Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.033 Lab, Spring 1999 — Handout 5 — A Web Proxy — February 19

Due date of preliminary design: Friday, February 26
Due date for implementation: Friday, March 12
Due date for design paper: Thursday, March 18

1 Introduction

In this lab you will have to make design decisions regarding what features to include and what tradeoffs to make between conflicting design criteria. Your final design paper will document these decisions. To help you with this process, we require that you also write a preliminary design document due next week.

In this handout, we call the *client* an application program that establishes connections for the purpose of sending requests[3]. Typically the client is a web browser (e.g., lynx or Netscape). We call the *server* an application program that accepts connections, processes requests, and sends responses for each request (e.g., the Apache web server)[1]. Note that a proxy acts as both a client and a server. Moreover, a proxy can even communicate with other proxies (e.g., a cache hierarchy).

2 Design Criteria

RFC 1945 defines a web proxy as a transparent, trusted intermediary between web clients and web servers. A web proxy differs significantly from a TCP proxy. Whereas a TCP proxy makes all decisions independent of the content of the data stream, the web proxy makes decisions based on the content of an HTTP request. Also, HTTP is a protocol built on top of TCP. As a result, the web proxy can make better optimizations (i.e., caching).

In general, a web proxy works by accepting requests from web clients and forwarding these requests to web servers on behalf of the clients. Often these requests must undergo some translation before they can be passed to servers. Thus, a proxy must interpret requests and perform any necessary translation before the forwarding the request. In some cases, however, a request can be serviced entirely by the web proxy. In such cases, the responses have been cached in the web proxy.

In this assignment, your proxy must be:

- *caching.* The proxy must cache web files. Web caches decrease latency at the cost of serving stale data[5].
- *asynchronous.* Your web proxy must allow multiple clients to operate simultaneously. The proxy should not allow a client or server to block other clients or servers from communicating.

- *transparent.* The proxy should be transparent to the client and server. Your design cannot rely on any special modification to the client or server.

Your proxy should function correctly for any HTTP/1.0 GET, POST, or HEAD request. However, you may ignore any references to cookies and authentication[1]. In addition, we are requiring that you make at least one major performance optimization to the web proxy.

2.1 Desirable Properties of Your Web Proxy

Your web proxy should tolerate many simultaneous requests. The web proxy will accept connections from multiple clients and forward them using multiple connections to the appropriate servers. No client or server should be able to hang the web proxy by refusing to read or write data on its connection.

You may use as much of the code from the last assignment as you would like, including any code we release. Thus, you may use the asynchronous package from the previous assignment or a threads package of your own choosing. However, we must be able to compile and test your code in Solaris or Redhat Linux.

You will also need to devise a way to keep the web cache coherent. Clients dislike stale pages. But unlike file system caches, stale home pages do have some value[5]. For this lab, you could use some form of a *validation check* to keep the cache up-to-date. For example, you could test for freshness by using the `If-Modified-Since` or `Expires` header. In this case, servers determine determine the freshness of a document. On the other hand, the proxy could cache pages for a fixed period of time. In this case, the proxy determines freshness.

Search RFC 1945 for any warnings about "proxy" behavior. The lab TA's will test that your proxy handles requirements as stated in RFC 1945.

2.2 Optimizing Your Web Proxy

In addition to caching, we require that you perform one major optimization on your web proxy. The optimization should help speed up the handling of requests in some way. We leave the choice of the optimization to you. However, we will briefly describe a few examples later in this section to help get you thinking.

Your optimization will most likely be in one of two areas: *caching* or *prefetching*. Although we require you to implement a cache, you may choose to use more sophisticated caching strategy as a major optimization. Prefetching is a strategy in which the web proxy fetches documents from servers before they are actually requested by the client. The proxy anticipates future requests by the clients based upon their past requests. Any prefetching optimization would qualify as a major optimization.

Examples of optimizations you might use are:

- *Multiple Layer Caching.* You could implement your cache using multiple levels of memory. You will probably find that you will need to use a disk cache to get a large enough cache.

[1] Although you are welcome to handle cookies and authentication.

However, for a request that results in a cache hit, you still have to get the file from the disk before transferring it to the client. To improve performance, you could implement some strategy through which some documents could be moved from the disk cache into a main memory cache in anticipation of a future request.

- *Structured Caching.* Web documents have links to other documents. You could implement a strategy that understands the relationship between files. The proxy would then manage these files in the cache, taking this relationship into account. For example, if an HTML page contained a link to a second page, you might decide to throw out the second page only if the first page was also stale.
- *Link Prefetching.* When the web proxy fetches an HTML page for a client, it can anticipate that the client will probably follow one of the links on the page. To take advantage of this information, the proxy could fetch links before the client actually requests them.
- *Watch Dog.* Some web documents will have an expiration timestamp indicating to a proxy when they should be discarded. If such documents are frequently requested, the proxy may take it upon itself to fetch a fresh copy of the document before it expires in the cache.
- *Persistent Connections.* When a client requests a web document from a server, it will probably request additional documents a short while later. Instead of opening a new TCP connection for each HTTP request, a proxy can keep the server connection open for a limited time even after the request was served. By reusing a TCP connection, the proxy eliminates much of the overhead in starting a new connection. Read up on the `Keep-Alive` header in RFC 2068. If you choose this optimization, will you actually implement HTTP/1.0 with parts of HTTP/1.1.

Of course, you do not actually have to optimize using caching or prefetching as the last example shows. However for any optimization you make, you must get the approval of a lab TA ahead of time. A good place to get our approval is in the preliminary design report next week. Before you write up the preliminary report, you should have already weighed your design options carefully, and have made design decisions regarding the implementation of your web proxy.

To help you implement your optimization, we will provide code to parse and tokenize HTTP headers and HTML documents. This code will be ready after you turn in the preliminary design document.

3 The HTTP Protocol

The Hypertext Transfer Protocol (HTTP) is the most commonly used protocol on the web today. For this lab, you will use a less bleeding-edge specification: HTTP version 1.0.

The HTTP protocol assumes a reliable connection and, in current practice, uses the TCP protocol to provide this reliable connection. The TCP protocol provides the reliable transport of bytes between programs on two separate machines. Luckily for us, the TCP protocol is built into the UNIX operating system.

The HTTP protocol is a request/response protocol. When a client opens a connection, it immediately sends its request for a file. A web server then responds with the file or an error message. You can try out the protocol yourself. For example, try:

```
(~/)% telnet web.mit.edu 80
```

Then type

```
GET /6.033/www/ HTTP/1.0
```

followed by two carriage returns. See what you get.

To form the path to the file to be retrieved on a server, the client takes everything after the machine name and port number. For example, `http://www.mit.edu/original/` means we should ask for the file `/original/`. If you see a URL with nothing after the machine name and port, then `/` is assumed[2].

On most servers, the HTTP server lives on port 80. However, it turns out that port 80 is protected on most UNIX systems, so we will have to run our web proxy on a higher port (≥ 1024). To use other ports, we need to modify our URLs a bit, adding the port number after the machine name. For example, entering `http://www.mit.edu:8008/` into your favorite web browser connects to the machine www.mit.edu on port 8008 using the HTTP protocol.

The format of the request for HTTP is quite simple. A request consists of a method followed by arguments, each separated by a space and terminated by a carriage return/linefeed pair. Your web proxy should support three methods: `GET`, `POST`, and `HEAD`[3]. Methods take two arguments: the file to be retrieved and the HTTP version. Additional headers can follow the request. The web proxy will especially care about the following headers: `Allow, Date, Expires, From, If-Modified-Since, Pragma: no-cache, Server`. However, your proxy must handle all HTTP/1.0 headers[3]. Fortunately, the web proxy can forward most of the requests verbatim to the appropriate server. Only a handful of headers require proxy intervention.

Once the proxy receives the request line, the it should continue reading the input from the client until it encounters a blank line. The proxy should then fetch the appropriate file and send back a response (usually the file contents) and close the connection[3].

4 Using a Web Proxy

To use a web proxy, you must configure your web browser. For Lynx, wget, or Mosaic, you must set an environment variable. The following sets your proxy to web.mit.edu.

```
(~/)% setenv http_proxy http://web.mit.edu/
```

In Netscape, find the Network Preferences and manually setup a proxy. For instance, you can set the HTTP proxy to web.mit.edu and the port to 80. Remember to revert your changes. Not all requests will work transparently through the web.mit.edu proxy.

[2] The server determines what page to return when just given `/`. Typically this default page is `index.html` or `home.html`

[3] A persistent connection would not close here.

5 HTTP in Action!

How does one watch an HTTP request in action? To simulate an HTTP request, most people will use `telnet`. However, `telnet` does not let you watch incoming HTTP requests. For a more sophisticated connection, use netcat. Netcat (`nc`) lets you read and write data across network connections using UDP or TCP[10]. The format of commandline arguments may vary depending on your version of `nc` or your operating system. Type `nc -h` for the usage. You can also read `/mit/sipb/src/netcat/README` for detailed information and examples. This command listens to the network on port 8000:

```
(~/)% add sipb
(~/)% nc -l  -p 8000 -v localhost
listening on [any] 8000 ...
```

Now point your favorite web browser to `http://localhost:8000/`. My version of Netscape generates:

```
connect to [127.0.0.1] from localhost [127.0.0.1] 5854
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.01 (X11; U; Linux 2.0.30 i586)
Host: localhost:8000
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The first line asks for a file called `/` using HTTP version 1.0. Look in RFC 1945 for details on the remaining lines. Lynx produces a similar request:

```
connect to [127.0.0.1] from localhost [127.0.0.1] 5917
GET / HTTP/1.0
Host: localhost:8000
Accept: application/postscript, image/gif, application/postscript, */*;q=0.001
Accept-Encoding: gzip, compress
Accept-Language: en
User-Agent: Lynx/2.6  libwww-FM/2.14
```

If you configure your web browser with `localhost.mit.edu` and port 8000 as a proxy, retrieving `http://c0re.l0pht.com/~weld/netcat/readme.html` produces:

```
connect to [127.0.0.1] from localhost.mit.edu [127.0.0.1] 2328
GET http://c0re.l0pht.com/~weld/netcat/readme.html HTTP/1.0
If-Modified-Since: Thursday, 12-Sep-96 02:25:13 GMT; length=63340
Referer: http://c0re.l0pht.com/~weld/netcat/
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/3.01Gold (X11; U; OpenBSD 2.2 i386)
Pragma: no-cache
Host: c0re.l0pht.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The above shows what a web browser sends to a web proxy. Obtaining sample data from a real web proxy is a little trickier. Here we use `web.mit.edu` as a proxy (it is both a web server and proxy server). Set your browser to use `web.mit.edu` on port 80 as a proxy. Then run locally:

```
nc -v  -p 8000 -l web.mit.edu
listening on [127.0.0.1] 8000 ...
```

When I ask my web browser for `http://ultrasparc.mit.edu:8000/` (my local machine), netcat reports:

```
connect to [18.152.0.140] from SALTICUS-PECKHAMAE.MIT.EDU [18.69.0.38] 48943
GET / HTTP/1.0
Proxy-Connection: Keep-Alive
Host: ultrasparc.mit.edu:8000
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
User-Agent: Mozilla/3.01 (X11; U; Linux 2.0.30 i586)  via proxy gateway
  CERN-HTTPD/3.0 libwww/2.17
```

Try this on Athena. Look for differences between the web browser's request and the corresponding proxy request.

6 Administrivia

6.1 Where to Start?

Read over some of the suggested literature at the end of this document. You can find the Internet Request for Comments (RFC's) in the `rfc` locker: `attach rfc; cd /mit/rfc; ls rfc1945.txt`.

After you have a general understanding of the problem, play with netcat and begin your web proxy design. Once you have convinced yourself of the correctness of your design, you should implement the web proxy. You will likely discover new, fascinating problems and will need to modify your design appropriately.

6.2 Hand-in Procedure

When you are ready to hand in the lab, copy your code to `/mit/6.033/lab/handins/p2/$USER`. You should not copy object code or binary executables. We will compile the web proxy by typing `make`.

The project should include a valid Makefile and the source code for the web proxy. The Makefile should create a proxy named `webproxy`. The proxy must support at least two command-line options: `-help` should explain the usage of your web proxy and `-p` *<port>* should run the proxy on port *<port>*. You may add any other options as long as you document them. If you make any improvements to the web proxy specifications, please include comments in a README file.

For example, to run a proxy on port 8000 of your local machine:

```
(~/)% ./webproxy -p 8000 &
```

6.3 Due Dates

A preliminary design document describing your web proxy is due at the beginning of recitation on **Friday, February 26**. This is a very important deadline because we will conduct short oral presentations in class. You should not prepare a formal presentation – it will be impromptu. The document should contain a description of the web proxy with reasoning as to how it meets the design constraints. Any additional twists that you may have decided to add to the specification should also be in the paper. Also, the paper should clearly describe the optimization that you have chosen. We have imposed no length requirements for the document, but you should try to specify as much of the design as possible so that we can help you by spotting potential problems early.

The final code for the web proxy is due on **Friday, March 12**. Follow the directions in section 6.2 to hand in your project.

The final design document for the web proxy is due on **Thursday, March 18**. This design paper should be comparable in length (maximum 10 pages) and quality to the first design projects for 6.033. You can use the final design document in place of your 6.033 design project 1 (eligible for the Phase II writing requirement). Read the Mayfield Handbook for a detailed description on how to write a design document[11]. At a minimum, your final design document should contain an abstract, introduction, design criteria, implementation, recommendation, and summary.

6.4 Collaboration

Like the TCP Proxy, this is an individual project. The design documents and code must be yours, but you are otherwise free (and encouraged) to discuss the design and implementation details with other 6.033 lab students. Use the acknowledgments section of your final design document to give credit where it is due.

One last piece of advice: Do not wait until the last week to begin.

References

[1] *Apache Web Proxy*, `http://www.apache.org/docs/mod/mod_proxy.html`.

[2] T. Berners-Lee. *Propagation, Replication and Caching on the Web*, `http://www.w3.org/Propagation/`.

[3] T. Berners-Lee, et al. *RFC 1945: Hypertext Transfer Protocol – HTTP/1.0*, `http://web.mit.edu/rfc/rfc1945.txt`, May 1996.

[4] *CERN Web Proxy*, `http://www.w3.org/Daemon/User/Proxies/Proxies.html`.

[5] A. Dingle, T. Partl. *Web Cache Coherence*, `http://sun3.ms.mff.cuni.cz/~dingle/webcoherence.html`, May 1996.

[6] *SquidCache*, `http://squid.nlanr.net/Squid/`.

[7] R. Fielding, et al. *RFC 2068: Hypertext Transfer Protocol – HTTP/1.1*, `http://web.mit.edu/rfc/rfc2068.txt` January 1997.

[8] J. Franks, et al. *RFC 2069: An Extension to HTTP : Digest Access Authentication*, http://web.mit.edu/rfc/rfc2069.txt January 1997.

[9] J. C. Mogul, et al. *RFC 2145: Use and Interpretation of HTTP Version Numbers*, http://web.mit.edu/rfc/rfc2145.txt May 1997.

[10] Netcat. http://c0re.l0pht.com/~weld/netcat/.

[11] L. Perelman, et al. *The Mayfield Handbook of Technical & Scientific Writing*, http://tute.mit.edu/afs/athena/course/21/21.guide/www/home.htm.

[12] D. Wessels. *Web Caching Reading List*, http://ircache.nlanr.net/Cache/reading.html.