15-213, Fall 2002
Lab Assignment L7: Logging Web Proxy
Assigned: November 21, Due: December 5, 11:59PM

Rajesh is in charge of this lab. Please email him `rajesh@cs.cmu.edu` with any questions or concerns.

Introduction

A web proxy is a program which acts as a middleman between a web server and browser. Instead of contacting the server directly to get a web page, the browser contacts the proxy, which forwards the request on to the server. When the 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 a server outside. The proxy may do translation on the page, for instance, to make it viewable on a web-enabled cell phone. Proxies are used as “anonymizers” – by stripping a request of all identifying information, a proxy can make the browser anonymous to the 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 server. Squid (available at `http://squid.nlanr.net`) is a free proxy cache.

In this lab, you will write a simple web proxy that logs and filters requests. In the first part of the lab, you will set up the proxy to accept requests, check if the request should be blocked, forward the requests to the server if not, and return the result back to the browser, keeping a log of such requests in a disk file. In this part, you will learn how to write programs that interact with each other over a network (socket programming), as well as some basic HTTP.

In the second part of the lab, you will upgrade your proxy to deal with multiple open connections at once. You may implement this in two ways: Your proxy can spawn a separate thread to deal with each request, or it may multiplex between the requests using the `select(2)` Unix system call. Either option will give you an introduction to dealing with concurrency, a crucial systems concept. Using threads for this lab is recommended.

Logistics

This is an individual assignment. All files you need are in the directory

```
/afs/cs.cmu.edu/academic/class/15213-f02/L7
```

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.tar`. This will cause a number of files to be unpacked in the directory. The files you will be modifying and turning in are proxy.c, csapp.c, and csapp.h.

The proxy.c file contains the bulk of the logic for your proxy. The csapp.c and csapp.h files are described in your textbook. The csapp.c file contains error handling wrappers and helper functions such as the RIO functions (Section 11.4), the open_clientfd function (Section 12.4.4), and the open_listenfd function (Section 12.4.7). The csapp.h file contains the prototypes for the functions in csapp.c.

Part I: Implementing a web proxy

The first step is implementing a basic logging proxy. When started, your proxy should open a socket and listen for connections. When it gets a connection (from a browser), it should accept the connection, read the request, check if the address is blocked, and parse it to determine the server that the request was meant for. It should then open a socket connection to that server, send it the request, receive the reply, and forward the reply to the browser if the request is not blocked.

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

Filtering

The blocked URLs are stored in the file called proxy.filter. The web proxy may read in the addresses at the initialization. When a request on a blocked address comes from a web browser, the proxy should return a *permission denied* information to the browser in the following format:

```
<html><head><title>Proxy error</title></head>
<body>You are not allowed to access this web page.</body></html>
```

Logging

Your proxy should also keep track of all requests in a log file named proxy.log. Each line should be 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 server, from the time the connection is opened to the time it is closed. Only requests that are met by a response from a server (or cached response) should be logged. We have provided the function void format_log_entry() to create a log entry in the required format.

Graphics Display

To see the number of connections currently being serviced by your proxy, we have provided an xlib graphics display. To see the display, you need to do the following things

- On the machine you are logging in from:
  type `xhost +<fishmachine>`
  e.g. xhost +bass.cmcl.cs.cmu.edu

- On the fish machine, set the display to the machine you are logging in from using
  `setenv DISPLAY <ip address>:0.0`
  e.g. setenv DISPLAY 128.2.64.32:0.0

  This step may not be necessary if your `telnet` or `ssh` session automatically sets the DISPLAY environment variable for you.

The code for initializing and destroying the display have already been provided in the main routine of `proxy.c`. You will need to call `change_display` with the current number of connections to change the display. The display also logs the number of connections currently being serviced to a file called `display.log`. While building the concurrent server, take care to ensure that all calls to `change_display` are properly serialized.

Port Numbers

Since there will be many people working on the same machine, all of you can not use the same port to run your proxies. You are allowed to select any non-privileged port for your proxy, as long as it is not taken by other system processes. Selecting a port in the upper thousands is suggested (i.e., 3070 or 8104).

Part II: Dealing with multiple requests

Real proxies do not process requests sequentially. They deal with multiple requests in parallel. This is particularly important when handling a request can involve a lot of waiting (as it can when you are, for instance, contacting a remote web server). While your proxy is waiting for a remote server to respond to a request so that it can serve one browser, it could be working on a pending request from another browser.

Thus, once you have a logging proxy, you should alter it to handle multiple requests simultaneously. There are two basic approaches to doing this:

Threads A common way of dealing with concurrent requests is for a server to spawn a thread to deal with each request that comes in. In this architecture, the main server thread simply accepts connections and spawns off worker threads that actually deal with the requests (and terminate when they are done).

If you choose this method, however, you will have the problem that multiple peer threads will be trying to access the log file at once. If they do not somehow synchronize with each other, the log file will be corrupted (for instance, one line in the file might begin in the middle of another). You will need to use a semaphore to control access to the file, so that only one peer thread can modify it at a time.

select. Another way to deal with concurrent requests is to multiplex between connections by hand using the select system call. As described in your text, the select function takes a set of file descriptors and waits until one of them is ready for reading or writing. You can use this call to simultaneously wait on all open socket connections, and process whichever one is ready first. For instance, your proxy might be waiting for a request to come from a client on one socket, for a response to come from a server on another socket, and at the same time listening for new connection requests on its listening socket. Your code would use select to wait for all of these things at once, handling whichever happened first.

With this architecture, you will not need to deal with synchronization among concurrent processes, since there is only one process running, but you will need to correctly multiplex on several connections, without blocking on any of them.

Evaluation

- Logging Proxy (30 points). Half credit will be given for a program that accepts connections, forwards the requests to a server, and sends the reply back to the browser, blocking requests for filtered URLs, and making a log entry for each request.
- Handling concurrent requests (20 points). Most of the rest of the credit will require handling multiple concurrent connections with threads. We will test to make sure that one slow web server does not hold up other requests from completing, and that your log file does not get corrupted by multiple competing processes.
- Style (10 points). Up to 10 points will be awarded for code that is readable and well commented. Define macros or subroutines where necessary to make the code more understandable.

Checking your work

We have provided a driver program called driver.pl to help you check your work. We will use this driver program to help determine the correctness of your proxy.

The driver.pl program starts the proxy as a child process, sends it requests, checks replies from the proxy and display a sample score for this lab. The sample score would give you an idea of how you are going to be graded.

```
unix> ./driver.pl
usage: ./driver.pl [port-number] [proxy-log] [lab-part]
```

port-number is the port number that your proxy will be listening to accept requests and proxy-log is the log file output by your proxy. lab-part is the the part of the lab to test and it can be part1, part2, all. For example,

```
unix> ./driver.pl 4502 proxy.log part1
```

will start to test Part I of the lab at port 4502, assuming a log file named proxy.log.

Resources and Hints

- Read Chapter 10 - 12 in your textbook. They contain useful information on system-level I/O, network programming, HTTP protocols, and concurrent programming.
- Use the RIO (Robust I/O) package described in your textbook for all I/O on sockets. Do not use standard I/O on sockets. You will quickly run into problems if you do.
- You can start with the Tiny Web server described in your textbook. This is a fully functional Web server. You may copy any of the Tiny code and use it in your proxy.
- Initially, use the `telnet` program to debug your proxy, as described in your text.
- 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. With Netscape, choose Edit, then Preferences, then Advanced, then Proxies, then Manual Proxy Configuration. In Internet Explorer, choose Tools, then Options, then Connections, then LAN Settings. Check 'Use proxy server,' and click Advanced. Just set your HTTP proxy, because that's all your code is going to be able to handle.
- In certain cases, a client closing a connection prematurely results in the kernel delivering a SIGPIPE signal to the proxy. This is particularly the case with Internet Explorer. To prevent your proxy from crashing, you should ignore this signal by adding the following line early in your code:

  ```
  signal(SIGPIPE, SIG_IGN);
  ```

  Also, in certain cases, writing to a socket whose connection has been closed prematurely results in the `write` system call returning a -1 and setting `errno` to EPIPE. Your proxy should not terminate when a `write` elicits an EPIPE error. Simply close the socket, update the display to reflect one less connection, and continue.

  The easiest way to handle EPIPE errors is to modify the `Rio_writen` wrapper in `csapp.c` to test specifically for EPIPE errors and return some special value (say 0) when it encounters them.
- To test how your proxy handles requests under high concurrency, try accessing the following web sites from your proxy with Internet Explorer:

  ```
  http://www.nfl.com/
  http://www.cnn.com/
  http://www.weather.com/
  ```

  The IE would set up tens of concurrent connections with your proxy.
- **IMPORTANT:** If you use threads to handle connection requests, you must run them as *detached*, not *joinable*, to avoid memory leaks that will likely crash the FISH machines. To run a thread detached, add the line `Pthread_detach(thread_id)` in the parent after calling `Pthread_create()`.

Handin

- Remove any extraneous print statements.
- Make sure that you have included your identifying information in `proxy.c`.
- Create a team name of the form:
  - "ID" where ID is your andrew ID.
- To hand in your `proxy.c` file, type:

  ```
  make handin TEAM=teamname
  ```

  where `teamname` is the team name described above.
- After the handin, you can submit a revised copy by typing

  ```
  make handin TEAM=teamname VERSION=2
  ```

  You can verify your handin by looking at

  ```
  /afs/cs.cmu.edu/academic/class/15213-f02/L7/handin
  ```