CS 213, Fall 2000
Lab Assignment L5: Logging Web Proxy
Assigned: Nov. 28, Due: Mon. Dec. 11, 11:59PM

Jason Crawford (`jasonc@cs.cmu.edu`) is the lead person for this assignment.

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. They proxy may do translation on the page, for instance, to make it viewable on a web-enabled cell phone. Proxies are used as "anonomizers"—by stripping a request of all identifying information, a proxy can make the browser anonymous to the server (for example, `www.silentbrowser.com`). 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.[1]

In this lab, you will write a simple web proxy that logs requests. In the first part of the lab, you will set up the proxy to accept requests, parse the HTTP, forward the requests to the server, 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.

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, 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.

[1] Jeff Sarnat notes that proxies can also be used to circumvent software that blocks porn sites.

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.

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, and `size` is the size in bytes of the object that was returned. For instance:

```
Fri 17 Nov 2000 02:51:02 EST: 128.2.111.38 http://www.cs.cmu.edu/ 34314
```

To do this part, you will need to understand socket programming and basic HTTP. See the Resources section below for help on these topics.

Part II: Dealing with multiple requests

Real servers do not process one request at a time, 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 peer 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 or mutex to control access to the file, so that only one peer thread can modify it at a time.

select(2) Another way to deal with concurrent requests is to multiplex between connections by hand using the `select(2)` system call. The select call 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 connections on its server socket. Your code would use `select(2)` 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.

Again, see the Resources section for further information on these topics.

Logistics

The tar file for this assignment can be retrieved from
`/afs/cs.cmu.edu/academic/class/15213-f00/L5/L5.tar`

As usual, you may work in a group of up to 2 people.

Resources and Hints

- Since HTTP is just plain text, you can actually try out both the client and server halves of your proxy by hand. You can use `telnet` to simulate a web browser: Just telnet to the host and port where you are running the proxy, and type your request by hand (like `GET http://www.yahoo.com/`).

  To experiment with the client side of your proxy, we are providing a "reverse telnet" utility (courtesy of Blake Scholl). This program listens on a given port for a connection, then accepts the connection and displays incoming text on the screen. Whatever you type on stdin is sent to the process that connected. So if you start `reverse_telnet` on port 8000 and point your web browser or proxy to that port, you will see HTTP requests on the screen and can actually type back a web page.

  The reverse telnet program is available in the course directory under `L5/reverse_telnet`.

- 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". Just set your HTTP proxy, 'cause that's all your code is going to be able to handle.

- For an example of a real, working web server in only 282 lines of source code, we are providing `tiny`, a minimal server written by Dave O'Hallaron. The program can be found in the L5 directory in `tiny.c`. Of course, you can (and should) test your proxy on real web sites, but `tiny` can give you a more controlled environment, as well as an excellent example of server-side socket programming and HTTP processing.

- Getting all the details right in socket programming is a pain. I recommend looking at examples (such at `tiny` and `reverse_telnet`). However, I *strongly* recommend *not* just cutting and pasting code—use the examples to figure out what's going on and what you're supposed to do.

  Here's a quick reference on how to set up a server socket to listen for connections (this is just a summary; you'll need to dig deeper in the example code and man pages for more details):

  1. Obtain a socket with the `socket(2)` system call.
  2. Helpful, but not necessary: set the `SO_REUSEADDR` option on the socket.
  3. Bind it to an address with `bind(2)`. This requires properly setting up a `sockaddr_in` structure. Set the `sin_family` to `AF_INET` (to get a regular network socket), the `sin_addr.s_addr` to `INADDR_ANY` (to have the system fill in the default IP address), and the `sin_port` to the desired port. Don't forget that this stuff is in network byte order!
  4. Set the socket up to listen for connections with `listen(2)`.

  When a connection is ready, you can accept with `accept(2)`.

  On the client side, the process of connecting to a server (given the hostname and port number) is a little bit easier:

1. Obtain a socket with `socket(2)`.
2. Lookup the host entry of the host with `gethostbyname(2)`.
3. Use `connect(2)` to actually connect to the server. Again, this requires filling out a `sockaddr_in` struct. Use `AF_INET` for the `sin_family`, and copy the `sin_addr` directly from the `h_addr` field of the `struct hostent` returned by `gethostbyname`. You can figure out how to set the port.

- As mentioned, you will need to know some basic HTTP for this assignment. You could, of course, read RFC 2616, the HTTP 1.1 spec, available from the Web Consortium at `www.w3.org`—but it happens to be about 100 pages long. So, here's all you really need to know:

  The only type of request your proxy will need to deal with is the `GET` request. Requests will be several lines of text (separated by CRLF—"`\r\n`" in C, not just "`\n`"), beginning with a line of the form: `GET` *`url version`*. The following lines will be headers; the end of the headers is signified by a blank line.

  The version is just something like "`HTTP/1.0`"; you will simply have to read this and echo it when you forward the request. You also need to parse the URL to determine the host, port, and pathname of the HTML object you want. (You can assume the URL is of the form "`http://`*`host`*`[:`*`port`*`]/`*`path`*". The default HTTP port is 80.) Then open a connection to the proper server (on the given host and port) and send the GET request, using *only the path* instead of the full URL. Be sure to use the correct version and to copy all the other header lines and send them too.

  So for instance, you might get:

```
GET http://www.cs.cmu.edu/csd/bscs/index.html HTTP/1.0
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.04 [en] (X11; U; SunOS 5.5.1 sun4m)
Host: www.cs.cmu.edu
Accept: image/gif, image/jpeg, image/pjpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

  You would send the following to `www.cs.cmu.edu` on port 80:

```
GET /csd/bscs/index.html HTTP/1.0
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.04 [en] (X11; U; SunOS 5.5.1 sun4m)
Host: www.cs.cmu.edu
Accept: image/gif, image/jpeg, image/pjpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

  Make sure you are sending an extra blank line after all the headers, to signify the end of the request.

  If you're curious for more details on all of the above, ask me personally (`jasonc@cs.cmu.edu`). The details are hairy.

- If you choose to use threads for Part II, handout #8, "Concurrent Programming with Threads", contains all the information you will need.

- **IMPORTANT:** If you use threads to handle connection requests, run must them as *detached*, not *joinable*, to avoid memory leaks that will likely crash the fish servers. See Handout #8 for details on detached threads.

- In general, use the man pages for documentation on system calls. Man pages should always be your first line of defense, but unfortunately they are not adequate in themselves. For excellent in-depth explanations of network programming, `select(2)`, and threaded servers, see W. Richard Stevens, "Unix Network Programming: Networking APIs (Second Edition), Prentice-Hall, 1998.

- A random hint: Remember that when calling `read(2)` on a socket, the read may return before all data has been received (i.e., you may get only part of the message). If you are expecting a certain number of bytes, or a certain "end-of-data" marker, you may need to do multiple reads to get all the data. (The `read(2)` call returns the number of bytes read, or 0 on end-of-file.)

Evaluation

- Logging web proxy (30 points). Half credit will be given just for getting a program which accepts connections, forwards the requests to a server, and sends the reply back to the browser, making a log entry for each request.

- Handling concurrent requests (30 points). The other half of the credit will require handling multiple concurrent connections, either with threads or with `select(2)`. 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.

Handin

See the web page for this lab for how to hand in your solution to the web server.

NOTE that this lab is due on the last day of class! There will be no late turnins and no extensions.