PKU-ICS
Proxy Lab: Writing a Caching Web Proxy

Please note the unusual last day for this lab.

1 Introduction

A proxy server is a computer program that acts as an intermediary between clients making requests to access resources and the servers that satisfy those requests by serving content. A web proxy is a special type of proxy server whose clients are typically web browsers and whose servers are the same servers that browsers use. When a web browser uses a proxy, it contacts the proxy instead of communicating directly with the web server; the proxy forwards its client's request to the web server, reads the server's response, then forwards the response to the client.

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. A proxy may also perform translations on pages, for example, to make them viewable on web-enabled phones. Importantly, proxies are used 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 upgrade your proxy to deal with multiple concurrent connections. This will introduce you to dealing with concurrency, a crucial systems concept. In the third and last 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. You can do your work on one of the ICS class machines or on your own Linux machine.

3 Handout instructions

As usual, start by downloading the lab handout (`proxylab-handout.tar`) from Autolab and extracting it into the directory in which you plan to work—issue the following command:

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

If possible, extract the files directly onto a Linux machine; some operating systems and file transfer programs clobber Unix file system permission bits.

3.1 Project structure overview

The handout directory contains a README file, you should look through it to understand what each file is used for. Pay attention to the usage of `driver.sh`, `webdriver_test.py` and `tiny/tiny`.

3.2 Robust I/O package

The handout directory contains the files `csapp.c` and `csapp.h`, which comprise the CS:APP package discussed in the CS:APP3e textbook. In addition to various error-handling wrapper functions and helper functions, the CS:APP package includes the robust I/O (RIO) package. You should use the RIO package instead of vanilla POSIX I/O functions, such as `read`, `write`, `fread`, and `fwrite`.

Do not feel obligated to use the RIO functions or anything at all from the CS:APP package. In fact, feel free to create your own version of a robust I/O package if you find the provided code deficient in any way. Moreover, keep in mind that the error-handling functions in CS:APP may not be appropriate for use in your proxy. Before blindly using wrapper functions or writing any of your own, carefully consider the proper action a server should take on each particular error.

3.3 Modularity

The skeleton file `proxy.c` is provided in the handout. `proxy.c` contains a `main` function that does practically nothing, and you should fill in that file with the guts of your implementation. Modularity, though, should be an important consideration, and it is permissible and encouraged for you to separate the individual modules of your implementation into different files. For example, your cache should be largely (or completely) decoupled from the rest of your proxy, so one popular idea is to move the implementation of the cache into separate files like `cache.c` and `cache.h`.

3.4 Makefile

Since you are free to add your own source files for this lab, you are responsible for updating the Makefile. The entire project should compile without warnings (you may want to use the `-Werror` flag), and you will want to determine the appropriate set of compilation flags (including optimization, linking, and debugging flags) for your final submission.

4 Part I: Implementing a sequential web proxy

The first step is implementing a basic sequential proxy that handles HTTP/1.0 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/1.0 GET requests

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 at least the following fields: the hostname, `www.cmu.edu`; and the path or query and everything following it, `/hub/index.html`. 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/1.0
```

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

Request headers are very important elements of an HTTP request. Headers are essentially key-value pairs provided line-by-line following the first request line of an HTTP request. Of particular importance for this lab are the Host, User-Agent, Connection, and Proxy-Connection headers. Your proxy must perform the following operations with regard to the listed HTTP request 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 web server your proxy is trying to access. 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.

- 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.

With the exception of the `Host` header, your proxy should ignore the values of the request headers described above; instead, your proxy should always send the headers this document specifies.

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://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 12345:

```
> ./proxy 12345
```

You will have to supply a port number every time you wish to test your proxy by running it. You may select any non-privileged port (greater than 1,024 and less than 65,536) as long as it is not used by other processes.

5 Part II: Dealing with multiple concurrent requests

Production web proxies usually do not process requests sequentially; they process multiple requests in parallel. This is particularly important when handling a single request can involve a lengthy delay (as it might when contacting a remote web server). While your proxy waits for a response from the remote web server, it can work on a pending request from another client. Thus, once you have a working sequential proxy, you should alter it to simultaneously handle multiple requests.

5.1 POSIX Threads

You will use the POSIX Threads (Pthreads) library to spawn threads that will execute in parallel to serve multiple simultaneous requests. A simple way to implement concurrent request service is to spawn a new thread to process each new incoming request. In this architecture, the main server thread simply accepts connections and spawns off independent worker threads that deal with each request to completion and terminate when they are done. Other designs are also viable: you might alternatively decide to have your proxy create a pool of worker threads from the start. You may use any architecture you wish as long as your proxy exhibits true concurrency, but spawning a new worker thread for each request is the simplest and historically most common method.

The basic usage of Pthreads involves the implementation of a function that will serve as the start routine for new threads. Once a start routine exists, you can use the `pthread_create` function to create and start a new thread. New threads are by default joinable, which means that another thread must clean up spare resources after the thread exits, similar to how an exited process must be reaped by a call to `wait`. Luckily, it is possible to detach threads, meaning spare resources are automatically reaped upon thread exit. To properly detach threads, the first line of the start routine should be as follows:

```
    pthread_detach(pthread_self());
```

5.2 Race conditions

While multithreading will almost certainly improve the performance of your web proxy, concurrency comes at a price: the threat of race conditions. Race conditions most often arise when there is a shared resource between multiple threads. You must find ways to avoid race conditions in your concurrent proxy. That will likely involve both minimizing shared resources and synchronizing access to shared resources. Synchronization involves the use of objects called locks, which come in many varieties. The Pthreads library contains all of the locking primitives you might need for synchronization in your proxy. See your textbook for details.

5.3 Thread safety

The `open_clientfd` and `open_listenfd` functions described in the CS:APP3e textbook are thread safe are based on the modern and protocol-independent `getaddrinfo` function, and thus are thread safe.

6 Part III: Caching web objects

For the final part of the lab, you will add a cache to your proxy that will keep 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.

6.1 Maximum cache size

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

```
MAX_CACHE_SIZE = 1 MiB
```

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.

6.2 Maximum object size

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

```
MAX_OBJECT_SIZE = 100 KiB
```

For your convenience, both size limits are provided as macros in `proxy.c`.

The easiest way to implement a correct cache is to allocate a buffer for each active connection and accumulate data as it is received from the server. If the size of the buffer ever exceeds the maximum object size, the buffer can be discarded. If the entirety of the web server's response is read before the maximum object size is exceeded, then the object can be cached. Using this scheme, the maximum amount of data your proxy will ever use for web objects is the following, where `T` is the maximum number of active connections:

```
MAX_CACHE_SIZE + T * MAX_OBJECT_SIZE
```

6.3 Eviction policy

Your proxy's cache should employ an eviction policy that approximates a least-recently-used (LRU) eviction policy. It doesn't have to be strictly LRU, but it should be something reasonably close. Note that both reading an object and writing it count as using the object.

6.4 Synchronization

Accesses to the cache must be thread-safe, and ensuring that cache access is free of race conditions will likely be the more interesting aspect of this part of the lab. As a matter of fact, there is a special requirement that multiple threads must be able to simultaneously read from the cache. Of course, only one thread should be permitted to write to the cache at a time, but that restriction must not exist for readers.

For this reason, protecting accesses to the cache with one large exclusive lock is not an acceptable solution. You may want to explore options such as partitioning the cache, using Pthreads readers-writers locks, or using semaphores to implement your own readers-writers solution. In either, the fact that you don't have to implement a strictly LRU eviction policy will give you some flexibility in supporting multiple readers.

7 Part IV: Handling HTTPS Requests

In the previous parts, you have implemented an efficient HTTP proxy, which is a great job! However, data transferred using HTTP is just plain text and can be read by anyone in the middle, including the proxy. What if the client wants a secure connection to the target server? The answer is HTTPS (HyperText Transfer Protocol Secure), which starts a connection with a SSL/TLS handshake. During the handshake, a secret key is exchanged between client and server, and from then on all the transferred data is encrypted with that key. Nowadays, HTTPS is widely used in replace of HTTP to ensure the security. Let's enable our proxy to handle HTTPS requests!

Note that this section counts only 2 points, you may pause and consider whether to continue. And of course you are welcome to challenge yourself and have fun!

To implement this part, here are the steps you need to follow:

First, when the client needs to start a HTTPS connection, the client should send a CONNECT request to the proxy server. A CONNECT request looks like this:

```
CONNECT <target_host>:<port> HTTP/1.1
Host: <target_host>:<port>
User-Agent: ...
(some other information)
```

Second, after the proxy server receives the request, and after the proxy server successfully connects to the upstream (here, you can assume that the proxy server can always connect to the upstream successfully), you need to let the proxy server send a response to the client with a 200 Connection Established code. The format of the response looks like this:

```
HTTP/1.1 200 Connection Established
```

(Hint: this response should end with `"\r\n\r\n"`)

Since then, the connection is established, and your last step is to let the proxy server implement the following two functions at the same time:

1.receive data from the upstream and send it to the client.

2.receive data from the client and send it to the upstream.

Hint: When the proxy notices that the client (the upstream) has closed the TCP connection, the proxy should also close the upstream (the client) connection and release resources. But to be simple, we don't request you to implement this, but you should at least make sure that the proxy doesn't crash when the client or the upstream closes the connection.

You don't need to implement cache in this part, and also you can't in theory. :)

8 Evaluation

This assignment will be graded out of a total of 100 points, which will be awarded based on the following criteria:

- *BasicCorrectness*: 40 points for basic proxy operation (autograded)
- *Concurrency*: 15 points for handling concurrent requests (autograded)
- *Cache*: 15 points for a working cache (autograded)
- *RealPages*: 20 points for passing browser-based tests: total 10 tests, 2pt each, including 1 test for Part IV (autograded)
- *Style*: 10 points for style (manual)

8.1 Preparation

Before you run autograder, some packages need to be installed. Execute following commands in your Linux machine to install them.

```
> pip install selenium

> wget https://chromedriver.storage.googleapis.com/107.0.5304.62/chromedriver_linux64.zip
> unzip chromedriver_linux64.zip
> chmod +x chromedriver
> sudo mv chromedriver /usr/local/bin/chromedriver

> wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
> sudo dpkg -i google-chrome-stable_current_amd64.deb

> sudo apt update
> sudo apt --fix-broken install
```

All these packages have been installed in class machine. So if you write your code in class machine, you can just skip it.

8.2 Autograding

Your handout materials include an autograder, called `driver.sh`, that Autolab will use to assign scores for *BasicCorrectness*, *Concurrency*, *Cache*, and *RealPages*. From the `proxylab-handout` directory:

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

After running it, you will see your scores for each part. If you pass all the tests in `driver.sh`, you are done! However, you are more likely to fail a few (or all the) tests at first. Don't be frustrated, refer to next section for debugging help.

Note: the *RealPages* test will trigger a chromium browser to run browser-based tests. If you are on the ICS class machine or other systems without GUI, you will not see the browser window, but it should not affect your score. It is fully automated and we will use it on Autolab to grade your handin.

8.3 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.

8.4 Style

Style points will be awarded based on the usual criteria. Proper error handling is as important as ever, and modularity is of particular importance for this lab, as there will be a significant amount of code. You should also strive for portability.

9 Testing and debugging

Besides the simple autograder, you will not have any sample inputs to test your implementation. You may 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.

9.1 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.

9.2 `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: 120
< 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
```

9.3 netcat

netcat, also known as nc, is a versatile network utility. You can use netcat just like telnet, to open connections to servers. Hence, if your proxy is running on port 12345, you can execute the following command to manually test your proxy:

```
linux> nc 127.0.0.1 12345
GET http://www.cmu.edu/hub/index.html HTTP/1.0

HTTP/1.1 200 OK
...
```

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.

9.4 Web browsers

Eventually you should test your proxy using one of the modern Web browsers.

You need to configure proxy settings in your browser. 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 many 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.

We also provide you a handy utility called `webdriver_test.py`, which will help you check the correctness of your proxy with several browser-based tests. These are exactly the same tests used for *RealPages* part in autograder.

To use this utility, run following commands from `proxylab-handout` directory:

```
> cd tiny && ./tiny <tiny_port> &
> cd ..
> ./proxy <proxy_port> &
> ./webdriver_test.py <proxy_port> <tiny_port>
```

Above commands run your proxy as well as Tiny (which serves as the backend server of these tests) in background, and then start the test in a chromium browser with the proxy settings configured. Testing will stop on your first failure. You can rerun any test by clicking on the corresponding "run" button on the left panel in the webpage.

Note that you must `cd tiny/` before starting Tiny, becuase it serves content under current working directory.

9.5 Web browsers from remote (Optional)

If you do your work on a remote machine, like the ICS class machine, you can't see the graphic debugging information provided by `webdriver_test.py`. To work around the problem, you can do port forwarding and debug on your local webbrowser. Here we recommand you use Google Chrome and follow these steps:

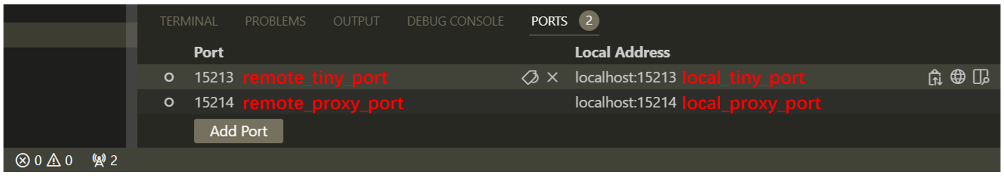
First, run the following steps on **remote** command line as you do in the previous section:

```
> cd tiny && ./tiny <remote_tiny_port> &
> cd ..
> ./proxy <remote_proxy_port> &
> ./webdriver_test.py <remote_proxy_port> <remote_tiny_port>
```

Then, set up port forwarding by running the following command on **local** machine, using `ssh` command with `-L` flag:

```
> ssh <username>@autolab.pku.edu.cn -p <port> \
    -L 127.0.0.1:<local_tiny_port>:127.0.0.1:<remote_tiny_port> \
    -L 127.0.0.1:<local_proxy_port>:127.0.0.1:<remote_proxy_port>
```

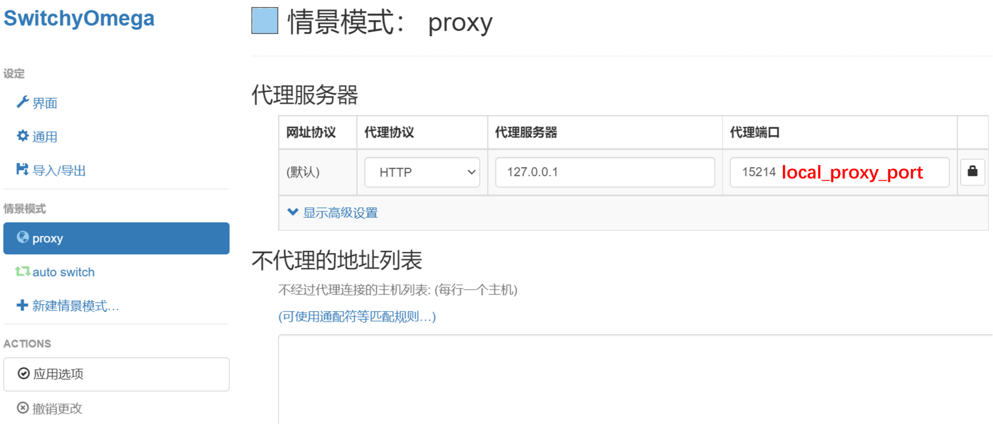
Alternatively, you can also implement port forwarding in VSCode like this:



Port	Local Address
15213 remote_tiny_port	localhost:15213 local_tiny_port
15214 remote_proxy_port	localhost:15214 local_proxy_port

In order to prevent possible bind failure, please use high port number (port number larger than 49152 is safe) while binding. After this, you can access tiny server from <local_tiny_port>, and access proxy from <local_proxy_port> on local machine like you do on the remote machine.

The remaining problem is how to set up the proxy on your local machine. Simply changing the system proxy doesn't work some times because it will bypass proxy for localhost. Here we recommand you use a Chrome extension called SwitchyOmega, and configure it like this:



情景模式： 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)	HTTP	127.0.0.1	15214 local_proxy_port

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...)

If the non-proxied address is not empty, please clear it before proceeding. After all these are done, access the following url and enjoy!
http://127.0.0.1.nip.io:<remote_tiny_port>/browser-testbench/index.html

10 Handin instructions

The provided Makefile includes functionality to build your final handin file. Issue the following command from your working directory:

```
> make handin
```

The output is the file ../proxylab-handin.tar.gz. Simply upload it to Autolab. Autolab will use your Makefile to rebuild your proxy from scratch, discarding any binaries or object files in your handin directory.

11 Resources

- Chapters 11-13 of the textbook contains useful information on system-level I/O, network programming, HTTP protocols, and concurrent programming.
- RFC 1945 (http://www.ietf.org/rfc/rfc1945.txt) is the complete specification for the HTTP/1.0 protocol.

12 Hints

- Your proxy will have to do something about the generation of the SIGPIPE signal. The kernel will sometimes deliver a SIGPIPE to a process that has a handle to a socket which has been broken. Although the default action for a process that receives SIGPIPE is to terminate, your proxy should not terminate due to that signal. (To generate the SIGPIPE signal in Firefox, press F5 twice in quick succession to reload the page.)
- 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.
- Sometimes, calling write to send bytes on a socket that has been prematurely closed will cause write to return -1 with errno set to EPIPE. Your proxy should not terminate due to this error.
- 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 and cache (e.g. use memcpy instead of strcpy).
- Forward all requests as HTTP/1.0 even if the original request was HTTP/1.1.
- Modern web browsers have useful debugging tools you may take advantage of. For example, the keyboard shortcut to the the developer tools in Firefox and Chrome is Ctrl+Shift+I. The network tab allows you to view the status of network requests, which is useful to troubleshoot *RealPages*.
- To avoid overflowing the Autolab server, total number of processes is limited to 100 during autograding. This will not be a problem as long as your proxy use resources reasonably (e.g. terminating unused processes). If you exceed this limit, your proxy, Tiny or chromium browser (in *RealPages* part) may crash.

Good luck!