PROGRAMMING ASSIGNMENT – 3

Due: 13th April 2021 11:59 pm

NOTE: This assignment is a group (up to 2) assignment.

1. Objective:

In this programming assignment, you are required to build a simple web proxy server that is capable of relaying HTTP requests from clients to the HTTP servers.

2. 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 a 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.

Here are several advantages of utilizing a Proxy:

1. **Performance:** If we cache a page that is frequently accessed by clients connected to this proxy, this can reduce the extra need of creating new connections to the actual HTTP server every time.
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 reform web pages (e.g. image transcoding for clients with limited processing capability).
3. **Privacy:** Sometimes when the HTTP server tries to log IP address information from the HTTP client, it can only get the information of the proxy but not the actual client.

3. Assignment Description:

a. The basics

In this assignment one proxy application **webproxy** need to be created using ANSI 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 connection 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 (by calling gethostbyname() more details are mentioned later in the assignment.)

If the criteria are not meet, a corresponding error message needs to be sent to the HTTP 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)

For example, the message received by the proxy is nothing different from the HTTP client (as in our previous programming assignment) since you only need to implement the GET method. Below we set the browser to use the local proxy running on port 8000 and captured the GET request to www.yahoo.com. It's just an example you don't have to run it in your computer.

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

```
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.71 Safari/537.36
DNT: 1
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Cookie: B=fgvem7ha9r7hv&b=3&s=0o; ucs=sfcTs=1425977267&sfc=1
--
```

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

After the proxy parsed the information from the client, it constructs a new HTTP request and send it over a new socket connection to the HTTP server. Then the proxy will forward the reply from the HTTP server back to the HTTP client.

e. Summary

In summary, you will have two socket connections -- one from the client to the proxy (sk1) and another one from the server to the proxy (sk2). Your proxy listening on the port number and accepts the connection. Then your proxy will have a socket descriptor sk1 at that time. You parse the request received from the socket description sk1 and checks if the request packet is properly formatted. If so, your proxy needs to create another TCP connection and sends this request the HTTP server. When this request is accepted by the HTTP server, your proxy will have another descriptor (sk2). Your proxy most likely copies from sk1 to sk2 and sk2 to sk1.

f. Testing your proxy

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

# webproxy <port> &

Then try 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
```
Please press two Enter keys after this.

If your proxy is working correctly, the headers and HTML contents of the Google homepage should be displayed. This is just example, you may use any website to test.

You must test your Proxy with your web browsers as we will test your proxy using the web browser; detailed steps are mentioned in Appendix A.

4. Features Required:

a. Multi-threaded Proxy

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.

Your proxy must check the HTTP method. Only 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 error message) must be sent to the client and displayed in the browser. You can do this right after your gethostbyname() call as this will return the IP address of only the valid hostnames.

b. 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 once 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 the most of 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 runtime options. For example, "./proxy 10001 60" will run proxy with timeout value 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 to store retrieved page.
3. **Expiration:** you will need to somehow implement cache expiration. The timing does not need to be exact (i.e. it's okay if a page is still in your cache after the timeout has expired, but it's not okay to serve a cached page after its timeout has expired), but you want to ensure that pages that are older than the user-set timeout are not served from the cache. To make the problem simpler, we are not handling the Cache-Control header, which is sent by the server and specifies how long the client or the proxy can cache the content (which is in standard). Instead, we will use the timeout specified by the proxy.
4. **Hostname' IP address cache:** Your proxy must have a cache of IP addresses it resolved for any hostname and store (hostname,IP) pair in local cache file (a file on a disk or within program memory, any approach should be okay). Thus, if same hostname is requested again, your proxy should skip the DNS query to reduce DNS query time. You don't have to implement expiration on this "hostname-IP" mapping because the hostname to IP address mapping usually won't change frequently.
5. **Blacklist:** Make a file which has a list of websites and IP addresses that must be blocked. Thus, for every incoming request, proxy must look up this data and deny requests for these websites or IPs. When client request one of them, you return "ERROR 403 Forbidden" immediately from the proxy itself. This file can be just one line after line and can have both hostname or the IP address.

   For eg.

   www.facebook.com

   www.cnn.com

   192.168.43.20

..

..

The structure of the file can be anything. You can have all the hostnames at the start and then list of IPs or one after another. Anything should be okay. However, 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 check if the specific URL exists in the cache, you can compare hash codes instead of the entire URL. For example, suppose that you store http://www.yahoo.com/logo.jpg with the hash key 0xAC10DE97073ACD81 using your own hash function or md5sum of the URL. When you receive the same URL from the client, you can simply calculate the hash value of the requested URL and compares it with hash keys stored in the cache.

c. Link Prefetch (10 Extra Credits)

Building on top of your caching and content transformation code, the last piece of functionality that you can implement is called 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 have to use multi-threading. One thread should 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. In a separate thread, the proxy will parse a page and extract the HTTP links, request those links from the remote server, and add them to the cache.

5. Submission and Grading

You should submit your completed proxy by the date posted on the course website to GitHub Classroom. You will need to submit 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 code should compile without errors or warnings. a MAKEFILE is needed to compile your source code. Reminder to compile on the CSEL machines.
4. Your proxy should work with browsers: Firefox, Chrome etc.

Access the GitHub Classroom link below. Note: The httpechosrv-basic.c skeleton code from assignment 2 has been added to each repo. This file is for reference only in case students need easy access to the file. No other skeleton code is provided, so you can either modify your previous assignment, modify httpechosrv-basic.c, or start from scratch. Please follow file naming conventions above.
https://classroom.github.com/g/vvNvD71T

Grading:

Multithreading and Caching features are worth 100 points.
On high level, if your proxy server works without any issues (program hangs or images not working, etc.) and is able to handle multiple requests and relay the response from server to the client successfully, 70% points will be awarded.

All the **caching features** are then for remaining **30%.**

Link prefetching extra credit is for **10** points.

Appendix A.

Configuring Firefox to Use a Proxy

Follow instructions at: There may be changes in UI of the browser, but you should be able to figure it out and configure it.

https://www.wikihow.com/Enter-Proxy-Settings-in-Firefox

Configure Chrome to use proxy:
Follow instructions at (except step 7):
https://www.cactusvpn.com/tutorials/how-to-set-up-proxy-on-chrome/

Note: since there are a variety of browsers and operating systems, your exact combination may not be listed above. Try to do some googling if you need help. If all else fails, please either post in Canvas Discussion or ask the TA/instructor.

HTTP Basics:
https://www3.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html