CS 105, Fall 2014
Web Proxy

Introduction

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

Proxies are 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. Finally, some evil proxies snoop on people's activities or modify Web pages, for example by inserting advertising.

In this lab, you will write a concurrent Web proxy that simply logs all the requests it sees. In the first part of the lab, you will write a simple sequential proxy that repeatedly waits for a request, forwards it to the end server, and returns the result back to the browser, keeping a log of the requests in a disk file. This part will help you understand basics about network programming and the HTTP protocol.

In the second part of the lab, 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.

Logistics

As always, you must work with your partner. Handin will be electronic, using `cs105submit`. You will also demonstrate your proxy to the professor or graders on the last day of lab.

Handout

The handout is distributed in a `tar` file named `proxylab-handout.tar`, which you will find linked from the lab Web page. Start by copying `proxylab-handout.tar` to a (protected) directory in which you plan to do your work. Then give the following command:

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

This will cause a number of files to be unpacked in the directory:

- `proxy.c`: This is the only file you will be modifying and handing in. It contains the bulk of the logic for your proxy.
- `csapp.c`: This is (almost) the file of the same name that is described in the textbook. It contains error handling wrappers and helper functions such as the RIO (Robust I/O) package (CS:APP 11.4), `open_clientfd` (CS:APP 12.4.4), and `open_listenfd` (CS:APP 12.4.7). It's "almost" because we've fixed a few small problems with the version in the textbook.
- `csapp.h`: This file contains a few manifest constants, type definitions, and prototypes for the functions in `csapp.c`.
- `strmanip.c`: This file contains a function, `substitute_re` (see below) that makes it easier to do string substitution in the C language.
- `strmanip.h`: This file defines the prototype for `substitute_re`.
- `Makefile`: Compiles and links `proxy.c`, `strmanip.c`, and `csapp.c` into the executable `proxy`.

Your `proxy.c` file may call any function in the `csapp.c` and `strmanip.c` files. However, since you are only handing in a single `proxy.c` file, please don't modify `csapp.c` or `strmanip.c`. If you want different versions of any of the functions in these files (see the Hints section), write new (renamed) ones in the `proxy.c` file.

Part I: Implementing a Sequential Web Proxy

In this part you will implement a sequential logging proxy. Your proxy should open a socket and `listen` for a connection request. When it receives a connection, it should `accept` it, read the HTTP request, and parse it to determine the name of the end server. It should then open a connection to the end server, send it the request, receive the reply, and forward the reply to the browser if the request is not blocked.

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 should keep track of all requests in a log file named `proxy.log`. Each log file entry should be a line of the form:

```
Date: browserIP URL size
```

where `browserIP` is the IP address of the browser, `URL` is the URL asked for, `size` is the size in bytes of the object that was returned. For instance:

```
Tue 16 Apr 2013 18:51:02 PDT: 134.173.42.2 http://www.cs.hmc.edu/ 8674
```

Only requests that are met by a response from an end server should be logged. We have provided the function `format_log_entry` in `csapp.c` to create a log entry in the required format.

Port Numbers

You proxy should listen for its connection requests on the port number passed in on the command line:

```
unix> ./proxy 15213
```

You may use any port number p, where $1024 \leq p \leq 65536$, and where p is not currently being used by any other system or user services (including other students' proxies). See `/etc/services` for a list of the port numbers reserved by other system services. We suggest that you use one of your team's login ID numbers (see the `id` command) to avoid collisions with other students.

HTTP

Your Web proxy will listen for HTTP requests of the form:

```
GET http://www.cs.hmc.edu HTTP/1.0
```

(the version can also be HTTP/1.1). More information can be found in Section 11.5.3 of the textbook.

IMPORTANT NOTE: Although you must accept both HTTP/1.0 and HTTP/1.1, you should convert the request to HTTP/1.0 when you forward it to the server. The reason is that in HTTP/1.1 it is difficult to detect the end of the server's response. For our purposes, it's easier to change a ".1" to a ".0" than to write code to properly handle HTTP/1.1.

A Threading Note

The supplied code contains a skeleton function for handling proxy requests. Because you'll be adding threading later, the skeleton is written on the assumption that it will run as a thread. Thus, you might find it easiest to call it as a thread (although it's not absolutely necessary to do so).

A Testing Note

Most modern Web pages are complicated, and modern Web browsers make many requests at high speed. The result is that if you point a browser at your proxy and try to fetch even a simple page like Google, you'll quickly be overwhelmed.

Instead, we suggest that you do your initial testing with `telnet`, as discussed below. After you have your proxy working, you can switch to a browser.

Part II: Dealing with multiple requests concurrently

Real proxies do not process requests sequentially, because it's not good for one client to have to wait for another, slower one. Instead, they handle multiple requests concurrently. Once you have a working sequential logging proxy, you should alter it to handle multiple requests concurrently. The simplest approach is to create a new thread to deal with each new connection request that arrives (see Section 12.3.8 of the textbook for an example).

With this approach, it is possible for multiple peer threads to write to the log file concurrently. Thus, you will need to use a semaphore or a pthreads mutex to synchronize access to the file such that only one peer thread can modify it at a time. If you do not synchronize the threads, the log file might be corrupted, for example by having one line in the file begin in the middle of another.

Note that the skeleton code we provided *does not* contain any thread synchronization. It is your responsibility to identify shared variables and protect them appropriately. Remember to watch out for thread-unsafe functions!

Evaluation

Each team will be evaluated on the basis of a demo to the graders and professors. The demos will take place in lab, the day it is due. Both team members must be present for the demo unless other arrangements are made in advance.

Grading is as follows:

- Basic proxy functionality (30 points). Your sequential proxy should correctly accept connections, forward requests to the end server, and pass the response back to the browser, making a log entry for each request. Your program should be able to proxy browser requests to (at least) the following Web sites and correctly log the requests:
  - `http://www.cs.hmc.edu/~geoff`
  - `http://www.cs.hmc.edu`
  - `http://www.cs.pomona.edu`

  You should also test with fancier Web sites such as Yahoo, CNN, etc.

- Handling concurrent requests (20 points). Your proxy should be able to handle multiple concurrent connections. We will determine this using the following test: (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.

  Furthermore, your proxy should be thread-safe, protecting all updates of the log file and protecting calls to any thread-unsafe functions such as `gethostbyaddr`. We will determine this by inspecting the code during the demo.

- Style/correctness (10 points). Up to 10 points will be awarded for code that is readable and well commented. Your code should begin with a comment block that describes in a general way how your proxy works. Furthermore, each function should have a comment block describing what it does. Your threads should run detached (see `man pthread_detach`

and Section 12.3.6 of the textbook), and your code should not have any memory leaks. We will determine this by inspection during the demo.

- Extra Credit (10 points). We will award up to ten points of extra credit for additional features in your proxy. The exact feature is up to you, but it should somehow modify the interaction with the Web server. For example, you might choose to insert or modify material on every Web page, or perhaps only on some pages. Be creative and evil!

Hints

- The best way to get going on your proxy is to start with the basic echo server (see the textbook, Section 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 (see the textbook, Section 11.5.3). Note that you may have to hit Enter twice before your proxy will respond.

- An odd quirk of the HTTP protocol is that a request must be terminated by a blank line. If your proxy seems to hang after contacting the server, without getting a response, make sure that it is sending TWO newlines at the end of the request.

- Later, test your proxy with a real browser. Explore the browser settings until you find "proxies", then enter an IP address of 134.173.42.167 (Wilkes) and the port you are using for your proxy. **Don't forget to undo the proxy setting after you are done testing, or your browser will stop working!**

  With Firefox, choose Edit/Preferences, then Advanced. Click the Network tab and then, under "Connection", click Settings and choose manual proxy configuration.

  In Safari, choose Safari/Preferences, then Advanced, and click "Change Settings" for Proxies. Check the box for "Web Proxy (HTTP)" and fill in the IP address in the big box and the port in the small one. (Note: you can't do this on the lab Macs, but it will work on your personal machine.)

  In Internet Explorer 8, choose Tools/Internet Options, the Connections, and click LAN Settings. Check 'Proxy server,' and click Advanced. Just set your HTTP proxy, because that's all your code is going to be able to handle.

  In Chrome, click on the customize button (the three horizontal lines at the top right of the window) and select Settings. Click "Show advanced settings" and then, under "Network", "Change Proxy Settings". From that point you're on your own because our version of Chrome decided to be snippy about the computer we were using. Gee, thanks, Google. Everybody else manages to make it work.

  Note: Because of the CS department's firewall, you won't be able to access your proxy except in the labs. If you're working from your room or from Pomona, you can work around that problem by running Firefox directly on Wilkes.

- Since we want you to focus on network programming issues for this lab, we have provided you with two additional helper routines: `parse_uri`, which extracts the hostname, path, and port components from a URI, and `format_log_entry`, which constructs an entry for the log file in the proper format.

- Be careful about 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 before terminating.

- You will find it 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, you can accurately track the activity of each thread.

- To avoid a potentially fatal memory leak, your threads should run as detached, not joinable (see the textbook, Section 12.3.6).

- Since the log file is being written to by multiple threads, you must protect it with mutual exclusion semaphores whenever you write to it (see the textbook, Sections 12.5.2 and 12.5.3). You can also use pthread mutexes if you prefer, as in the ring buffer lab.

- Be careful about calling thread-unsafe functions such as `inet_ntoa`, `gethostbyname`, and `gethostbyaddr` inside a thread. In particular, the `open_clientfd` function in `csapp.c` is thread-unsafe because it calls `gethostbyname`, a Class-3 thread unsafe function (see the textbook, Section 12.7.1). You will need to write a thread-safe version of `open_clientfd`, called `open_clientfd_ts`, that uses the lock-and-copy technique (Section 12.7.1) when it calls `gethostbyname`.

- Use the RIO (Robust I/O) package (Section 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`, `fwrite`, and `fprintf` are fine for I/O on the log file.

- Even though HTTP appears to be a text protocol, some of the data passed via an HTTP connection is actually binary. This means, in particular, that you can't use `strlen` to find out the length of data that you get from an HTTP server. Instead, you must take advantage of the fact that the functions like `rio_readn` return the number of bytes they transferred. Use those return values, not `strlen`!

- The `Rio_readn`, `Rio_readlineb`, and `Rio_writen` error checking wrappers in `csapp.c` are not appropriate for a realistic proxy because they terminate the process when they encounter an error. Instead, you should use new wrappers called `Rio_readn_w`, `Rio_readlineb_w`, and `Rio_writen_w` that simply return after printing a warning message when I/O fails. When either of the read wrappers detects an error, it should return 0, as though it encountered EOF on the socket. We have provided some of these wrappers for you.

- 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 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.

- The first time you write to a connection that has been closed by the peer, you will get an error with `errno` set to `EPIPE`. Writing to such a connection a second time elicits a `SIGPIPE` signal, whose default action is to terminate the process. To keep your proxy from crashing you can use the `SIG_IGN` argument to the `signal` function (see the textbook, Section 8.5.3) to explicitly ignore these `SIGPIPE` signals.

- To make it easier for you to rewrite Web pages in an appropriately evil fashion, we have provided a function named `substitute_re`, which you can use to perform regular-expression substitution on an arbitrary string. This function is described in detail in the header file `strmanip.h`. The regular expressions accepted are in the form of egrep(1) (see "`man 1 egrep`" for more information).

- When you are rewriting Web pages, be sure to use regular expressions that are unlikely to match random information in binary data, because you will wind up damaging embedded images. We suggest that you limit yourself to rewriting strings of five characters or more.

- Some Web sites have complex structures that make rewriting difficult. If you have trouble, use your browser's "View Source" feature to ensure that you're working correctly with the raw HTML.

- If your proxy corrupted a Web page and fixing the code doesn't seem to help, remember to clear your browser cache. Sometimes after a browser has fetched a corrupted page, it will stay in the cache and mislead you into thinking your code is still broken.

- If your browser is running security extensions like HTTPS Everywhere (Firefox), you may have to disable them. Your proxy won't intercept HTTPS requests.

Handin/Demo Instructions

Before your demo, use `cs105submit` to submit your code. Make sure you:

- Remove any extraneous print statements.

- Turn off debugging.

- Include your names in comments in `proxy.c`.