15-213, Spring 2002
Lab Assignment L7: Logging Web Proxy
Assigned: April 23, Due: May 3, 11:59PM

Please see one of us or email `staff-213@cs.cmu.edu` with any questions or concerns. As always, we're here to help.

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

You may work in a group of up to 2 people. You can retrieve the files needed for this lab from

`/afs/cs.cmu.edu/academic/class/15213-s02/labs/L7/L7.tar`.

Your code should only change `proxy.c` in this lab. Up to 2 late days can be used for this lab.

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.

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

Filtering

The blocked URLs are stored in `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:

```
Fri 13 Apr 2001 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.

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 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.

Evaluation

- Logging Proxy (15 points). A proxy that can filter and forward requests correctly will receive 10 points. Logging and error handling take the rest 5 points.
- Handling concurrent requests (10 points). 4 points are for correct filtering and forwarding and 6 points for concurrency handling.

Checking your work

We have provided some tools to help you check your work.

`driver.pl`

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 11 and 12 in your textbook. They contain useful information on network programming, HTTP protocols, and concurrent programming.
- 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/`). For example, suppose your proxy is listeing on port 2400. You can use the following command to test if the proxy gets the request:

  ```
  unix> telnet localhost 2400
  Trying 127.0.0.1...
  Connected to localhost.
  Escape character is '^]'.
  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 80 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 `L7/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. In Internet Explorer, choose Tools, then Options, then Connections, then LAN Settings. Check 'Use proxy server,' and click Advanced. 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 L7 directory in `tiny/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.

- In certain cases, a client closing a connection prematurely results in a `SIGPIPE` signal. This is particularly the case with Internet Explorer. To prevent your proxy from crashing, you should ignore this signal by adding this early in your code: `signal(SIGPIPE, SIG_IGN);`

- 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.

- Getting all the details right in socket programming is a pain. I recommend looking at examples (such as `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. `csapp.c` has provided useful wrapper functions as listed in the book.

  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.

- **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()`.

- 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.
- Random hint: Don't mix buffered and un-buffered reads/writes on a socket. Examples of buffered I/O: `fprintf(3), fscanf(3)`. Examples of un-buffered I/O: `write(2), read(2)`.
- Random hint 2: 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.)

Handin

- 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, if you are working alone, or
  - "ID_1+ID_2" where ID_1 and ID_2 are the andrew IDs of your two team members.
- 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-s02/labs/L7/handin
  ```

- Remove any extraneous print statements.