you will implement a Web proxy that passes requests and data between multiple Web clients and Web servers, **concurrently**. When you're done with the assignment, you should be able to configure Firefox to use your personal proxy server as a Web proxy.

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).

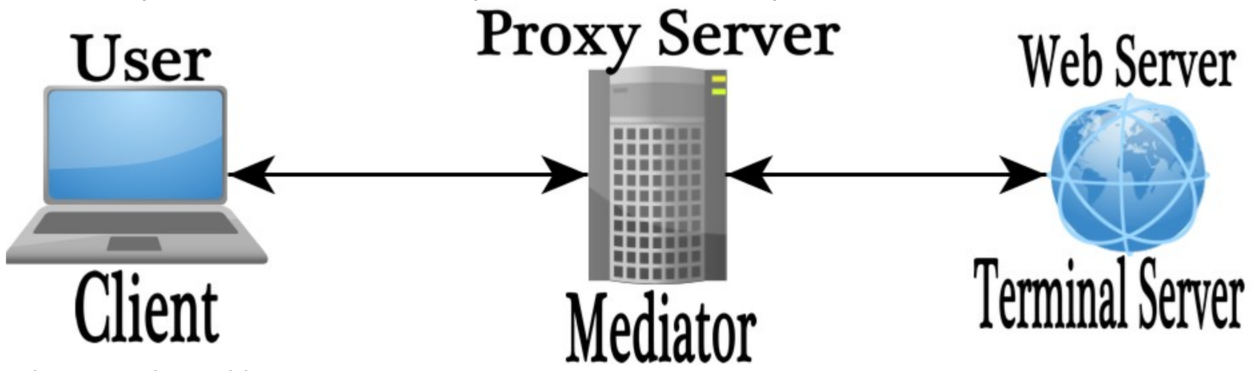


Figure 1. https://web.stupidproxy.com/

[1] This portion of the lab modifies the original content of the lab as presented by http://www.cs.princeton.edu/courses/archive/spr15/cos461/assignments/1-http.html

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.

References:
RFC 1945 The Hypertext Transfer Protocol, version 1.0

Assignment Details

Your task is to build a web proxy capable of accepting HTTP requests, forwarding requests to remote (origin) servers, and returning response data to a client. The proxy MUST handle **concurrent** requests by forking a process for each new client request. You will only be responsible for implementing the HTTP GET method. All other request methods received by the proxy should elicit a "Not Implemented" (501) error (see RFC 1945 section 9.5 - Server Error).
This assignment can be completed in either Python, C, or C++. It should compile and run without errors on Ubuntu 16.04. If you are using C or C++ you MUST submit a Make file that compiles your code and creates a binary titled [YOUR_NAU_ID]_proxy. If you are using Python, your script MUST be titled [YOUR_NAU_ID]_proxy.py. The program should take as its first argument a port to listen from. **Don't use a hard-coded port number.**
All submissions must include a README with a description of what the program does, how to compile (if necessary), how to run, and what to expect as output.
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 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. Each new client request is accepted, and a new process is spawned to handle the request. To avoid overwhelming your server, you should not create more than a reasonable number of child processes (for this experiment, use at most 20), in which case your server should wait until one of its ongoing child processes exits before forking a new one to handle the new request.

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 (see RFC 1945, Section 5.1.2), e.g.,

```
GET http://www.cs.princeton.edu/index.html HTTP/1.0
```

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/1.0

Host: www.cs.princeton.edu
```

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.

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. See the `URL (7)` manual page for more info. 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.

The following summarizes how status replies should be sent from the proxy to the client:

- For any error your proxy should return the status 500 'Internal Error'. This means for any request method other than GET, your proxy should return the status 500 'Internal Error' rather than 501 'Not Implemented'. Likewise, for any invalid, incorrectly formed headers or requests, your proxy should return the status 500 'Internal Error' rather than 400 'Bad Request' to the client. For any error that your proxy has in processing a request such as failed memory allocation or missing files, your proxy should also return the status 500 'Internal Error'. (This is what is done by default in this case.)
- Your proxy should simply forward status replies from the remote server to the client. This means most 1xx, 2xx, 3xx, 4xx, and 5xx status replies should go directly from the remote server to the client through your proxy. Most often this should be the status 200 'OK'. However, it may also be the status 404 'Not Found' from the remote server. (While you are debugging, make sure you are getting valid 404 status replies from the remote server and not the result of poorly forwarded requests from your proxy.)

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]_proxy.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/1.0
```

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.

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.

Part 4: Evaluate the Web proxy with Wireshark

In this section, we will look under the hood to understand at a packet level what it looks like to use a Web proxy.

1. Start a Wireshark capture on l0 (localhost).
2. Start your proxy.
3. Open two terminal windows and start a telnet client in each that connects to the port that your proxy is listening to.
4. With one telnet client, make a request to http://www.nau.edu. In the other, make a request to http://www.google.com.
5. Stop the Wireshark capture.

Now, examine the packets you captured as part of running your evaluation.

Q2. Post a screen shot of the telnet client's HTTP GET request to http://www.nau.edu

Q3. Post a screen shot of the Web proxy's HTTP GET request to http://www.nau.edu

Q4. Post a screen shot of the telnet client's HTTP GET request to http://www.google.com

Q5. Post a screen shot of the Web proxy's HTTP GET request to http://www.google.com

Q6. Post a screen shot of the Web proxy's received HTTP GET response from http://www.nau.edu

Q7. Post a screen shot of the Web proxy's forwarded HTTP GET response to the telnet client

Q8. Post a screen shot of the Web proxy's received HTTP GET response from http://www.google.com

Q9. Post a screen shot of the Web proxy's forwarded HTTP GET response to the telnet client

Q10. Did it take longer for the requests that you made in Q0 to be fulfilled than the request you made in Q2 and Q4?

Q11. Explain why the response time was longer or shorter using evidence and what you have learned about computer networking so far.

References and Helpful Hints

Be sure to close your server when you are done with it. Otherwise, you are leaving the socket open and using up a port number and this can lead to complications if you try running another instance with another port number.

Note that this is a very standard lab activity for computer networks courses and as such, it may be tempting to find completed projects online and submit them as your own. I would remind you of two consequences of succumbing to this temptation. A short-term consequence is that it would be a violation of the College-level and University-level Academic Integrity policy and would be prosecuted as such. More importantly, as a long-term consequence, you would be cheating yourself of an opportunity to delve a little bit deeper into a hands-on experience with new material in a manner that would maximize your learning.

Some tutorials and documentations

https://www.tutorialspoint.com/python/python_multithreading.htm
https://www.geeksforgeeks.org/multithreading-c-2/
https://docs.python.org/3/library/socket.html

Appendices

APPENDEX A. HTTP Error Handling

For your multithreaded web proxy server, you are only expected to implement support for GET methods.

- If the client attempts another method (e.g., POST or HEAD), you are to return a 501 error code.
- If the origin server returns an error, you are to pass that along to the client unmodified. This gets into the section of this document called "Data Encoding from the Origin Server"
- If the client sends the web proxy server a malformed GET request, you are to return a 400 error code.
- If the client sends the web proxy server a correctly formed GET request with malformed headers, you should return a 403 or 404 error code, per the RFC:

```
The server understood the request, but is refusing to fulfill it.
Authorization will not help and the request SHOULD NOT be repeated.
If the request method was not HEAD and the server wishes to make public why the
request has not been fulfilled, it SHOULD describe the reason for the refusal in
the entity.  If the server does not wish to make this information available to the
client, the status code 404 Not Found) can be used instead.
```

So in short, your proxy server should specifically be able to dole out: 501 Not Implemented, 400 Bad Request, and 403 Forbidden **OR** 404 Not Found. The origin server will be responsible for the formation of any other returns.

APPENDEX B. HTTP Version Handling

Most things on the Internet use HTTP 1.1 nowadays so this kind of botches the original language of this lab (which mentions only handling 1.0). A simple fix is that you can intercept GET requests from clients and then replace the version number with 1.0 as you relay to the origin server. This prevents any assumptions of persistence between the client and the proxy and the proxy and the origin server. Per the RFC:

```
The proxy server MUST signal persistent connections separately with its clients and
the origin servers (or other proxy servers) that it connects to. Each persistent
connection applies to only one transport link.
```

APPENDEX C. Data Encoding from Origin Server

The proxy does not actually need to read or interpret any data sent by the origin server and can just forward it along “as-is”—as in data from the server can be read straight from the socket and can be sent immediately back to the client without any subsequent processing.

APPENDEX D. Blocking Sockets

If you are experiencing an issue where you have definitely sent data to your server and it is unable to read that data even though it has been sent across the network, this means you have a blocking issue. Blocking calls like recv( ) have to wait on system calls (I/O) to complete before they can return a value. So, your program, the caller, is blocked until they’re done or a timeout or other error occurs.

One solution to this is to use nonblocking sockets and select(), but this actually more of a single-threaded solution because what it does it places your socket into a register of I/O streams that will experience EVENT_READ and poll through the I/O streams in the register until one of them is ready to be handled. This is great, but it does not scale well to many I/O streams (i.e., many socket connections). I’m fine if you used this for the Simple Web Server, but **know that select( ) not the proper solution for multithreads.**

With a multithreaded approach, your listening socket will accept new connection requests by making a new connection socket, and then will create a new thread to handle any incoming data being sent over that socket. Because it is being serviced by its own thread, it is ok if it blocks because there are no other incoming connections waiting to be serviced by that particular thread.

If you are using telnet and having issues on read, you are probably running into the problem that telnet does not flush its buffers and does not do a good job of differentiating automatically between input intended for local use and input intended to be sent afar. Reading through about “flush” in this article should help:
http://nersp.nerdc.ufl.edu/~dicke3/nerspcs/telnet.html

Using an actual browser to test is another way you can likely avoid this issue.