CS4400, Fall 2015
Lab Assignment 6: Web Proxy
Assigned: November 24, Due: December 7, 11:59PM

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.

In this lab, you will write a concurrent Web proxy that logs requests. In the first part of the lab, 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 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.

Hand Out Instructions

Retrieve `proxylab-handout.tar` from Canvas. Start by copying `proxylab-handout.tar` to a (protected) directory in which you plan to do your work. Then give the command "`tar xvf proxylab-handout.t`
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 the file of the same name that is described in the CS:APP textbook. It contains error handling wrappers and helper functions such as the RIO (Robust I/O) package (CS:APP 10.5), `open_clientfd` and `open_listenfd` (CS:APP 11.4.8).

- `csapp.h`: This file contains a few manifest constants, type definitions, and prototypes for the functions in `csapp.c`.
- `Makefile`: Compiles and links `proxy.c` and `csapp.c` into the executable `proxy`.

Your `proxy.c` file may call any function in the `csapp.c` file. However, since you are only handing in a single `proxy.c` file, please don't modify the `csapp.c` file. If you want different versions of functions in in `csapp.c` (see the Hints section), write new functions 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 request, it should accept the connection, 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 file 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:

```
Sun 27 Oct 2002 02:51:02 EST: 128.2.111.38 http://www.cs.cmu.edu/ 34314
```

Note that `size` is essentially the number of bytes received from the end server, from the time the connection is opened to the time it is closed. 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

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

```
unix> ./proxy 2115
```

Technically, 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. However, the CADE lab currently reserves only ports 2112-2120 for this sort of development.

Part II: Dealing with multiple requests concurrently (extra credit)

Real proxies do not process requests sequentially. They deal with 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 (CSAPP 12.3.8).

With this approach, it is possible for multiple peer threads to access the log file concurrently. Thus, you will need to use a semaphore 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 instance, one line in the file might begin in the middle of another.

Evaluation

We will be testing some standard things in evaluation.

- Basic proxy functionality (30 points). Your sequential 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. Your program should be able to proxy browser requests to the following Web sites and correctly log the requests:
  - `http://www.utah.edu`
  - `http://www.aol.com`
  - `http://www.nfl.com`

  So for example, we should be able to test your sequential proxy by running

  ```
  ./proxy <port>
  ```

  and the following commands on a remote machine:

  ```
  telnet <proxy machine url> <port>
  GET http://www.utah.edu/ HTTP/1.0
  ```

  This should behave similarly to the example in Figure 11.24 in the textbook.

- Handling concurrent requests (10 points Extra Credit).

  Your proxy should be able to handle multiple concurrent connections. We will determine this by running simultaneous `telnet` connections, but it is also possible to test by pointing a Web browser at your proxy using the proxy settings previously described.

  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 inspection of the code.

- Style (5 points). Up to 5 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 that function does. Furthermore, your threads should run detached, and your code should not have any memory leaks. We will determine this by inspection.

Hints

- The best 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. The code you will need to add has been described with TODO comments in the code from Canvas.
- You should debug your proxy using telnet as the client (CS:APP 11.5.3).
- Although not required for this assignment, it is actually possible to 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. (On Safari and Chrome, these are advanced settings that require admin priviledges.) Just set your HTTP proxy, because that's all your code is going to be able to handle.
- Two helper routines are provided: `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 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 before terminating.
- For the parallel version, 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.
- To avoid a potentially fatal memory leak, your threads should run as detached, not joinable (CS:APP 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 (CS:APP 12.5.2 and 12.5.3).
- Be very 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 `gethostbyaddr`, a Class-3 thread unsafe function (CSAPP 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 (CS:APP 12.7.1) when it calls `gethostbyaddr`.
- Use the RIO (Robust I/O) package (CS:APP 10.5) 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 realistic proxy because they terminate the process when they encounter an error. Instead, you should write 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.

- 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 connection that has been closed by the peer first time elicits 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 (CS:APP 8.5.3) to explicitly ignore these SIGPIPE signals

Turn in proxy.c in Canvas.