PROGRAMMING ASSIGNMENT 3
Due: Sunday, April 5th, 2020 @11:55 pm
NOTE: This assignment is NOT a group assignment, but an individual assignment

Objective:

In this programming assignment, you will build a simple web proxy capable of relaying HTTP requests from clients to HTTP servers.

Background:

HTTP is a client-server protocol; the client usually connects directly with the server. But often times it is useful to introduce an intermediate entity called proxy. The proxy sits between HTTP clients and HTTP servers. With the proxy, the HTTP client sends an HTTP request to the proxy. The proxy then forwards this request to the HTTP server, and then receives the reply from the HTTP server. The proxy finally forwards this reply to the HTTP client.

There are several advantages to utilizing a Proxy:

1. **Performance:** If we cache a page that is frequently accessed by clients connected to the proxy, we can reduce the need for new connections to the actual HTTP server.
2. **Content filtering and transformation:** The Proxy can inspect URLs from the HTTP client requests and decide whether to block connections to some specific URLs. Or to modify web pages (e.g. image transcoding for clients with limited processing capability).
3. **Privacy:** When an HTTP server attempts to log IP address information from the HTTP client, it will only get the information of the proxy, not the actual client.

Description:

a. The basics

In this assignment the proxy application **webproxy** needs to be written in C. It should compile and run without error with the following command:

 # webproxy 10001&

The first argument is the **PORT** number that is used to listen to HTTP client requests. Your application should handle any port number.

b. Listening and accepting connections from HTTP clients

When your application starts, it will create a **TCP** socket to be used for listening requests from HTTP clients.

When a HTTP request comes in, your application should be able to figure out whether this request is properly- formatted with the following criteria:

1) Whether the HTTP method is valid. Our web proxy **only supports GET** (other methods such as POST and CONNECT do not need to be supported)
2) Whether the HTTP server specified in the requested URL exists, usually done by calling gethostbyname().

If the criteria are not meet, a corresponding error message needs to be sent to the client (e.g., 400 bad request). If the criteria are met, then the request will be forwarded.

c. Parsing requests from HTTP clients

When a valid request comes in, the proxy will need to parse the requested URL, it basically disassembles the content into the following 3 parts.

1) Requested host and port number (you should pick the **correct remote port or use the default 80 port if none** is specified).
2) Requested path, this is used to access resources at the HTTP server.
3) Optional message body (this may not appear in every HTTP request)

If we set our browser to use the local proxy running on port 8000 we can capture a GET request to www.yahoo.com using nc:

```
PA#2$ nc -l 8000
GET http://www.yahoo.com/ HTTP/1.1
Host: www.yahoo.com
```

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:73.0) Gecko/20100101 Firefox/73.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cookie: B=7dktfqdf1kcio&b=3&s=kv; ucs=lnct=1580791591&bnpt=1580791592
Upgrade-Insecure-Requests: 1

d. Forwarding these requests to the HTTP server and relaying data

After the proxy parses the information from the client, it constructs a new HTTP request and sends it over a new socket connection to the HTTP server. The proxy will then forward the reply from the HTTP server back to the HTTP client.

e. Summary

In summary, you will have two connections for each request -- one from the client to the proxy (socket1) and another one from the proxy to the server (socket2). Your proxy listens on the designated port number and accepts connections on socket1.

You can then parse the request received on socket1 and check if the request is properly formatted. If so, your proxy initiates a new TCP connection and sends the request to the HTTP server over socket2.

f. Testing your proxy

First you need to run your proxy. It will run at 127.0.0.1 on the port number you specify:

```
# webproxy <port> &
```

Telnet into localhost (127.0.0.1) <port> and type "GET http://www.google.com HTTP/1.0"

```
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
```

Press the Enter key twice.

If your proxy is working correctly, the headers and HTML contents of the Google homepage should be displayed.

Features:

a. Basic Proxying

Your proxy should perform the basic function of proxying requests from the client to the origin webserver.

Your proxy must check the HTTP method. Only the GET method must be supported. For any other method, HTTP 400 Bad Request error message must be sent to the client.

If the requested hostname is not resolved to any IP address, then the error message (HTTP 404 Not Found) must be sent to the client and displayed in the browser. You can do this right after your gethostbyname() call as this will only return IP addresses for valid hostnames.

b. Concurrent Requests

You should implement your proxy to handle multiple requests at the same time, so that multiple clients can access different servers via your proxy simultaneously. You may choose to follow a multi-threaded or multi-process model in your code. Multi-process will probably be easier to implement and more familiar, but multi-threaded will have some advantages in terms of implementing the additional features asked for below.

c. Caching

Caching is one of the most common performance enhancements that web proxies implement. Caching takes advantage of the fact that most pages on the web don't change that often, and that any page that you visit, you (or someone else using the same proxy) are likely to visit again. A caching proxy server saves a copy of the files that it retrieves from remote servers. When another request comes in for the same resource, it returns the saved (or cached) copy instead of creating a new connection to a remote server. This can create more significant savings in the case of a more distant server or a remote server that is overloaded (it can also help reduce the load on heavily loaded servers).

Caching introduces a few new complexities as well. First of all, a great deal of web content is dynamically generated, and as such shouldn't really be cached. Second, we need to decide how long to keep pages in our cache. If the timeout is set too short, we negate most of the advantages of having a caching proxy. If the timeout is set too long, the client may end up looking at pages that are outdated or irrelevant.

There are a few steps to implementing caching behavior for your web proxy:

1. **Timeout setting**: alter your proxy so that you can specify a timeout value on the command line as a runtime option. For example, "./proxy 10001 60" will run the proxy with a cache timeout of 60 seconds.

2. **Page cache**: you'll need to alter how your proxy retrieves pages. It should now check to see if a page exists in the proxy before retrieving a page from a remote server. If there is a valid cached copy of the page, that should be presented to the client instead of creating a new server connection. You need to create a local file or files to store the retrieved pages.

3. **Expiration:** you will need to implement cache expiration. It's okay if a page is still in your cache after the timeout has expired, but you want to ensure that pages that are older than the user-set timeout are fetched from the origin server and not served from the cache.

   To make the problem simpler, you can choose to skip handling the Cache-Control header, which is normally sent by the server and specifies how long the client or the proxy should cache the content. Instead, use the timeout specified on the command line.

4. **Blacklist:** Make a file which has a list of websites and IP addresses that must be blocked. Thus, for every incoming request, the proxy must look up this data and deny requests for these websites or IPs. When the client request one of them, your proxy must return "ERROR 403 Forbidden". This file consists of one blacklist entry per line and can contain both hostnames and IP addresses.

   For example:

   www.facebook.com
   www.cnn.com
   192.168.43.20
   ...
   ...

Your proxy must be able to block requests based on hostname as well as IP address. (Ask TAs for clarification if this is not clear)

Tip: when you want to check if a specific URL exists in the cache, you might consider storing a hash of the URL instead of the entire URL itself. For example, suppose that you store http://www.yahoo.com/logo.jpg with a key generated by using a suitable hash

function. When you receive the same URL from the client, you can simply calculate the hash of the requested URL and compare it with the existing keys stored in the cache.

d. Link Prefetch (10 Extra Credits)

Building on top of your caching proxy, the last piece of functionality should implement link prefetching. The idea behind link prefetching is simple: if a user asks for a particular page, the odds are that he or she will next request a page linked from that page. Link prefetching uses this information to attempt to speed up browsing by parsing requested pages for links, and then fetching the linked pages in the background. The pages fetched from the links are stored in the cache, ready to be served to the client when they are requested without the client having to wait around for the remote server to be contacted.

Parsing and fetching links can take a significant amount of time, especially for a page with a lot of links. In order to fetch multiple links simultaneously in background, you'll most likely have to use multi-threading. One thread or threads could remain dedicated to the tasks that you have already implemented: reading requests from the client and serving pages from either the cache or a remote server. A separate thread, can parse a page and extract the HTTP links, request those links from the remote server, and add them to the cache.

Submission:

You will need to submit a tarball containing the following:

1. All of the source code for your proxy and files you create.
2. A README file describing your code and the design decisions that you made (will be graded).
3. Your tarball should be named <identikey>.tar where <identikey> is your identikey.
4. Your code should compile without errors or warnings. a MAKEFILE is needed to compile your source code.
5. Your proxy should be tested against a couple of browsers: Firefox, Chrome, etc.