Assignment 6: HTTP Web Proxy and Cache

Your penultimate CS110 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 sites—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 CS110 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.

Building a proxy is no small task, but with your smarts and my love to guide you, I am absolutely confident you can do it.

Due: Friday, March 6th at 11:59 p.m.

Getting The Code

Go ahead and clone the mercurial repository we've set up for you by typing:

```
myth22> hg clone /usr/class/cs110/repos/assign6/$USER assign6
```

Compile often, test incrementally and almost as often as you compile, `hg commit` so you don't lose your work if someone accidentally crushes your laptop, and run `/usr/class/cs110/tools/submit` when you're done.

If you descend into your assign6 directory, you'll notice a symlink called `http-proxy-soln`, which leads to a copy of the sample executable. You can run `http-proxy-soln` (and your own `http-proxy`) as you'd expect:

```
myth22> ./http-proxy-soln
Listening for all incoming traffic on port <port number>.
```

The port number issued depends on your SUNet ID, and with very high probability, you'll be the only one ever assigned it. If for some reason `http-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:

```
myth22> ./http-proxy-soln --port 12345
Listening for all incoming traffic on port 12345.
```

In isolation, `http-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. I'm recommending you download Firefox, because I've used it for two years now to specifically to exercise the `http-proxy` code base, and it's worked very well for me. Once you download and launch Firefox, you can configure it to connect to the Internet through `http-proxy` by launching the **Preferences Panel**, selecting **Advanced**, selecting **Network** within **Advanced**, selecting **Connection Settings** 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**).

Of course, you should enter the `myth` machine you're working on (and you should get in the habit of `ssh`'ing into the same exact `myth` machine for the next week so you don't have to continually change these settings), and you should enter the port number that your `http-proxy` is listening to.

Of course, you can use whatever web browser you want to, but I'm recommending Firefox for a few reasons. Here are two of them:

- Most of you don't use Firefox by default, so you won't need to manually toggle proxy settings on and off to surf the Internet using whatever browser it is you normally use. Firefox can be your CS110 browser for this assignment cycle, and Chrome, Safari, Internet Explorer or whatever it is you use normally can be your default.
- Some other browsers don't allow you to configure browser-only proxy settings, but instead prompt you to configure computer-wide proxy settings for all HTTP traffic—for all browsers, Dropbox and/or iCloud synchronization, iTunes downloads, and so forth. You don't want that level of interference.

If you'd like to start small and avoid the browser, you can use `telnet` from your own machine to converse to your proxy, like this (everything I type in is in bold, and everything sent back by the proxy running on `myth22:12345` is italicized):

```
$ telnet myth22.stanford.edu 12345
Trying 171.64.15.13...
Connected to myth22.stanford.edu.
Escape character is '^]'.
GET http://graph.facebook.com/stanford HTTP/1.0
Host: graph.facebook.com

HTTP/1.1 200 OK
Facebook-API-Version: v1.0
ETag: ''a4b17ae2e353a3c409122c0ec42b76279aed500b''
Content-Type: text/javascript; charset=UTF-8
Pragma: no-cache
Access-Control-Allow-Origin: *
X-FB-Rev: 1486784
Cache-Control: private, no-cache, no-store, must-revalidate
Expires: Sat, 01 Jan 2000 00:00:00 GMT
Date: Mon, 10 Nov 2014 00:23:35 GMT
```

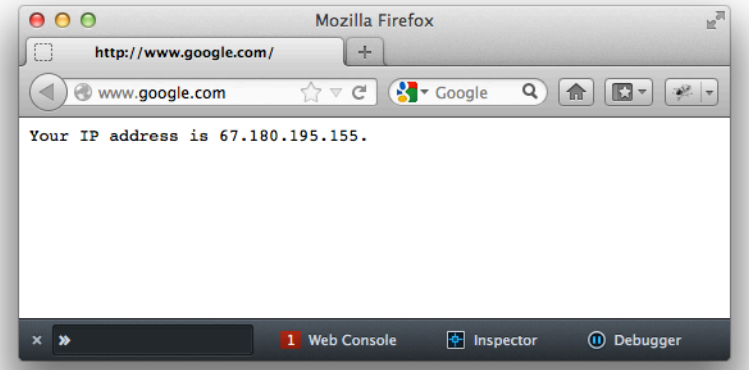
```
    Connection: close
    Content-Length: 2957

    {''id'':''6192688417'',''about'':''Welcome to Stanford! Like us to see
our best photos,
    videos and stories in your newsfeed.'', ''can_post'':false,
    ''category'':''Education'',''category_list'':[{''id'':''108051929285833'',''name'':''
&
    University''}],''checkins'':415773,''cover'': {''cover_id'':''10151995423148418'',
    ''offset_x'':0, ''offset_y'':0, ''source'':''https:\/\/
scontent-b.xx.fbcdn.net\/hphotos-
    prn2\/t31.0-8\/s720x720\/1403201_10151995423148418_507508275_o.jpg''},
    ''description'':''Located between San Francisco and San Jose in the
heart of Silicon
    Valley, Stanford University is recognized as one of the world's leading
research and
    teaching institutions.\n\nStanford's official Facebook page is curated
by
    http:\/\/ucomm.stanford.edu.\n\nComment policy: Stanford welcomes the
    community\u2019s contributions

    some payload omitted to save space
    {''lot'':0, ''street'':0, ''valet'':0},
    ''public_transit'':''http:\/\/transportation.stanford.edu'',
    ''talking_about_count'':12875, ''username'':''stanford'': ''website'':
    ''http:\/\/www.stanford.edu'', ''were_here_count'':415773}
    Connection closed by foreign host.
    $
```

(Note that after you enter `Host: graph.facebook.com`, you need to hit enter twice.)

Implementing v1: Sequential `http-proxy`

Your final product should be a multithreaded HTTP proxy and cache that blocks access to certain domains. As with a few previous assignments, 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 `myth` machine you're working on, go ahead and launch `http-proxy` and start visiting any and all web sites. Your proxy should intercept every HTTP request and respond with something like this:



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 (those're 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 `http-proxy` application should basically be a busybody application that nosily 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, 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 this 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.

- You should add a 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.

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, a placeholder implementation for that method, and that's it. You will 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` helper methods.

Once you've reached your v1 milestone, you'll be the proud owner of a sequential (but otherwise fully functional) `http-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. 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 `http-proxy` with blacklisting, caching

Once you've built `http-proxy`, v1, you'll have constructed a genuine HTTP proxy. (You'll also have gotten a good amount of experience with network and socket API programming, which is all kinds of awesome). 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 doesn'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''`.

- You should update the `HTTPRequestHandler` to check the cache to see if you're storing a copy of a previously generated response. The `HTTPCache` class you've been given can be used to see if a valid cache entry exists, rehydrate a cache entry into `HTTPResponse` form, examine a 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 `http-proxy`'s cache grows to store a good number of documents, sparing the larger Internetz from a good amount of superfluous network activity. (Again, to test the caching part, make sure you clear your browser's cache a whole lot—you might even set the browser cache size to 0 so the browser itself never caches anything and all requests are forward to your proxy.)

Implementing v3: Concurrent `http-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 `http-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();
    }
```

The **time-server** examples presented in the lecture slides take this approach, although we go with an anonymous function here that wraps a call to **serviceRequest**. (Note that **this** needs to be captured, else the anonymous thread routine won't have access to the **handler** member variable).

The above solution doesn't limit the number of threads that can be running at any one time, though. If your proxy were to receive hundreds of requests in the course of a few seconds—in practice, a very real possibility—the above approach 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 approach would create so many threads that the thread count would soon exceed a thread-manager-defined maximum. Of course, the above approach succeeds in getting the request off of the main thread (which is huge), but we can't employ an unbounded number of threads to do that. You'll paralyze the thread manager if you do.

Fortunately, you built a `ThreadPool` class for Assignment 5, which is exactly what you want to employ here. I've included the `thread-pool.h` file in the `assign6` repositories, and I've updated the `Makefile` to link against my working solution of the `ThreadPool` class. You should leverage a single `ThreadPool` with 16 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.

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 file system—changes managed by all of the file system tomfoolery used to implement the `HTTPCache`—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 (you probably took that approach in Assignment 4 to manage per-server connection limits as well), 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 1001 `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 1001. 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 figure out 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. In past quarters, I've made students make these changes, but they've resented me for it, so I've backed down and ensured that a reentrant version of a function is used whenever there is one. (Note your `assign6` repo includes `client-socket.[h/cc]`, updated to use `gethostbyname_r`.)

Implementing v4: Concurrent `http-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 services—better caching, custom blacklisting, and so forth—to the client.

The `http-proxy-soln` we've supplied you allows for a secondary proxy to be specified,

as with this:

```
myth22> ./http-proxy-soln --proxy-server myth12.stanford.edu
Listening for all incoming traffic on port 43383.
```

Requests will be directed toward another proxy at `myth12.stanford.edu:43383`.

Provided a second proxy is running on `myth12` and listening to port 43383, the proxy running on `myth22` 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 `myth12:43383`, 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:

```
myth22> ./http-proxy-soln --proxy-server myth12.stanford.edu
--proxy-port 12345
Listening for all incoming traffic on port 43383.
```

Requests will be directed toward another proxy at `myth12.stanford.edu:12345`.

In that case, the `myth22:43383` proxy would forward all requests to the proxy that's presumably listening to port `12345` on `myth12.stanford.edu`. (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 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—even if the port numbers are different—that lead to a cycle, you should respond with a status code of 504.)

For fun, we're supplying a python script called `run-proxy-farm.py`, which can be used to manage a farm of proxies that forward to each other. One you have proxy chaining implemented, open the python script up, update the `HOSTS` variable to be a list of one or

more myth machine numbers (e.g. `HOSTS = [14, 15, 18, 2]`) to get a daisy chain of `http-proxy` processes running on the different hosts.

Additional Tidbits

- You should *absolutely* add logging code and publish it to standard out. We won't be auto-grading 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 web products like `www.wikipedia.org`, `www.apple.com`, `www.sfgate.com`, `www.stanford.edu`, as 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 supposed to be. You'll probably want to avoid web sites like `www.google.com` and `www.yahoo.com` while testing your proxy, since they're all HTTPS as far as I can tell.
- Your `http-proxy` application maintains its cache in a subdirectory of your home directory called `.http-proxy-cache-myth<num>`. The accumulation of all cache entries might very well amount to megabytes of data over the course of the next two weeks. You should delete that `.http-proxy-cache-myth<num>` subdirectory from time to time, and once you've emerged victorious and handed the assignment in. Of course, you need to be extremely careful when deleting anything from your file system. My recommendation is that you descend into `.http-proxy-cache-myth<num>`, confirm that you're there, and only then issue the remove command needed to recursively delete everything below.
- 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 `http-proxy` application should, in theory, run until you explicitly quit by pressing either ctrl-Z or 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, particularly in light of the fact that many of you took CS106B and CS106X less than a year ago.