Programming Assignment 1 — An HTTP Proxy Server

Due: October 8, 2015

A web proxy is a program that acts as an intermediary between a web client (browser) and a web server. A user configures a browser to use a proxy by entering the Internet address or domain name and port number for the proxy server into one of the browser's preference dialogs. Once this is done, the browser sends all HTTP requests to the proxy instead of to the servers identified by the URLs associated with the requests. The proxy accepts such requests from its clients, makes connections to the servers actually associated with the URLs, forwards the requests from the clients to the servers and then forwards the servers' responses to the appropriate clients.

There are many possible advantages to using a proxy. For example, the Williams library pays for subscriptions to many online databases and periodicals. As a result, the providers of these resources allow access from any machine using an IP address associated with Williams (i.e. having a 137.165 prefix). To ensure that members of the Williams community still can access these resources when off campus, the college maintains a proxy server. Instructions for using this server are available on the library website (http://library.williams.edu/computing/proxy.php). If an individual who is off campus accesses a publisher's web resources while using the proxy the request will appear to come from the Williams campus even if the computer running the browser is off campus.

Proxies can also be used to reduce web traffic through caching, to provide anonymity to users, and in other ways.

For this assignment, you will implement a simple web proxy in the C programming language under Linux.

A proxy acts as a server with relation to the browsers configured to use it. At the same time, it acts as a client of the servers to which it forwards requests. By implementing a proxy, you will gain experience with both client and server programming. Best of all, a proxy has almost no user interface. As a result, you will not have to expend much effort on aspects of the program that are not related to network programming.

Basic Functionality

Your proxy should expect one argument: the number of the port on which it should listen for requests. You will use a non-reserved port for your proxy (something between 1024 and 65535).

Your program will begin by creating a socket accepting connections on its port number. Each time a client connects to your proxy's port and sends a request, your proxy should examine the first line of the request to ensure it is valid and to extract the domain name and port number for the server. It will then create a new socket and connect that socket to the desired server. It will forward a slightly modified version of the request to the server and finally forward any response returned by the server to the client.

When a browser connects to a web server, it sends an HTTP request starting with a line of the form:

```
command /path HTTP/version
```

In most cases, the "command" specified will be GET, and the HTTP version will be 1.1. For example, if someone uses their browser to access the CS department web site the first line is likely to look like:

```
GET /index.html HTTP/1.1
```

On the other hand, when a browser connects to a proxy, it includes extra information in each request. In particular, an HTTP request sent to your proxy will look like:

```
command protocol://host:port/path HTTP/version
```

The ":port" is only included if the original request would have been directed to a port other than 80 on the server identified by "host".

For example, if someone uses their browser to access the CS department web site while the browser is configured to use your proxy, the first line of the request the browser sends to your proxy will probably look like:

```
GET http://www.cs.williams.edu/index.html HTTP/1.1
```

When your server receives such a request (or a similar HEAD or POST request), it should extract the host and port number (if any) from the incoming request and use this information to attempt to establish a connection to the specified server. If this connection succeeds, it should send a similar request to the server except:

- only the path component of the URL should be forwarded to the server.
- the HTTP version should be 1.0 rather than 1.1

The first change would be performed by any proxy. The second change is designed to make your proxy a bit simpler to implement by avoiding persistent connections. HTTP 1.1 servers and clients are supposed to try to reuse connections for multiple requests/response pairs when possible. If, however, a server receives a request indicating the client only understands HTTP 1.0, it is supposed to close its connection as soon as it finishes sending the first response.

Following the rules of HTTP, after the initial request line has been received, your proxy should accept header lines from the client and forward them to the server until a blank line is encountered. This blank line indicates the end of the request header and, in the case of GET and HEAD, requests, the end of the request itself. For a POST request, however, the header lines may be followed by binary data. Your proxy will also need to forward this data to the server.

There are two main approaches your proxy can take to determine when the end of the data it must forward for a POST connection has been reached. Basically, it can either detect end of file on the socket from the client, or it can count the bytes being forwarded. Since HTTP defines at last two ways the client can provide information about the length of the data it is sending (a "Content-length" header or "chunked" encoding), detecting end of file is probably easier. A socket signals end of file only after the connected machine closes its socket. Note that for this to happen the client talking to your server will have to recognize that it cannot maintain a persistent connection to your proxy.

When forwarding header lines, there is an extra step you may want to perform to ensure the server does not attempt a persistent connection. Remove any "Connection: ..." header line sent by the client and include a new "Connection: close" header line in what you send to the server.

If the request has been forwarded to the server correctly, the server will send back header lines followed by the requested data back to your proxy. Your proxy should read/receive this data through the socket connected to the server and forward it to the socket connected to the requesting client.

Initially, you should implement a proxy that only handles GET requests and only processes a single request at a time. For performance, however, a real proxy must be designed to handle multiple requests in parallel. This would certainly be required for a proxy that would be used by multiple clients simultaneously. Therefore, once you have a working sequential proxy, you should use threads to extend you proxy so that it can handle multiple requests simultaneously. Basically, for each request received, you will create a new thread to handle the request. You should also ensure that your final proxy handles HEAD and POST requests.

All of the code manipulating the sockets connection your proxy to its clients and servers should carefully check for error conditions that might be caused by unexpected behavior on the part of the clients or servers. If a client or server behaves oddly, that request may fail, but your proxy should not crash and the handling of other requests should not be impacted.

Your proxy should write a log of the requests it handles to standard output (or a separate file if you prefer). This log should provide information to help you demonstrate the correctness of your proxy when it is complete.

Some Implementation Details

A packet sniffer is a program that can intercept packets traveling through a network and display them in a readable form. When implementing a networked application like a proxy, a packet sniffer can be an invaluable debugging tool. Unfortunately, a packet sniffer can also be a horrifying security concern.

With this in mind, you will do your implementation work for this project in machines set up just for 336 this semester. The accounts created for our use on these machines will give you the privileges needed to run a packet sniffer named Wireshark. Since only we will be using these machines, you will only be able to sniff your own packets. I will provide login details for the machines we will be using in lab.

Writing a web proxy is a fairly popular programming assignment for both networks and system programming courses. If you search the web, you will find lots of other assignment handouts for similar project. Feel free to look for hints in such handouts as you complete this project (I did!). This probably means that there are solutions to the project on the web too. Don't look (I didn't!). On the other hand, I will give you one suggestion. One of the handouts I found was for a course using a text entitled "Computer Systems: A Programmer's Perspective" (http://csapp.cs.cmu.edu). It comes with a library file (csapp.c) that includes some extensions of the Unix I/O system calls that may be handy. In particular, to handle POST requests, your code will want to start by treating input from the socket as lines of text but then have to switch to treating it as uninterpreted binary data. Feel free to borrow from csapp.c for this. If nothing else, reading through this code will teach you about a number of Unix system calls you should be familiar with.

One challenge I encountered when I was writing my sample solution was finding sites I could use to test my handling of POST requests. I finally realized that POST requests are frequently used by sites that provide a feedback or suggestions page. Once you find such a page, check the HTML source to make sure it contains the text "method=POST".