CMPSC 311, Fall 2019
Proxy Lab: Writing a Sequential Caching Web Proxy

Assigned: Monday, Dec 2nd, 2019

Due: Sunday, Dec 15th, 11:00 PM

1 Introduction

A Web proxy is a program that acts as a middleman between a Web browser and an *end server*. Instead of contacting the end server directly to get a Web page, the browser contacts the proxy, which forwards the request on to the end server. When the end 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 web objects. 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 Logistics

This is an individual project. Please refrain from looking up solutions for similar projects online.

3 Handout instructions

Download `proxylab-handout.tar` file from Canvas. Copy the handout file to a protected directory on the Linux machine where you plan to do your work, and then issue the following command:

```
linux> tar xvf proxylab-handout.tar
```

This will generate a handout directory called `proxylab-handout`. The README file describes the various files.

4 Part I: Implementing a sequential web proxy

The first step is implementing a basic sequential proxy that handles HTTP/1.1 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.

4.1 HTTP/1.1 GET requests

When an end user enters a URL such as `http://web.mit.edu/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://web.mit.edu/index.html HTTP/1.1
```

In that case, the proxy should parse the request into at least the following fields: the hostname, `web.mit.edu`; and the path or query and everything following it, `/index.html`. Use the `parse_url` function from hw9. That way, the proxy can determine that it should open a connection to `web.mit.edu` and send an HTTP request of its own starting with a line of the following form:

```
GET /index.html HTTP/1.0
```

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"`.

You should notice in the above example that the web browser's request line ends with `HTTP/1.1`, while the proxy's request line ends with `HTTP/1.0`. Modern web browsers will generate HTTP/1.1 requests, but your proxy should handle them and forward them as HTTP/1.0 requests.

It is important to consider that HTTP requests, even just the subset of HTTP/1.0 GET requests, can be incredibly complicated. The textbook describes certain details of HTTP transactions, but you should refer to RFC 1945 for the complete HTTP/1.0 specification. Ideally your HTTP request parser will be fully robust according to the relevant sections of RFC 1945, except for one detail: while the specification allows for multiline request fields, your proxy is not required to properly handle them. Of course, your proxy should never prematurely abort due to a malformed request.

4.2 Request headers

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

- Always send a `Host` header. While this behavior is technically not sanctioned by the HTTP/1.0 specification, it is necessary to coax sensible responses out of certain Web servers, especially those that use virtual hosting.

  The `Host` header describes the hostname of the end server. For example, to access `http://web.mit.edu/index.html`, your proxy would send the following header:

  ```
  Host: web.mit.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.

- You *may* choose to always send the following `User-Agent` header:

  ```
  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3)
      Gecko/20120305 Firefox/10.0.3
  ```

  The header is provided on two separate lines because it does not fit as a single line in the writeup, but your proxy should send the header as a single line.

  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.

- Always send the following `Connection` header:

  ```
  Connection: close
  ```

- 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.

- There are two other headers `If-Modified-Since` and `If-None-Match` that you should skip if the browser generated these request headers. The reason is these headers are used to handle caching done by the browser, and when included, the server instead of returning a 200 OK status, will return a 304 Not Modified status and will cause the cache in our proxy server to not function properly. Simply eliminate these headers will solve the problem.

Also keep in mind, when analyzing request headers from the browser, the header names are case-insensitive, different browsers might capitalize (or have lower cases) for the same field name. So your parsing should do comparsions that is also case insensitive.

To make your headers work, you will have to skip the browser supplied header for `Connection`, `User-Agent` and `Proxy-Connection`. You should also check if the request header coming from the browser already contains `Host`, if it does, use it; if it doesn't, make sure you add the `Host` header.

For your convenience, the values of the described `User-Agent` header is provided to you as a string constant in `proxy.c`.

Finally, if a browser sends any additional request headers as part of an HTTP request, your proxy should forward them unchanged.

4.3 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://cse-cmpsc311.cse.psu.edu:8080`, in which case your proxy should connect to the host `cse-cmpsc311.cse.psu.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 8081:

```
    linux> ./proxy 8081
```

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 each machine, the script `port-for-user.pl` is provided to help you pick your own personal port number. Use it to generate port number based on your user ID:

```
$ ./port-for-user.pl yuw17
yuw17: 62346
```

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.

5 Part II: Caching your Requests

For the second 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.

5.1 Maximum cache size

The entirety of your proxy's cache should have the following maximum size:

```
MAX_CACHE_SIZE = 4000000
```

When calculating the size of its cache, your proxy must only count bytes used to store the actual web objects; any extraneous bytes, including metadata, should be ignored.

5.2 Maximum object size

Your proxy should only cache web objects that do not exceed the following maximum size:

```
MAX_OBJECT_SIZE = 1000000
```

For your convenience, both size limits are provided as macros in `cache.h`. When you are writing your code, you should always use the Macro Name to refer to these constants instead of using the actual number. This will make it easy to adapt your code to different configurations.

To ensure that you cache only web objects that is correctedly sent through the network, you should only cache an object if all of the following are true:

- Only cache web objects when the server's response status code is 200 OK;
- Only cache web objects when the server supplies the object size in a header ”Content-Length:” (case insensitive);
- Only cache web objects whose size is no bigger than the maximum object size defined by the macro

```
MAX_OBJECT_SIZE
```

- Only cache web objects that you actually received the number of bytes for the web object that matches the size described in the Content-Length header.

The easiest way to implement a correct cache is to allocate a buffer for the active connection that meets the first three conditions stated above and accumulate data as it is received from the server. If the number of bytes received doesn't match expected size for the content(aka not meeting the 4th condition listed above), it is an indication that error has occured during the network transmission and the object is most likely corrupted and should'nt be cached. In that case, make sure you free the memory you are not caching to avoid memory leak.

```
MAX_CACHE_SIZE + MAX_OBJECT_SIZE
```

5.3 Eviction policy

Your proxy's cache should employ an eviction policy that is a least-recently-used (LRU) eviction policy for your sequential proxy server. Notice that both reading an object from the cache and writing it into the cache count as using the object.

5.4 Cache Implementation

We are including `cache.h` and `libcache.a` file you can use directly to implement cache in your proxy server. The Makefile creates two executables, one using implementation from provided libcache.a (called proxy), the other using your implementation in cache.c (called proxycache).

Once your cache works with the supplied cache library, you should implement your own cache.c file to implement the few functions described in `cache.h`. You should implement your cache as a doubly linked list, where each node `CachedItem` has a pointer pointing to the node in front of it and a pointer pointing to the node behind it.

You will mainly need to implement four functions:

1. `void cache_init(CacheList *list);`

   This function will initialize the list to an empty list by setting size to 0, and set first and last to NULL;

2. `void cache_URL(const char *URL, const char *headers, void *item, size_t size`

   This function will add a new node to the front of the doubly linked list. It will use strdup to allocate its own space to store the URL and headers, it will just store and own the pointer pointing to the dynamically allocated memory that stores the binary content of the file in item, it will set the size of the node and update the overall size of the whole CacheList accordingly. The hardest part is when we don't have enough space to cache this item, and we will have to evict stale items in our linked list. We keep the stale items at the tail of the list, we will keep evicting items from the last node in the list, and upate the cache size accordingly until we make enough room to add this new node with content of given size.

Your code should also check the size of this item to see if it exceeded `MAX_OBJECT_SIZE` and if it does, free up the pointer item and don't allocate this node.

Consider all the edge cases for this step: if there was no nodes in the CacheList when we are inserting it (you need to update both first and last pointer in CacheList); if there were nodes but some had to be evicted to make room, you should take care updating the linked list and first/last pointer.

3. `CachedItem *find(const char *URL, CacheList *list);`

   This function will look for a cached item in the CacheList with a matching URL key. And returns the pointer to the nodes if it finds it. If there is no matching node, it returns NULL.

   But that is not all of what this function does. To maintain our CacheList's structure to enforce LRU eviction policy, we should put freshly accessed items to the front of the list. So in this case, if we do find a matching node, and since it is just visited, it should be taken off the middle of the list (or whereever it was), and moved to the very front of the list.

   You should be very careful with all the edge cases for this step: for example, what if the node was already at the front of the list? what about the node was at the end of the list? what about the node was in the middle of the list?

4. `void cache_destruct(CacheList *list);`

   This last function is to clean up the whole linked list and free all memory. It will not be used in your proxylab code since proxy server never terminates and you should never clean the cache, but we will use this function combined with other functions to form a separate unit test set to check the correctness of your implementation.

6 Evaluation

This assignment will be graded out of a total of 60 points:

- *BasicCorrectness*: 20 points for basic proxy operation
- *Cache in Proxy*: 15 points for a working cache that passes the driver test.
- *Cache Implementation*: 15 points for your own correct cache.c code.
- *Realworld website*: 10 points to check your implementation with real browser (firefox) and real world website that you should be able to successfully load (maybe slowly) without crashing/stalling.
- *Style and memory correctness*: up to -20 points to check your style and in particular memory correctness of your program. Do you have memory leak? Are you accesssing memory out of bound? Does your code have good indentation and comments and so on. Does your proxy server have proper error handling so that it doesn't just terminate or crash if anything recoverable went wrong?

6.1 Autograding

Your handout materials include an autograder, called `driver.sh`, that your instructor will use to get preliminary scores for *BasicCorrectness*, and *Cache*. From the `proxylab-handout` directory:

```
linux> ./driver.sh
```

You must run the driver on a Linux machine.

The autograder does only simple checks to confirm that your code is acting like a caching proxy. For the final grade, we will do additional manual testing to see how your proxy deals with real pages. Here is a list of some pages that still uses http protocol (as of December 2nd, 2019) that you can use to test.

- `http://web.mit.edu`
- `http://www.xinhuanet.com/english/`

6.2 Robustness

As always, you must deliver a program that is robust to errors and even malformed or malicious input. Servers are typically long-running processes, and web proxies are no exception. Think carefully about how long-running processes should react to different types of errors. For many kinds of errors, it is certainly inappropriate for your proxy to immediately exit.

Robustness implies other requirements as well, including invulnerability to error cases like segmentation faults and a lack of memory leaks and file descriptor leaks.

7 Testing and debugging

Besides the simple autograder, you will not have any sample inputs or a test program to test your implementation. You will have to come up with your own tests and perhaps even your own testing harness to help you debug your code and decide when you have a correct implementation. This is a valuable skill in the real world, where exact operating conditions are rarely known and reference solutions are often unavailable.

Fortunately 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.

7.1 Tiny web server

Your handout directory the source code for the CS:APP Tiny web server. While not as powerful as `thttpd`, the CS:APP Tiny web server will be easy for you to modify as you see fit. It's also a reasonable starting point for your proxy code. And it's the server that the driver code uses to fetch pages.

7.2 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 8080, and proxy is listening on port 8081, then you can request a page from Tiny via your proxy using the following `curl` command:

```
$ curl -v --proxy localhost:8081 http://localhost:8080
* About to connect() to proxy localhost port 8081 (#0)
*   Trying ::1... Connection refused
*   Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 8081 (#0)
> GET http://localhost:8080/ HTTP/1.1
> User-Agent: curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7 NSS/3.27.1 zlib/1.2.3
> Host: localhost:8080
> Accept: */*
> Proxy-Connection: Keep-Alive
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: Tiny Web Server
< Connection: close
< Content-length: 121
< Content-type: text/html
<
<html>
<head><title>test</title></head>
<body>
<img align="middle" src="godzilla.gif">
Dave O'Hallaron
</body>
</html>
* Closing connection #0
```

7.3 netcat

`netcat`, also known as `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 `localhost` using port 8081 you can do something like the following to manually test your proxy:

```
$ nc -C localhost 8081
GET http://www.cse.psu.edu/~yuw17/cmpsc311/vagrant/Vagrantfile HTTP/1.1

HTTP/1.1 200 OK
Date: Tue, 03 Dec 2019 16:11:41 GMT
Server: Apache
```

```
Last-Modified: Wed, 04 Sep 2019 03:34:17 GMT
ETag: "41a2e68-624-591b1e132e040"
Accept-Ranges: bytes
Content-Length: 1572
Connection: close
Content-Type: text/plain

....
```

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:

```
    sh> 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`.

7.4 Web browsers

Eventually you should test your proxy using the *most recent version* of Mozilla Firefox. Visiting `About Firefox` will automatically update your browser to the most recent version.

To configure Firefox to work with a proxy, visit

```
    Preferences>Advanced>Network>Settings
```

It will be very 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.

An important caveat is that you must be very careful when testing caching using a Web browser. All modern Web browsers have caches of their own, which you should disable before attempting to test your proxy's cache.

If you want to be able to let traffic to localhost (where you are running tiny webserver) to also go through proxy, you will have to manually change the setting at:

```
about:config
network.proxy.allow_hijacking_localhost;true
```

8 Handin instructions

Submit `cache.c` and `proxy.c` file to gradescope.

9 Hints

- Chapters 10 of the textbook contains useful information on system-level I/O, network programming, HTTP protocols.
- RFC 1945 (`http://www.ietf.org/rfc/rfc1945.txt`) is the complete specification for the HTTP/1.0 protocol.
- As discussed in Section 10.11 of your textbook, using standard I/O functions for socket input and output is a problem. Instead, we recommend that you use the Robust I/O (RIO) package, which is provided in the `csapp.c` file in the handout directory.
- The error-handling functions provide in `csapp.c` are not appropriate for your proxy because once a server begins accepting connections, it is not supposed to terminate. You'll need to modify them or write your own.
- As discussed in the Aside on page 964 of the CS:APP3e text, your proxy must ignore SIGPIPE signals and should deal gracefully with `write` operations that return EPIPE errors.
- Sometimes, calling `read` to receive bytes from a socket that has been prematurely closed will cause `read` to return `-1` with `errno` set to `ECONNRESET`. Your proxy should not terminate due to this error either.
- Remember that not all content on the web is ASCII text. Much of the content on the web is binary data, such as images and video. Ensure that you account for binary data when selecting and using functions for network I/O.
- Forward all requests as HTTP/1.0 even if the original request was HTTP/1.1.

Good luck!