The Proxy Lab

**1. Introduction**

A Web proxy is a program that acts as a middleman between a Web browser and an *origin server*. Instead of contacting the origin server directly to get a Web page, the browser contacts the proxy, which forwards the request on to the origin server. When the origin server replies to the proxy, the proxy sends the reply on to the browser.  
  
Proxies are useful for many purposes.  Sometimes proxies are used in firewalls, so that browsers behind a firewall can only contact a server beyond the firewall via the proxy.  Proxies can also act as anonymizers: by stripping requests of all identifying information, a proxy can make the browser anonymous to Web servers.  Proxies can even be used to cache web objects by storing local copies of objects from servers then responding to future requests by reading them out of its cache rather than by communicating again with remote servers.  
  
In this lab, you will write a simple HTTP proxy that caches webobjects.  For the first part of the lab, you will set up the proxy to accept incoming connections, read and parse requests, forward requests to web servers, read the servers' responses, and forward those responses to the corresponding clients.  This first part will involve learning about basic HTTP operation and how to use sockets to write programs that communicate over network connections.   In the second part, you will add caching to your proxy using a simple main memory cache of recently accessed web content.

**2. Hand Out Instructions**

You have the option of working alone or working with someone else on this lab assignment.  If you want to work with someone else then figure out who that will be before continuing.  You cannot add a person later on.

After you've found a teammate or decided to work alone, use a browser to log into github and then open up this page:

<https://classroom.github.com/g/OWNVFRHx>

Both you and your teammate (if you have one) will visit this URL.  One of you will create the team.  The other of you will select the created team.

Next, get into your 3482 directory on the student2 machine and create your local repository by executing a clone.

﻿This will cause a number of files to be unpacked into the directory.  The files you will be modifying  are*proxy.c*and *cache.c*.

**3. Implementing a Proxy Server**

The first step is implementing a basic sequential proxy that handles GET requests.  Other requests type, such as POST, are strictly optional.  
  
When started, your proxy should listen for incoming connections on a port whose number will be specified on the command line. Once a connection is established, your proxy should read the entirety of the request from the client and parse the request. It should determine whether the client has sent a valid HTTP request; if so, it can then establish its own connection to the appropriate web server then request the object the client specified.  Finally, your proxy should read the server's response and forward it to the client.

**3.1 Request Line**

When an end user enters a URL such as:

http://www.cmu.edu/hub/index.html

into the address bar of a web browser, the browser will send an HTTP request to the proxy that begins with a line that might resemble the following:

GET http://www.cmu.edu/hub/index.html HTTP/1.1

In that case, the proxy should parse the request into the following fields:

the hostname: **www.cmu.edu**

the path and/or query: **/hub/index.html**

the method:  **HTTP/1.1**

That way, the proxy can determine that it should open a connection to www.cmu.edu and send an HTTP request of its own starting with a line of the following form:

GET /hub/index.html HTTP/1.1

Note that all lines in an HTTP request end with a carriage return, \r, followed by a newline, \n. Also important is that every HTTP request is terminated by an empty line: \r\n\r\n

**3.2 Request Headers**

The important request headers for this lab are the Host, User-Agent, Connection, and Proxy-Connection headers:

A. The **Host** header describes the hostname of the end server.  
For example, to access http://www.cmu.edu/hub/index.html, your proxy would send the following header:

Host: www.cmu.edu

It is possible that web browsers will attach their own Host headers to their HTTP requests.  If that is the case, your proxy should use the same Host header as the browser.

B. The **User-Agent** header identifies the client (in terms of parameters such as the operating system and browser), and web servers often use the identifying information to manipulate the content they serve.  Sending this particular User-Agent: string may improve, in content and diversity, the material that you get back during simple telnet-style testing.  The code provided contains a string literal for this header that you can use.

User-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:10.0.3) Gecko/20120305 Firefox/10.0.3

C. Always send the following **Connection** header:

Connection: close

D. Always send the following  **Proxy-Connection** header:

Proxy-Connection: close

The Connection and Proxy-Connection headers are used to specify whether a connection will be kept alive after the first request/response exchange is completed. It is perfectly acceptable (and suggested) to have your proxy open a new connection for each request. Specifying close as the value of these headers alerts web servers that your proxy intends to close connections after the first request/response exchange.

**3.3 Port Numbers**

There are two significant classes of port numbers for this lab: HTTP request ports and your proxy's listening port.  
  
The HTTP request port is an optional field in the URL of an HTTP request. That is, the URL may be of the form,

http://www.cmu.edu:8080/hub/index.html

in which case your proxy should connect to the host www.cmu.edu on port 8080 instead of the default HTTP port, which is port 80. Your proxy must properly function whether or not the port number is included in the URL.  
  
The listening port is the port on which your proxy should listen for incoming connections. Your proxy should accept a command line argument specifying the listening port number for your proxy. For example, with the following command, your proxy should listen for connections on port 15213:

./proxy 15213

You may select any non-privileged listening port (greater than 1,024 and less than 65,536) as long as it is not used by other processes. Since each proxy must use a unique listening port and many people will simultaneously be working on the student2 machine, the script  port-for-user.pl is provided to help you pick your own personal port number. Use it to generate the port number based on your user ID:

./port-for-user.pl droh  
droh: 45806

The port, p, returned by port-for-user.pl is always an even number. So if you need an additional port number, say for the Tiny server, you can safely use ports p and p+1.  
  
Please don't pick your own random port. If you do, you run the risk of interfering with another user.

**3.4 Incremental Development**

Your instructor has provided you with stubs in proxy.c for the functions you need to implement.  The function documentation describes what the function should do, lists some of the functions that the function will call, and indicates existing code in the Tiny server that is similar to the function you will write.

Do not try to implement the code in proxy.c in one step and then try to debug it.  You should develop the code incrementally.  (In fact, ***every***non-trivial program you write should be developed incrementally.)

A. *First write the code to accept connection requests.* Specifically, write the main function but omit the call to handle the connection.  (This function is called "doit" in tiny.c.)   Use make to create a proxy executable and run it bytyping:

./proxy <port>

where port is the port returned by ./port-for-user.pl

Now, via another terminal window, use ncat to ensure that a client can connect to your server:

ncat localhost <port>

(Alternatively, you can do this in one window by starting the proxy program in the background by adding a & at the end of the command line. However, be sure to kill proxy after your test.)  
  
You should see the message "Accepted connection from (localhost, <client port>).

B. *Write the code to read the request from the client.*  Write the handleRequest function that is called by the main. (This is in place of the function called doit.) At this point, the handleRequest function should simply call getRequest. The getRequest method should read the request and set the method, uri, version, and potentially the header (if the request contains a Host: header line).  The handleRequest function should simply call getRequest and then print those values out.

Test the code at this point by starting up the proxy in one terminal window and then using the curl command in the other terminal window:

curl -v --proxy http://localhost:<proxyport> http://localhost:<tinyport>/home.html

There is no reason to actually start up the tiny server at this point (so you can make up a tinyporty value) since the proxy isn't actually contacting it yet.

C. *Write and test the isValid function*. Call that function from your handleRequest function after the call to getRequest.  Test it by starting up your proxy and using the curl command:

curl -v -d --proxy http://localhost:<proxyport> http://localhost:<tinyport>/home.html

The -d option instructs curl to generate a POST instead of a GET.  
  
By default, curl uses the version HTTP/1.1.  You can get curl to generate the request with HTTP/1.0 by adding the -0 option:

curl -v -0 --proxy http://localhost:17548 http://localhost:17549/home.html

So you can test if isValid accepts HTTP/1.0 and HTTP/1.1.  However, our version of curl won't generate any other HTTP versions so you can't use curl to see if other versions are rejected.

D. Write and test the parseURI function. You can test it by using the curl command and printing out the host, path, and port.  For example, for this command:

curl -v --proxy http://localhost:<proxyport> http://localhost:12345/home.html

The code should print out: (localhost, home.html, 12345)

For this command:

curl -v --proxy http://localhost:<proxyport> http://student2.cs.appstate.edu/index.html

The code should print out: (student2.cs.appstate.edu, index.html, 80)

E. *Write and test the buildRequest function*.  Call the buildRequest function from the handleRequest function.  Print out the request after returning from the call.  If you test it by using this curl command:

curl -v --proxy http://localhost:<proxyport> http://localhost:<tinyport>/home.html

The request will look like this:

GET /home.html HTTP/1.1  
Host: localhost  
User-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:10.0.3) Gecko/20120305 Firefox/10.0.3  
Connection: close  
Proxy-Connection: close

(Remember the \r\n after each line and the two \r\n after the last line.)

If you test it using this curl command:

curl -v --proxy http://localhost:<proxyport> http://student2.cs.appstate.edu/index.html

The request displayed will look like this:

GET /index.html HTTP/1.1  
Host: student2.cs.appstate.edu  
User-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:10.0.3) Gecko/20120305 Firefox/10.0.3  
Connection: close  
Proxy-Connection: close

F. *Write and test the makeRequest function.* Ignore anything having to do with the cache at this point.  Thus, this function should simply send the request to the origin server, read the response from the origin server, and send the response (unaltered) to the client.

If all goes well, you should be able to run the testing script:

driver.sh

and all of the basic tests pass.  If not, take a look at the section on testing and debugging for debugging ideas.

One thing that is useful is to put several curl commands in a file and make the file executable.  You can then startup the tiny server in one window, start up gdb with your proxy server in another window, and then run the curl command executable file in another.  If your proxy is seg faulting, you'll be able to debug it  using gdb.  Of course, you can also set break points before running the curl commands.

Also don't forget to use valgrind for help finding memory related errors.

**4. Caching web objects**  
  
For the final part of the lab, you will add a cache to your proxy that stores recently-used Web objects in memory.  HTTP actually defines a fairly complex model by which web servers can give instructions as to how the objects they serve should be cached and clients can specify how caches should be used on their behalf.  However, your proxy will adopt a simplified approach.  
  
When your proxy receives a web object from a server, it should cache it in memory as it transmits the object to the client.  If another client requests the same object from the same server, your proxy need not reconnect to the server; it can simply resend the cached object.  
  
Obviously, if your proxy were to cache every object that is ever requested, it would require an unlimited amount of memory. Moreover, because some web objects are larger than others, it might be the case that one giant object will consume the entire cache, preventing other objects from being cached at all.  To avoid those problems, your proxy should have both a maximum cache size and a maximum cache object size.  The MAX\_CACHE\_SIZE and MAX\_OBJECT\_SIZE values are defined in cache.h.  These values are actually quite small in order to test your code with relatively small objects and few requests.

The easiest way to implement a correct cache is to dynamically allocate a buffer for each active connection and accumulate data as it is received from the server.  If the size of the object ever exceeds the maximum object size, the buffer can be freed. If the entirety of the web server's response is read before the maximum object size is exceeded, then the object can be cached.

Your proxy's cache should employ a least-recently-used eviction policy.  To implement this, your cache objects will be stored in a doubly-linked list with the most recently used object at the front of the list and the least recently used object at the end. Note that  
both reading an object and writing it count as using the object.

**4.1 Incremental Development**

This portion of the proxy should also be implemented incrementally.

A. *Write and test the cacheInit, addCacheItem, and printCacheList functions.* At this point, don't worry about evicting any objects.  The addCacheItem function should simply add the new node to the front of the list.  Add a call to cacheInit to the main.  Add a call to printCacheList to the end of the handleRequest function.  Modify the makeRequest function so that it dynamically allocates space for the object (MAX\_OBJECT\_SIZE) and copies pieces of the object as it is received from the origin server into the dynamically allocated space, as long as there is room.  (You need to use memcpy here, not strcpy, because the object might be binary.) After the entire object has been read, if it didn't exceed the MAX\_OBJECT\_SIZE, call the addCacheItem function to add it to the cache. Otherwise, free the dynamically allocated space.

Test this code by starting up the tiny server, starting up the proxy server, and creating a file with several curl commands to get the files godzilla.jpg, home.html, csapp.h, whatIDo.png, and tiny.c.  You should see each request adds a new URL to the cache, except for the file whatIDo.png.  The code should determine that the whatIDo.png object is too large to add to the cache.

B. *Write and test the findCacheItem function.* At this point, don't worry about implementing the LRU policy in findCacheItem. Modify the handleRequest so that it calls the findCacheItem after the call to getRequest. If the item is in the cache, the handleRequest function should simply write the object to the connected socket.

You can test this by running:

driver.sh

It should pass all of the cache test for files in the cache (1 - 5).  It will fail tests 6 and 7 until the LRU policy is implemented.

C. *Write and test the evict and moveToFront functions.* The evict function will be called by the addCacheItem function and moveToFront will be called by findCacheItem.  If you get this working correctly then driver.sh should report that all of the tests pass.

**5.0 Evaluation**

This assignment will be graded out of a total of 90 points:

Basic Proxy: 40 points  
Cache: 40 points  
Style: 10 points

Your code should be properly indented and easy to read.  You may further decompose the functions that are described in the assignment.  You may not drastically redesign the code without discussing it in advance with your instructor.

**6.0 Testing and Debugging**

There are many tools you can use to debug and test your proxy.  Be sure to exercise all code paths and test a representative set of inputs, including base cases, typical cases, and edge cases.

**6.1 Tiny Server**

Your handout directory contains the source code for the CS:APP Tiny web server. While not as powerful as httpd, the  Tiny web server will be easy for you to understand. It's  a reasonable starting point for some of your proxy code. And it's the server that the driver code uses to fetch pages.

**6.2 Curl**

You can use  curl to generate HTTP requests to any server, including your own proxy.  It is an extremely useful debugging tool. For example, if your proxy and tiny are both running on the local machine, tiny is listening on port 15213, and proxy is listening on port 15214, then you can request a page from tiny via your proxy using  
the following curl command:

linux> curl -v --proxy http://localhost:15214 http://localhost:15213/home.html  
\* About to connect() to proxy localhost port 15214 (#0)  
\*   Trying 127.0.0.1... connected  
\* Connected to localhost (127.0.0.1) port 15214 (#0)  
> GET http://localhost:15213/home.html HTTP/1.1  
> User-Agent: curl/7.19.7 (x86\_64-redhat-linux-gnu)...  
> Host: localhost:15213  
> Accept: \*/\*  
> Proxy-Connection: Keep-Alive  
>   
\* HTTP 1.0, assume close after body  
< HTTP/1.0 200 OK  
< Server: Tiny Web Server  
< Content-length: 135  
< Content-type: text/html<html>  
<head><title>test</title></head>  
<body>   
How do you like godzilla?  
<img align="middle" src="godzilla.gif">  
</body>  
</html>  
\* Closing connection #0

**6.3 netcat**

netcat, invoked as ncat or nc , is a versatile network utility. You can use netcat just like telnet, to open connections to servers.  Hence, imagining that your proxy were running on port 12345 you can do something like the following to manually test your proxy:

linux> nc localhost 12345  
GET http://student2.cs.appstate.edu/index.html HTTP/1.0  
^D  
HTTP/1.1 200 OK  
Date: Sun, 21 Apr 2019 22:29:54 GMT  
Server: Apache/2.4.6 (CentOS) PHP/5.6.40 OpenSSL/1.0.2k-fips mod\_perl/2.0.10 Perl/v5.16.3  
Last-Modified: Thu, 05 Oct 2017 18:46:12 GMT  
ETag: "5-55ad12364e206"  
Accept-Ranges: bytes  
Content-Length: 5  
Connection: close  
Content-Type: text/html  
  
test

Notice the user entered the GET request line and then a ctrl-D.

In addition to being able to connect to Web servers, netcat can also operate as a server itself.  With the following command, you can run netcat as a server listening on port 12345:

 linux> nc -l 12345

Once you have set up a netcat server, you can generate a request to a phony object on it through your proxy, and you will  
be able to inspect the exact request that your proxy sent to netcat.

**6.4 Web Browsers**

Eventually you should test your proxy using Mozilla Firefox (or another browser).

To configure Firefox to work with a proxy, visit  
  
    Preferences>Advanced>Network>Settings

Choose the port that your proxy is listening on.  (Most of the ports on the student2 machine are behind the firewall.  However, the ports 15000 through 15040 are open.  You can choose one of these for the short period of time you want to test your proxy using a browser.)

It is exciting to see your proxy working through a real Web browser.  Although the functionality of your proxy will be limited, you will notice that you are able to browse the vast majority of websites through your proxy.

**7. Finally**

You should test your proxy on bad input.  For example, if the origin server  cannot be contacted, your proxy should be able to continue execution, accepting new connection requests.  In addition, if the origin server closes it connection prematurely, your proxy should not crash.  It should also not crash for badly formed uri or invalid methods or versions.  Note that a uri might start with http:// or it might not.  It could simply be a path.