COMP 321: Introduction to Computer Systems

Project 6: Web Proxy
Assigned: 3/30/17, **Due: 4/20/17**

Important: This project may be done individually or in pairs. Be sure to carefully read the course policies for assignments (including the honor code policy) on the assignments page of the course web site:

`http://www.clear.rice.edu/comp321/html/assignments.html`

Overview

In this project, you will write a concurrent Web proxy that logs requests. A Web proxy is a program that acts as a middleman between a Web browser and an *end server*. Instead of contacting the end server directly to get a Web page, the browser contacts the proxy, which forwards the request on to the end server. When the end server replies to the proxy, the proxy sends the reply on to the browser.

Proxies are used for many purposes. Sometimes proxies are used in firewalls, such that the proxy is the only way for a browser inside the firewall to contact an end server outside. The proxy may do translation on the page, for instance, to make it viewable on a Web-enabled cell phone. Proxies are also used as *anonymizers*. By stripping a request of all identifying information, a proxy can make the browser anonymous to the end server. Proxies can even be used to cache Web objects, by storing a copy of, say, an image when a request for it is first made, and then serving that image in response to future requests rather than going to the end server.

In the first part of the project, you will write a simple sequential proxy that repeatedly waits for a request, forwards the request to the end server, and returns the result back to the browser, keeping a log of such requests in a disk file. This part will help you understand the basics about network programming and the HTTP protocol.

In the second part of the project, you will upgrade your proxy so that it uses threads to deal with multiple clients concurrently. This part will give you some experience with concurrency and synchronization, which are crucial computer systems concepts.

Part I: Implementing a Sequential Web Proxy

In this part, you will implement a sequential logging web proxy. Your proxy must open a socket and listen for a connection request. When your proxy receives a connection request, it must accept the connection, read the HTTP request, and parse the request to determine the name of the end server. Your proxy must then open a connection to that end server, forward the request to it, receive the reply, and forward the reply to the browser.

Your proxy must be able to support both HTTP/1.0 and HTTP/1.1 requests. However, your proxy does not need to support persistent connections.

Your proxy need only support the HTTP GET method. It is not required to support HEAD, POST, or any other HTTP method, aside from GET.

Since your proxy is a middleman between client and end server, it will have elements of both. It will act as a server to the web browser, and as a client to the end server. Thus you will get experience with both client and server programming.

Logging

Your proxy must keep track of all completed transactions in a log file named `proxy.log`. Each entry in this log file must be a line of the form:

```
Date: browserIP URL size
```

where `browserIP` is the IP address of the client, `URL` is the requested URL, `size` is the total size in bytes of the response that was returned by the end server, including the first line, headers, and body. For instance:

```
Tue 05 Apr 2016 22:23:20 CDT: 10.87.76.143 http://www.rice.edu/ 16316
```

In effect, since your proxy does not support persistent connections, `size` is the total number of bytes that your proxy receives from the end server between the opening and closing of the connection.

Only requests that are met by a response from an end server must be logged.

Since opening and closing files are costly operations, your proxy must only open the log file *once*, during initialization. In other words, your proxy may not open and close the log file each time it writes a log entry.

We have provided the function `create_log_entry` in `proxy.c` to create a log entry in the required format. Note that `create_log_entry` does not put a trailing newline on the returned string. Also, note that you are responsible for freeing the memory that stores the returned string.

Error Handling

In most cases, when your proxy is unable to complete a transaction, it must itself create an HTTP response containing an HTML error page and write this response to the client. This error page must describe why the transaction failed, for example, because the request is malformed or the end server does not exist. The exception to this mandate is when your proxy has already begun forwarding an HTTP response from the end server to the client. Once the end server's response is partially written to the client, your proxy can not send an error page.

We have provided the function `client_error` in `proxy.c` to write an HTTP response containing an HTML error page to the client. There are standardized meanings for a small set of the possible `err_num` and `short_msg` parameter values, for example, 400 and "Bad Request" are appropriate when the request is malformed[1]. In contrast, there are no standardized values for the `cause` and `long_msg` parameters. Use them as you see fit.

Port Numbers

Your proxy must listen for its connection requests on the port number passed in on the command line:

```
unix> ./proxy 18181
```

You may use any port number p, where $18000 \leq p \leq 18200$, and where p is not currently being used by any other user services (including other students' proxies).

Note that the port number restrictions are imposed by CLEAR. In general, you would be able to run a proxy on a wider range of ports.

[1] See RFC 2616 for a complete list.

Part II: Dealing with multiple requests concurrently

Real proxies do not process requests sequentially. They deal with multiple requests concurrently. Once you have a working sequential logging proxy, you will alter it to handle multiple requests concurrently. Specifically, in this part, you will implement the *prethreading* approach to handling each new connection request (CS:APP 12.5.5).

To implement the prethreading approach, you must buffer the new connection requests as they come in. To implement this buffering, you may adapt the `sbuf` that is discussed in the notes and textbook. However, to synchronize access to your buffer, you must use Pthread mutex and condition variables, which are discussed in Lab 12. You may not use semaphores.

Note that, with prethreading, it is possible for multiple peer threads to write to the log file concurrently. Thus, you will need to ensure that the log file entries are written atomically. Otherwise, the log file might become corrupted.

Provided Files

The provided files are all available in

```
/clear/www/htdocs/comp321/assignments/proxy/
```

To begin working on the project, do the following:

- Copy all of the files in this directory to the directory in which you plan to do your work.
- Type your team member names and Rice Net IDs in the header comment at the top of `proxy.c`.

The `proxy.c` file contains a few support routines to help you parse URIs (`parse_uri`), create log file entries (`create_log_entry`), and write an HTTP response containing an HTML error page to the client (`client_error`).

The provided `Makefile` is set up to compile the `csapp.c` file along with your `proxy.c` file and to link in all of the appropriate libraries.

An executable for our reference solution (`proxyref`) is also provided. However, this proxy outputs to a log file named `proxyref.log` so that it does not write to your proxy's log file. Your final proxy should behave just like the reference proxy, except for its debugging output. You are not required to match the debugging output of the reference proxy (which is also provided to help you develop your own proxy).

Notes

- A good way to get going on your proxy is to start with the basic echo server (CS:APP 11.4.9) and then gradually add functionality that turns the server into a proxy.
- Initially, you should debug your proxy using telnet as the client (CS:APP 11.5.3).
- Later, test your proxy with a real browser. Explore the browser settings until you find "proxies", then enter the host and port where you're running yours. You should only set the HTTP proxy, as that is all your code is going to be able to handle.
- When you test your proxy with a real browser, you should start with very simple web pages. We have provided a simple, text-only web page at:

`http://www.cs.rice.edu/~alc/proxytest.html`

Once you get that working, we suggest you try the textbook web page and the links therein:

`http://csapp.cs.cmu.edu`

Then you can try more complicated sites with multiple files and images, such as `www.rice.edu`. If you have trouble with a particular site, you may always use `proxyref` to see what features may be required to access that site.

- You will find it very useful to assign each thread a small unique integer ID (such as the current request number) and then pass this ID as one of the arguments to the thread routine. If you display this ID in each of your debugging output statements, then you can accurately track the activity of each thread.
- Be very careful about calling thread-unsafe functions inside a thread.
- Since the log file is being written to by multiple threads, in order for the output from different threads to not be scrambled, all of the output from a single thread needs to be printed out atomically. Fortunately, the POSIX standard requires that individual stream operations, such as fprintf(3) and fwrite(3), be atomic.
- Be careful about memory leaks. When the processing for an HTTP request fails for any reason, the thread must close all open socket descriptors and free all memory resources.
- Use the RIO (Robust I/O) package (CS:APP 10.4) for all I/O on sockets. Do not use standard I/O on sockets. You will quickly run into problems if you do. However, standard I/O calls, such as `fopen` and `fwrite`, are fine for I/O on the log file.
- The `Rio_readn`, `Rio_readlineb`, and `Rio_writen` error checking wrappers in `csapp.c` are not appropriate for a proxy because they terminate the process when they encounter an error.
- The HTTP/1.1 specification does *not* place an upper limit on the length of a URI. Moreover, in testing your proxy at web sites with rich content, you will sooner or later encounter a URI that is longer than `csapp.h`'s defined `MAXLINE`. In other words, you will sooner or later encounter a start or header line in an HTTP request message that will not fit in a `char` array of size `MAXLINE`. Nonetheless, to process the line, *e.g.*, to perform `parse_uri`, your proxy will need to store the entire URI, if not the entire line, in a `char` array. You should explore how `rio_readlineb` behaves when the length of the line being read exceeds the given buffer size.
- Reads and writes can fail for a variety of reasons. The most common read failure is an `errno == ECONNRESET` error caused by reading from a connection that has already been closed by the peer on the other end, typically an overloaded end server. The most common write failure is an `errno == EPIPE` error caused by writing to a connection that has been closed by its peer on the other end. This can occur, for example, when a user hits their browser's "Stop" button during a long transfer.
- Writing to a connection that has been closed by the peer elicits an error with `errno` set to `EPIPE` the first time. Writing to such a connection a second time elicits a `SIGPIPE` signal whose default action is to terminate the process. One relatively easy way to keep your proxy from crashing is to use the `SIG_IGN` argument to the Signal function (CS:APP 8.5.3) to explicitly ignore these `SIGPIPE` signals (or you can catch them and print an appropriate warning message).

- Modern web browsers and servers support persistent connections, which allow back-to-back requests to reuse the same connection. Your proxy will not do so. However, your browser is likely to set the headers `Connection`, `Keep-Alive`, and/or `Proxy-Connection` to indicate that it would like to use persistent connections. If you pass these headers on to the end server, it will assume that you can support them. If you do not support persistent connections, then subsequent requests on that connection will fail, so some or all of the web page will not load in your browser. Therefore, you should strip the `Connection`, `Keep-Alive`, and `Proxy-Connection` headers out of all requests, if they are present. Futhermore, HTTP/1.1 requires a `Connection: close` header be sent if you want the connection to close. Note that you must leave the other headers intact as many browsers and servers make use of them and will not work correctly without them.

Evaluation

To turnin your code, you should use the `turnin` process on CLEAR (`https://docs.rice.edu/confluence/x/qAiUAQ`). First, create a `proxy` directory within your `comp321` directory. Then, be sure that your code is entirely contained in a file named `proxy.c` inside this `proxy` directory. (Note that we will compile only your `proxy.c` file along with the provided `csapp.c`.) Finally, submit your solution by running the following command from inside your `comp321` directory:

```
UNIX% turnin comp321-S17:proxy
```

After which, you should receive an e-mail confirming the submission of your solution.

Instead of a writeup, each individual/group will be evaluated on the basis of a demonstration to either the instructor or a TA. During the demo, you will demonstrate that your proxy behaves correctly and you will answer basic questions about your proxy. These demos will start during the last day of classes and will be scheduled near the end of the term.

The project will be graded as follows:

- *Basic proxy functionality (50 points).*

  Your proxy should correctly accept connections, forward the requests to the end server, and pass the response back to the browser, making a log entry for each request. As a starting point, your program should be able to proxy browser requests to the following Web sites and correctly log the requests:

  - `http://www.cs.rice.edu/~alc/proxytest.html`
  - `http://csapp.cs.cmu.edu`
  - `http://www.rice.edu`
  - `http://www.nfl.com`
  - `http://www.cnn.com`

  Additional web sites will be tested at the demo.

- *Handling concurrent requests (20 points).*

  Your proxy should be able to handle multiple concurrent connections. The following test is one way to determine this: (1) Open a connection to your proxy using `telnet`, and then leave it open without typing in any data. (2) Use a Web browser (pointed at your proxy) to request content from some end server.

- *Style (15 points).*

  This includes general program style, the thoroughness of your comments, and whether you check the return code from all system calls. Furthermore, your code should not have any memory leaks, and your proxy should be thread-safe, protecting all updates of the log file and protecting calls to any thread unsafe functions.

- *Demonstration (15 points).*

  At the demonstration you will be asked basic questions about your proxy that would normally have been in your writeup on previous projects (i.e. descriptions of your design and testing strategy). This is your opportunity to explain any interesting design features and show off your proxy.