CS 213, Fall 2001
Lab Assignment L7: Logging Web Proxy
Assigned: Nov. 27, Due: Mon. Dec. 10, 11:59PM

Yinglian Xie (`ylxie@cs.cmu.edu`) is the lead person for this lab.

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 (for example, `www.silentbrowser.com`). Proxies are also used to block inappropriate sites by setting up URL filter lists. Filters can be applied to specific web pages or groups of websites. 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 filters and 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 results back to the browser, filtering inappropriate requests and keeping a log of all 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 by spawning a separate thread to deal with each request. While your proxy is waiting for a remote server to respond to a request so that it can serve one browser, it should be working on a pending request from another browser.

Logistics

The tar file for this Lab can be retrieved from
`/afs/cs.cmu.edu/academic/class/15213-f01/L7/L7.tar`

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

Part I: Implementing a web proxy

The first step is implementing a basic filtering and logging proxy. When started, your proxy should open a socket and listen for connections. When it gets a connection request (from a browser), it should accept the connection request, 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.

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 the Resources section below for help on these topics.

Filtering

When processing client requests, your proxy should block requests for certain websites. The URLs of the websites to be filtered are listed in the file `proxy.filter`, with each line containing one URL.

On reception of a client request, your proxy should check if the requested URL should be blocked. If not, your proxy should proceed with normal request processing; otherwise, your proxy should send a *permission denied* message back to the client in the following format:

```
HTTP/1.0  403 Forbidden
Content-type: text/html
\r\n
<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. Each line should be of the form:

```
Year-Month-Date Hour-Minute-Second 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:

```
2001-11-19 02-51-02 128.2.111.38 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 should be logged. You don't need to log requests that are filtered.

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., 3020 or 8104). Once you have selected a port, make sure your proxy program accepts the port number as the only argument. For example, if your choose 3020 as the port number, you should be able to run your proxy using the following command:

```
./proxy 3020
```

Part II: Dealing with multiple requests

Real servers do not process request sequentially, one at a time. Instead, they deal with multiple requests concurrently. 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 working proxy, you should alter it to handle multiple requests simultaneously by spawning a separate thread to process each request that comes in. In the thread 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).

However, with threading, you will have the problem that multiple peer threads might be trying to access the log file at the same time. 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 mutex or semaphore to control access to the file, so that only one peer thread can modify it at a time.

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

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 (25 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 (5 points). Up to 5 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 some tools to help you check your work and generate statistical information from your log.

driver.pl

The `driver.pl` program starts the proxy as a child process, sends it requests, checks replies from the proxy and displays a sample score for this Lab. The sample score would just give you an idea of how you are going to be graded. See Handin section below and web page for this lab for more grading information.

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

When using `driver.pl`, `port-number` is the port number at which your proxy will be listening to accept client requests and `proxy-log` is the log file output by your proxy. `lab-part` is the part of the lab to test and should be specified as "part1", "part2", or "all". For example, typing

```
unix> ./driver.pl 3120 proxy.log part1
```

would start testing your proxy correctness (Part I of this lab) at port number `3120`, assuming your proxy is generating a log file named `proxy.log`. And typing

```
unix> ./driver.pl 3120 proxy.log all
```

would test both your proxy correctness (Part I) and its ability to handle multiple requests concurrently (Part II).

We have also provided a set of tests in

```
/afs/cs/academic/class/15213-f01/L7/tests
```

to test your proxy against different types of HTTP requests.

The `driver.pl` program automatically tests all cases in the test set. It also checks whether your proxy can accept simultaneous requests. These tests may not be the same ones used to grade your proxy, but are representative of the tests used.

Log statistics generator

We have also provided a tool for generating statistic graphs based on your log file. The tool is modified from a open source software Awstats (`http://awstats.sourceforge.net/`). The statistical information to be generated includes:

- Number of visits, and unique visitors, list of last visits,
- Days of week and rush hours (pages, hits, kb for each hour and day of week),

- Domains/countries of hosts visitors (pages, hits, kb, 259 domains/countries detected),
- Most often pages viewed and entry pages,
- File types

The statistical information is going to be generated dynamically based on your proxy log. To view your log statistical results, you need to:

1. Register your group at the web server.
2. Make sure your proxy generates the log in a file named

   `/afs/cs/academic/class/15213-f01/L7/log/teamname.log`

   where "teamname" is the group ID provided by you on registration.

You can then browse the real time statistical graphs generated from your log by clicking links from the Lab 7 statistics web page or typing the following URL in your browser:

`http://bluegill.cmcl.cs.cmu.edu:3000/cgi-bin/getstat.pl?config=teamname`

where "teamname" is your supplied group ID.

Resources and Hints

- Read Chapter #11 and #12 in your textbook. They contain all the information you will need.
- 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 `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, because that's all your code is going to be able to handle.
- 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.weather.com/
http://www.cnn.com/
```

For each of the above tests, the IE browser would set up tens of concurrent connections with your proxy. These web sites are also the typical ones that would be used for grading Part II.

- For an example of a real, working web server in only 224 lines of source code, we are providing `tiny`, a minimal server written by Dave O'Hallaron. The `tiny` program can be found in the course directory under `L7/tiny`. 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 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.

  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.

- In certain cases, a client closing a connection prematurely will result in a SIGPIPE signal in proxy. This is particularly the case with Internet Explorer. When your proxy writes to a socket that has received a RST (TCP reset) from client, a SIGPIPE signal is sent to the proxy process. The default action of this signal is to terminate the proxy process. To prevent your proxy from being involuntarily terminated, you should ignore this signal by adding the following statement at the beginning of your code:

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

Your proxy should also check the return value of the Unix I/O function `write` and terminate the connection if an EPIPE error is catched. The `driver.pl` program also tests whether your proxy handles SIGPIPE signal correctly.

- Although you can use standard I/O functions such as `fgets` and `fprintf` for input and output on network sockets, you should be very careful since they pose some tricky problems. See Chapter #12 for the restrictions when using standard I/O functions on network sockets.

  We *strongly* recommend using robust file I/O functions `readn`, `writen` and `readline` from W.Richard Stevens's classic network programming text. Chapter #12 provides the detailed information of these functions. You should also use the reentrant version of `readline_r` to avoid race conditions between peer threads. The `readline_r` function takes as input an `Rline` structure, which is initialized by the `readline_rinit` function. Also you should initialize the `Rline` structure only **ONCE** before any subsequent use of `readline_r` on the same `Rline`.

- 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 "`HTTP/1.0`" or "`HTTP/1.1`". If it is "`HTTP/1.0`", you will simply need to read and echo it when you forward the request. However, if it is "`HTTP/1.1`", you will need to replace "`HTTP/1.1`" with "`HTTP/1.0`". This is because `HTTP/1.1` by default assumes the server will maintain a persistent connection with the client, which is not handled by the proxy.

  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.

- **IMPORTANT:** While using threads to handle connection requests, you must run them as *detached*, not *joinable*, to avoid memory leaks that will likely crash the fish servers. See Chapter #11 in the textbook 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, 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.)

Handin

We will be grading this lab by demo. You will need to sign up for a 20-minute slot. See the web page of this lab for detailed information.

NOTE that this lab is due right before the last day of class! There will be no late turn-ins and no extensions.

Have fun and Good luck!