Assignment 3: HTTP Web Proxy and Cache

This assignment has you implement a multithreaded HTTP proxy and cache. An HTTP proxy is an intermediary that intercepts each and every HTTP request and (generally) forwards it on to the intended recipient. The servers direct their HTTP responses back to the proxy, which in turn passes them on to the client. In its purest form, the HTTP proxy is little more than a nosy network middleman that reads and propagates all incoming and outgoing HTTP activity.

Here's the neat part, though. When HTTP requests and responses travel through a proxy, the proxy can control what gets passed along. The proxy might, for instance:

- block access to social media products—sites like Google Plus, Twitter, and LinkedIn.
- block access to large documents, like videos and high-definition images, so that slow networks don't become congested and interfere with lightweight applications like email and instant messaging.
- block access to all web sites hosted in Canada. You know, as payback for Justin Bieber.
- strip an HTTP request of all cookie and IP address information before forwarding it to the server as part of some larger effort to anonymize the client.
- intercept all requests for GIF, JPG, and PNG files and instead serve a proxy-stored image of your lovely lecturer.
- cache the HTTP responses to frequently requested, static resources that don't change very often so it can respond to future HTTP requests for the same exact resources without involving the origin servers.
- redirect the user to an intermediate paywall to collect payment for wider access to the Internet, as some airport and coffee shop WiFi systems are known for.

Due: Sunday, January 5th at 11:59 p.m.

Getting The Code

To get your own copy of the starter code, you should copy the master git repository we've set up for you by typing:

```
ssh$ cp /home/assigments/assign3/ ~/ -r
```

If you descend into your **`assign3/slink`** directory, you'll notice a symlink called **`proxy_soln`**, which leads to a copy of the sample executable. You can run **`proxy_soln`** (and your own **`proxy`**) as you'd expect:

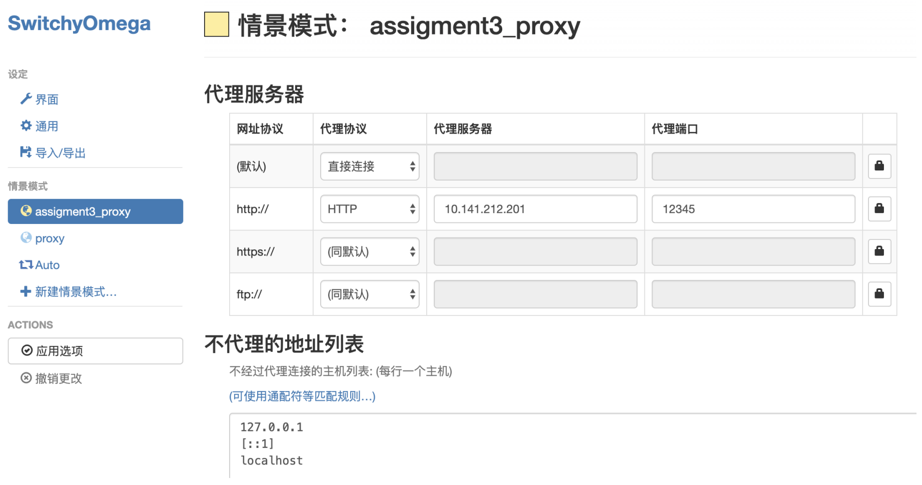
```
ssh> ./slink/proxy_soln
Listening for all incoming traffic on port <port number>.
```

The port number has a default number. If for some reason **`proxy`** says the port number is in use, you can select any other port number between 1024 and 65535 (I'll choose 12345 here) that isn't in use by typing:

```
ssh> ./slink/proxy_soln --port 12345
Listening for all incoming traffic on port
12345.
```

In isolation, **`proxy_soln`** doesn't do very much. In order to see it work its magic, you should download and launch a web browser that allows you to appoint a proxy for just HTTP traffic.

In Chrome, you need to install plug-in (https://chrome.google.com/webstore/detail/proxy-switchyomega/padekgcemlokbadohgkifijomclgjgif?hl=zh-CN), and create a new configuration as the following picture.



情景模式： assigment3_proxy

代理服务器

网址协议	代理协议	代理服务器	代理端口
(默认)	直接连接		
http://	HTTP	10.141.212.201	12345
https://	(同默认)		
ftp://	(同默认)		

不代理的地址列表

不经过代理连接的主机列表: (每行一个主机)

(可使用通配符等匹配规则...)

```
127.0.0.1
[::1]
localhost
```

In Firefox, you can configure it (on Macs) to connect to the Internet through proxy by launching Preferences from the Apple menu, selecting Advanced, selecting Network within Advanced, selecting Connection within Network, and then activating a manual proxy as I have in the screenshot on the right. (On Windows, proxy settings can be configured by selecting Tools → Options).

If you'd like to start small and avoid the browser, you can use **`telnet`** from your own machine to talk HTTP with your proxy, like this (everything I type in is in bold, and everything sent back by the proxy running on **`10.141.212.201:12345`** is italicized):

```
$ telnet 10.141.212.201 12345

Trying 10.141.212.201...

Connected to 10.141.212.201.

Escape character is '^]'.

GET http://xkcd.com/info.0.json HTTP/1.1

Host: xkcd.com



HTTP/1.1 301 Permanently Moved
accept-ranges: bytes
connection: close
content-length: 0
date: Mon, 09 Dec 2019 01:20:20 GMT
location: https://xkcd.com/info.0.json
retry-after: 0
server: Varnish
via: 1.1 varnish
x-cache: HIT
x-cache-hits: 0
x-served-by: cache-hkg17935-HKG
x-timer: S1575854420.393439,VS0,VE0
```

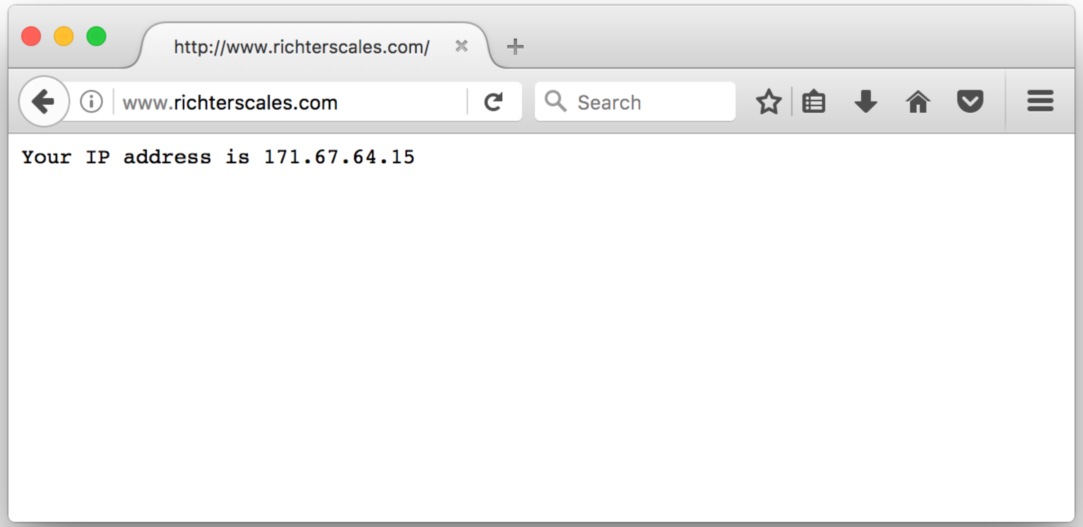
(Note that after you enter **`Host: xkcd.com`**, you need to hit enter twice and wait a few minutes. In this case, the response is a valid one that says the official URL for the resource being fetched involved HTTPS instead of HTTP, but it's nonetheless a valid HTTP conversation.)

All requests for this assignment are HTTP, HTTPS website will fail.

Implementing v1: Sequential `proxy`

Your final product should be a multithreaded HTTP proxy and cache that blocks access to certain domains. As with all nontrivial programs, we're encouraging you to develop through a series of milestones instead of implementing everything in one extended, daredevil swoop. You'll want to read and reread Sections 11.5 and 11.6 of your B&O textbook to ensure a basic understanding of the HTTP protocol.

For the v1 milestone, you shouldn't worry about threads or caching. You should transform the initial code base into a sequential but otherwise legitimate proxy. The code you're starting with responds to **all** HTTP requests with a placeholder status line consisting of an **`"HTTP/1.0"`** version string, a status code of 200, and a curt **`"OK"`** reason message. The response includes an equally curt payload announcing the client's IP address. Once you've configured your browser so that all HTTP traffic is directed toward the relevant port of the **`ssh`** machine you're working on, go ahead and launch **`proxy`** and start visiting any and all web sites. Your proxy should at this point intercept every HTTP request and respond with this (with a different IP address, of course):



```
Your IP address is 171.67.64.15
```

For the v1 milestone, you should upgrade the starter application to be a true proxy—an intermediary that ingests HTTP requests from the client, establishes connections to the origin servers (which are the machines for which the requests are actually intended), passes the HTTP requests on to the origin servers, waits for HTTP responses from these origin servers, and then passes those responses back to the clients. Once the v1 checkpoint has been implemented, your **`proxy`** application should basically be a busybody that intercepts HTTP requests and responses and passes them on to the intended recipients.

Each intercepted HTTP request is passed along to the origin server pretty much as is, save for three small changes.

- You should modify the intercepted request URL within the first line — the request line as it's called — as needed so that when you forward it as part of the request, it includes only the path and not the protocol or the host. The request line of the intercepted HTTP request should look something like this:

  ```
  GET http://news.yahoo.com/science/ HTTP/1.1
  ```

  Of course, **`GET`** might be any one of the legitimate HTTP method names, the protocol might be **`HTTP/1.0`** instead of **`HTTP/1.1`**, and the URL will be any one of a jillion URLs. But provided your browser is configured to direct all HTTP traffic through your proxy, the URLs are guaranteed to include the protocol (e.g. the leading **`"http://"`**) and the host name (e.g. **`news.yahoo.com`**). The protocol and the host name are included whenever the request is directed to a proxy, because the proxy would otherwise have no clue where the forwarded HTTP request should go. When you **do** forward the HTTP request to the origin server, you need to strip the leading **`"http://"`** and the host name from the URL. For the specific example above, the proxy would need to forward the HTTP request on to **`news.yahoo.com`**, and the first line of that forwarded HTTP request would need to look like this:

```
GET /science/ HTTP/1.1
```

I've implemented the `HTTPRequest` class to manage this detail for you automatically (inspect the implementation of `operator<<` in `request.cc` and you'll see), but you need to ensure that you don't break this as you start modifying the code base, because you'll need to change the implementation of `operator<<` once you support proxy chaining for the final milestone.

- You should add a new request header entity named `"x-forwarded-proto"` and set its value to be `"http"`. If `"x-forwarded-proto"` is already included in the request header, then simply add it again.
- You should add a new request header entity called `"x-forwarded-for"` and set its value to be the IP address of the requesting client. If `"x-forwarded-for"` is already present, then you should *extend* its value into a comma-separated chain of IP addresses the request has passed through before arriving at your proxy. (The IP address of the machine you're directly hearing **from** would be appended to the end). **Your reasons for adding these two new fields will become apparent later on, when you support proxy chaining.**

Most of the code you write for your v1 milestone will be confined to `request-handler.h` and `request-handler.cc` files (although you'll want to make a few changes to `request.h/cc` as well). The `HTTPRequestHandler` class you're starting with has just one `public` method, with a placeholder implementation.

You need to familiarize yourself with all of the various classes at your disposal to determine which ones should contribute to the v1 implementation. I repeat: You need to familiarize yourself with all of the various classes at your disposal to determine which ones should contribute to the v1 implementation. Of course, you'll want to leverage the client socket code presented in lecture to open up a connection to the origin server. Your implementation of the one `public` method will evolve into a substantial amount of code—substantial enough that you'll want to decompose and add a good number of `private` methods.

Once you've reached your v1 milestone, you'll be the proud owner of a sequential (but otherwise fully functional) `proxy`. You should visit every popular web site imaginable to ensure the round-trip transactions pass through your proxy without impacting the functionality of the site (caveat: see the note below on sites that require login or are served up via HTTPS). Of course, you can expect the sites to load very slowly, since your proxy has this much parallelism: **zero**. For the moment, however, concern yourself with the networking and the proxy's core functionality, and worry about improving application throughput in later milestones.

Implementing v2: Sequential `proxy` with blacklisting, caching

Once you've built v1, you'll have constructed a genuine HTTP proxy. In practice, proxies are used to either block access to certain web sites, cache static resources that rarely change so they can be served up more quickly, or both.

Why block access to certain web sites? There are several reasons, and here are a few:

- Law firms, for example, don't want their attorneys surfing Yahoo, AOL, or Facebook when they should be working and billing clients.
- Parents don't want their kids to accidentally trip across a certain type of web site.
- Professors configure their browsers to proxy through a university intermediary that itself is authorized to access a wide selection of journals, online textbooks, and other materials—all free of charge—that shouldn't be accessible to the general public. (This is the opposite of blocking, I guess, but the idea is the same).
- Some governments forbid their citizens to visit Facebook, Twitter, The New York Times, and other media sites.
- Microsoft IT might "encourage" its employees to use Bing by blocking access to other search engines during lockdown periods when a new Bing feature is being tested internally.

Why should the proxy maintain copies of static resources (like images and JavaScript files)? Here are two reasons:

- The operative adjective here is *static*. A large fraction of HTTP responses are dynamically generated—after all, the majority of your Facebook, LinkedIn, Google Plus, Flickr, and Instagram feeds are constantly updated—sometimes every few minutes. HTTP responses providing dynamically generated content should never be cached, and the HTTP response headers are very clear about that. But some responses—those serving images, JavaScript files, and CSS files, for instance—are the same for all clients, and stay the same for several hours, days, weeks, months—even years! An HTTP response serving static content usually includes information in its header stating the entire thing is cacheable. Your browser uses this information to keep copies of cacheable documents, and your proxy can too.
- Along the same lines, if a static resource—the omnipresent Google logo, for instance—rarely changes, why should a proxy repeatedly fetch the same image over and over again an unbounded number of times? It's true that browsers won't even bother issuing a request for something in its *own* cache, but users clear their browser caches from time to time (in fact, you should clear it very, very often while testing), and some HTTP clients aren't savvy enough to cache anything at all. By maintaining its own cache, your proxy can drastically reduce network traffic by serving up cached copies when it knows those copies would be exact replicas of what it'd otherwise get from the origin servers. In practice, web proxies are on the same local area network, so if requests for static content don't need to leave the LAN, then it's a win for all parties.

In spite of the long-winded defense of why caching and blacklisting are reasonable features, incorporating support for each is relatively straightforward, provided you confine your changes to the **`request-handler.h`** and **`.cc`** files. In particular, you should just add two **`private`** instance variables—one of type **`HTTPBlacklist`**, and a second of type **`HTTPCache`** to **`HTTPRequestHandler`**.
Once you do that, you should do this:

- Update the **`HTTPRequestHandler`** constructor to construct the embedded **`HTTPBlacklist`**, which itself should be constructed from information inside the **`"blocked-domains.txt"`** file. The implementation of **`HTTPBlacklist`** relies on the C++11 **`regex`** class, and you're welcome to read up on the regular expression support they provide. You're also welcome to ignore the **`blacklist.cc`** file altogether and just use it.

Your **`HTTPRequestHandler`** class would normally forward all requests to the relevant original servers without hesitation. But, if your request handler notices the origin server matches one of the regexes in the **`HTTPBlacklist`**-managed set of verboten domains, you should punt on the forward and immediately respond with a status code of 403 and a payload of **`"Forbidden Content"`**. Whenever you have to respond with your own HTML documents (as opposed to ones generated by the origin servers), just go with a protocol of HTTP/1.0.

- You should update the **`HTTPRequestHandler`** to check the cache to see if you've stored a copy of a previously generated response for the same request. The **`HTTPCache`** class you've been given can be used to see if a valid cache entry exists, repackage a cache entry into **`HTTPResponse`** form, examine an origin-server-provided **`HTTPResponse`** to see if it's cacheable, create new cache entries, and delete expired ones. The current implementation of **`HTTPCache`** can be used as is—at least for this milestone. It uses a combination of HTTP response hashing and timestamps to name the cache entries, and the naming schemes can be gleaned from a quick gander through the **`cache.cc`** file.

  Your to-do item for caching? Before passing the HTTP request on to the origin server, you should check to see if a valid cache entry exist. If it does, just return a copy of it— verbatim!—without bothering to forward the HTTP request. If it does **not**, then you should forward the request as you would have otherwise. If the HTTP response identifies itself as cacheable, then you should cache a copy before propagating it along to the client.

  What's cacheable? The code I've given you makes some decisions—technically off specification, but good enough for our purposes—and implements pretty much everything. In a nutshell, an HTTP response is cacheable if the HTTP request method was **`"GET"`**, the response status code was 200, and the response header was clear that the response is cacheable and can be cached for a reasonably long period of time. You can inspect some of the **`HTTPCache`** method implementations to see the decisions I've made for you, or you can just ignore the implementations for the time being and use the **`HTTPCache`** as an off-the-shelf.

Once you've hit v2, you should once again pelt your proxy with oodles of requests to ensure it still works as before, save for some obvious differences. Web sites matching domain **`regex`**es listed in **`blocked-domains.txt`** should be shot down with a 403, and you should confirm your **`proxy`**'s cache grows to store a good number of documents, sparing the larger Internet from a good amount of superfluous network activity. (Again, to test the caching part, make sure you clear your browser's cache a whole bunch.)

Implementing v3: Concurrent `proxy` with blacklisting and caching

You've implemented your **`HTTPRequestHandler`** class to proxy, block, and cache, but you have yet to work in any multithreading magic. For precisely the same reasons threading worked out so well with your RSS News Feed Aggregator, threading will work miracles when implanted into your **`proxy`**. Virtually all of the multithreading you add will be confined to the **`scheduler.h`** and **`scheduler.cc`** files. These two files will ultimately define and implement an über-sophisticated **`HTTPProxyScheduler`** class, which is responsible for maintaining a list of socket/IP-address pairs to be handled in FIFO fashion by a limited number of threads.

The initial version of **`scheduler.h`**/**`.cc`** provides the lamest scheduler ever: It just passes the buck on to the **`HTTPRequestHandler`**, which proxies, blocks, and caches on the main thread. Calling it a scheduler is an insult to all other schedulers, because it doesn't really schedule anything at all. It just passes each socket/IP-address pair on to its **`HTTPRequestHandler`** underling and blocks until the underling's **`serviceRequest`** method sees the full HTTP transaction through to the last byte transfer.

One extreme solution might just spawn a separate thread within every single call to **`scheduleRequest`**, so that its implementation would go from this:

```
void HTTPProxyScheduler::scheduleRequest(int connectionfd,
                     const string& clientIPAddress) {
 handler.serviceRequest(make_pair(connectionfd, clientIPAddress));
}
```

to this:

```
void HTTPProxyScheduler::scheduleRequest(int connectionfd,
                     const string& clientIPAddress) {
  thread t([this](const pair<int, string>& connection) {
   handler.serviceRequest(connection);
 }, make_pair(connectionfd, clientIPAddress));
 t.detach();
}
```

While the above approach succeeds in getting the request off of the main thread, it doesn't limit the number of threads that can be running at any one time. If your proxy were to receive hundreds of requests in the course of a few seconds—in practice, a very real possibility—the above would create hundreds of threads in the course of those few seconds, and that would be bad. Should the proxy endure an extended burst of incoming traffic—scores of requests per second, sustained over several minutes or even hours, the above would create so many threads that the thread count would immediately exceed a thread-manager-defined maximum.

Therefore, you need to build a **`ThreadPool`** class, which is exactly what you want here. I've included the **`thread-pool.h`** file in the **`assign3`** repositories. You should leverage a single **`ThreadPool`** with 64 worker threads, and use that to elevate your sequential proxy to a multithreaded one. Given a properly working **ThreadPool**, going from sequential to concurrent is actually not very much work at all.

How does one implement this thread pool thing? Well, your **ThreadPool** constructor—at least initially—should do the following:

- launch a single *dispatcher* thread like this (assuming **dt** is a **privatethread** data member):

```
dt = thread([this]() {
  dispatcher();
});
```

- launch a specific number of *worker* threads like this (assuming **wts** is a **private vector<thread>** data member):

```
for (size_t workerID = 0; workerID < numThreads; workerID++) {
  wts[workerID] = thread([this](size_t workerID) {
    worker(workerID);
  }, workerID);
}
```

The implementation of **schedule** should append the provided function pointer (expressed as a **function<void(void)>** , which is the C++ way to classify a function that can be invoked without any arguments) to the end of a queue of such functions. Each time a function is scheduled, the dispatcher thread should be notified. Once the dispatcher has been notified, **schedule** should return right away so even *more* functions can be scheduled.

Aside: Functions that take no arguments at all are called **thunks**. The **function<void(void)>** type is a more general type than **void (*)()**, and can be assigned to anything invokable—a function pointer, or an anonymous function—that doesn't require any arguments.

The implementation of the private **dispatcher** method should loop almost interminably, blocking within each iteration until it has confirmation the queue of outstanding functions is nonempty. It should then wait for a worker thread to become available, select it, mark it as unavailable, dequeue the least recently scheduled function, put a copy of that function in a place where the selected worker (and **only** that worker) can find it, and then signal the worker thread to execute it.

The implementation of the private **worker** method should also loop repeatedly, blocking within each iteration until the dispatcher thread signals it to execute an assigned function (as described above). Once signaled, the worker should go ahead and invoke the function, wait for it to execute, and then mark itself as available so that it can be discovered and selected again (and again, and again) by the dispatcher.

The implementation of **`wait`** should block until all previously-scheduled-but-yet-to-be-executed functions have been executed. The **`ThreadPool`** destructor should wait until all scheduled functions have executed to completion, somehow inform the dispatcher and worker threads to exit (and wait for them to exit), and then otherwise dispose of all **`ThreadPool`** resources.

Your **`ThreadPool`** implementation shouldn't orphan any memory whatsoever. We'll be analyzing your **`ThreadPool`** to ensure no memory is leaked.

Finally, Your **`HTTPProxyScheduler`** class should encapsulate just a single **`HTTPRequestHandler`**, which itself already encapsulates exactly one **`HTTPBlacklist`** and one **`HTTPCache`**. You should stick with just one scheduler, request handler, blacklist, and cache, but because you're now using a **`ThreadPool`** and introducing parallelism, you'll need to implant more synchronization directives to avoid any and all data races. Truth be told, you shouldn't need to protect the blacklist operations, since the blacklist, once constructed, never changes. But you need to ensure concurrent changes to the cache don't actually introduce any races that might threaten the integrity of the cached HTTP responses. In particular, if your proxy gets two competing requests for the same exact resource and you don't protect against race conditions, you may see problems.

Here are some basic requirements:

- You must, of course, ensure there are no race conditions—specifically, that no two threads are trying to search for, access, create, or otherwise manipulate the same cache entry at any one moment.
- You can have at most one open connection for any given request. If two threads are trying to fetch the same document (e.g. the HTTP requests are precisely the same), then one thread must go through the entire examine-cache/fetch-if-not-present/add-cache-entry transaction before the second thread can even look at the cache to see if it's there.

You **should not** lock down the entire cache with a single **`mutex`** for all requests, as that introduces a huge bottleneck into the mix, allows at most one open network connection at a time, and renders your multithreaded application to be essentially sequential. You *could* take the **`map<string, unique_ptr<mutex>>`** approach that the implementation of **`oslock`** and **`osunlock`** takes, but that solution doesn't scale for real proxies, which run uninterrupted for months at a time and cache millions of documents.

Instead, your **`HTTPCache`** implementation should maintain an array of 997 **`mutex`**es, and before you do anything on behalf of a particular request, you should hash it and acquire the **`mutex`** at the index equal to the hash code modulo 997. You should be able to inspect the initial implementation of the **`HTTPCache`** and figure out how to surface a hash code and use that to decide which **`mutex`** guards any particular request. A specific **`HTTPRequest`** will always map to the same **`mutex`**, which guarantees safety; different

`HTTPRequest`s may very, very occasionally map to the same **`mutex`**, but we're willing to live with that, since it happens so infrequently.

I've ensured that the starting code base relies on thread safe versions of functions (**`gethostbyname_r`** instead of **`gethostbyname`**, **`readdir_r`** instead of **`readdir`**), so you don't have to worry about any of that. (Note your **`assign3`** repo includes **`client-socket.[h/cc]`**, updated to use **`gethostbyname_r`**.)

Implementing v4: Concurrent `proxy` with blacklisting, caching, and proxy chaining

Some proxies elect to forward their requests not to the origin servers, but instead to secondary proxies. Chaining proxies makes it possible to more fully conceal your web surfing activity, particularly if you pass through proxies that pledge to anonymize your IP address, cookie jar, etc. A proxied proxy might even have more noble intentions—to simply rely on the services of an existing proxy while providing a few more—better caching, custom blacklisting, and so forth—to the client.

The **`proxy_soln`** we've supplied you allows for a secondary proxy to be specified, as with this:

```
ssh> ./slink/proxy_soln --proxy-server 10.141.212.129
Listening for all incoming traffic on port 51630.
```

Requests will be directed toward another proxy at **`10.141.212.129: 51630`**.

Provided a second proxy is running on **`10.141.212.129`** and listening to port **`51630`**, the proxy running on **`10.141.212.129`** would forward all HTTP requests—unmodified, save for the updates to the **`"x-forwarded-proto"`** and **`"x-forwarded-for"`** header fields—on to the proxy running on **`10.141.212.129:51630`**, which for all we know forwards to another proxy!

We actually don't require that the secondary proxy be listening on the same port number, so something like this might be a legal chain:

```
ssh> ./slink/proxy_soln --proxy-server 10.141.212.129 --proxy-port
23456

Listening for all incoming traffic on port 51630.
```

Requests will be directed toward another proxy at **`10.141.212.129:23456`**.

In that case, the **`10.141.212.129:23456`** proxy would forward all requests to the proxy that's presumably listening to port **`23456`** on **`10.141.212.129`**. (If the **`--proxy--port`** option isn't specified, then the proxy assumes the same port number it's using is used by the secondary.)

The **`HTTPProxy`** class we've given you already knows how to parse these additional **`--proxy-server`** and **`--proxy-port`** flags, but it doesn't do anything with them. You're to update the hierarchy of classes to allow for the possibility that a (or several) secondary proxy is being used, and if so, to forward all requests (as is, except for the modifications to the **`"x-forwarded-proto"`** and **`"x-forwarded-for"`** headers) on to the secondary proxy. This'll require you to extend the signatures of many methods and/or add methods to the hierarchy of classes to allow for the possibility that requests will be forwarded to another proxy instead of the origin servers. If you notice a chained set of proxy IP addresses that lead to a cycle (even if the port numbers are different), you should respond with a status code of **504**.

Additional Tidbits

- You should *absolutely* add logging code and publish it to standard out. We won't be autograding the logging portion of this assignment, but you should still add tons of logging so that you can confirm your proxy application is actually moving and getting stuff done. (For obvious reasons, your logging code should be thread-safe).
- You can assume your browser and all web sites are solid and respect HTTP request and response protocols. While testing, you should hit as many sites as possible, sticking to major (HTTP, not HTTPS) web products like **`www.berkeley.edu`**, **`www.sfgate.com`**, and so forth. You should avoid sites that require a login or some other form of authentication, since they'll likely mingle HTTP and HTTPS requests.
- Your proxy will intercept all HTTP traffic, but it won't even see any HTTPS traffic. As your surf the net, note whether the site switches over to HTTPS, lest you think your proxy isn't actually doing anything, when in fact it's not because it's not supposed to be. You'll probably want to avoid web sites like **`www.google.com`** , **`www.facebook.com,`** and **`www.yahoo.com`** while testing your proxy, since they're all HTTPS as far as I can tell.
- Your **`proxy`** application maintains its cache in a subdirectory of your home directory called **`.proxy-cache-ubuntu16t.`** The accumulation of all cache entries might very well amount to megabytes of data over the course of the next week, so you should delete that **`.proxy-cache-ubuntu16t`** by invoking your proxy with the **`--clear-cache`** flag.
- If you want to impose a maximum time-to-live value on all cacheable responses, you can invoke your proxy with **`--max-age <max-ttl>`**. If you go with a 0, then the cache is turned off completely. If you go with a number like 10, then that would means that cacheable items can only live in the cache for 10 seconds before they're considered invalid.
- Note that responses to **`HEAD`** requests—as opposed to responses to **`GET`** and **`POST`** requests— never include a payload, even if the response header includes a content length. Make sure you circumvent the call to **`ingestPayload`** for **`HEAD`** requests, else your proxy will get held up once the first **`HEAD`** request is intercepted.
- Your **`proxy`** application should, in theory, run until you explicitly quit by pressing ctrl-C. A real proxy would be polite enough to wait until all outstanding proxy requests have been handled, and it

would also engage in a bidirectional rendezvous with the scheduler, allowing it the opportunity to bring down the **`ThreadPool`** a little more gracefully. You don’t need to worry about this at all—just kill the proxy application without regard for any cleanup.

I hope you enjoy the assignment as much as I’ve enjoyed developing it. It’s genuinely thrilling to know that all of you can implement something as sophisticated as an industrial-strength proxy.