6.824 - Fall 2002

6.824 Lab 1: A simple web proxy

Introduction

Please read "Getting started with 6.824 labs" before starting this assignment. You will also need "Using TCP through sockets" at a later stage.

In this lab assignment you will write a simple web proxy. A web proxy is a program that reads a request from a browser, forwards that request to a web server, reads the reply from the web server, and forwards the reply back to the browser. People typically use web proxies to cache pages for better performance, to modify web pages in transit (e.g. to remove annoying advertisements), or for weak anonymity.

You'll be writing a web proxy to learn about how to structure servers. For this assignment you'll start simple; in particular your proxy need only handle a single connection at a time. It should accept a new connection from a browser, completely handle the request and response for that browser, and then start work on the next connection. (A real web proxy would be able to handle many connections concurrently.)

In this handout, we use *client* to mean an application program that establishes connections for the purpose of sending requests[3], typically a web browser (e.g., lynx or Netscape). We use *server* to mean an application program that accepts connections in order to service requests by sending back responses (e.g., the Apache web server)[1]. Note that a proxy acts as both a client and server. Moreover, a proxy could communicate with other proxies (e.g., a cache hierarchy).

Design Requirements

Your proxy will speak a subset of the HTTP/1.0 protocol, which is defined in RFC 1945. You're only responsible for a small subset of HTTP/1.0, so you can ignore most of the spec. You should make sure your proxy satisfies these requirements:

- GET requests work.
- Images/Binary files are transferred correctly.
- Your webproxy should properly handle Full-Requests (RFC 1945, Section 4.1) up to, and including, 65535 bytes. You should close the connection if a Full-Request is larger than that.
- You must support URLs with a numerical IP address instead of the server name (e.g. http://18.181.0.31/).
- You are not allowed to use `fork()`.
- You may not allocate more than 100MB of memory.

- You can not have more than 32 open file descriptors.
- Your proxy should correctly service each request if possible. If an error occurs, and it is possible for the proxy to continue with subsequent requests, it should close the connection and then proceed to the next request. If an error occurs from which the proxy cannot reasonably recover, the proxy should print an error message on the standard error and call `exit(1)`. There are not many non-recoverable errors; perhaps the only ones are failure of the initial `socket()`, `bind()`, `listen()` calls, or a call to `accept()`. The proxy should never dump core except in situations beyond your control (e.g. a hardware or operating system failure).

You do **not** have to worry about correct implementation of any of the following features; just ignore them as best you can:

- POST or HEAD requests.
- URLs of any type other than http.
- HTTP-headers (RFC 1945, Section 4.2).

If your browser can fetch pages and images through your proxy, and your proxy passes our tester (see below), you're done.

HTTP example without a web proxy

HTTP is a request/response protocol that runs over TCP. A client opens a connection to a web server and sends a request for a file; the server responds with some status information and the file contents, and then closes the connection.

You can try out HTTP yourself:

```
% telnet web.mit.edu 80
```

This connects to `web.mit.edu` on port 80, the default port for HTTP (web) servers.

Then type

```
GET / HTTP/1.0
```

followed by two carriage returns. This ends the header section of the request. The server locates the web page and sends it back. You should see it on your screen.

To form the path to the file to be retrieved on a server, the client takes everything after the machine name. For example, `http://web.mit.edu/resources.html` means we should ask for the file `/resources.html`. If you see a URL with nothing after the machine name and port, then `/` is assumed---the server figures out what page to return when just given `/`. Typically this default page is `index.html` or `home.html`.

On most servers, the HTTP server lives on port 80. However, one can specify a different port number in the URL. For example, typing `http://web.mit.edu:2206` in your browser will tell it to find a web server on port 2206 on web.mit.edu. (No, this doesn't work for this address.)

HTTP (request) example with a web proxy

Before you can do this example, you need to tell your web browser to use a web proxy. This explanation assumes you are running Mozilla, but things should be remarkably similar for Netscape. Choose ``Edit'' ---> ``Preferences''. Then choose ``Advanced'' ---> ``Proxies''. Click on ``Manual proxy configuration''. Now set the ``HTTP proxy'' to `sure.lcs.mit.edu` and port 3128. Mozilla will now send all HTTP request to this web proxy rather than directly to web servers.

Lynx---a poor man's browser---can be told to use this web proxy by setting the environment variable `http_proxy` to `sure.lcs.mit.edu:3128`.

Now to the real stuff.

You can use `nc` to peak at HTTP requests that a browser sends to a web proxy. `nc` lets you read and write data across network connections using UDP or TCP[10]. The class machines have `nc` installed.

First we'll examine the requests that a browser sends to the proxy. We'll use `nc` to listen on a port and direct our web browser (Lynx) to use that host and port as a proxy. We're going to let `nc` listen on port 8888 and tell Lynx to use a web proxy on port 8888.

```
% nc -lp 8888
```

This tells `nc` to listen on port 8888. Chances are that you will have to choose a different port number than 8888 because someone else may be using that port. Choose a number greater than 1024, less than 65536. Now try, on the same machine, to retrieve a web page port 8888 as a proxy:

```
% env http_proxy=http://localhost:8888/ lynx -source
http://www.yahoo.com
```

This tells Lynx to fetch `http://www.yahoo.com` using a web proxy on port 8888, which happens to be our spy friend `nc`.

Netcat neatly prints out the request headers that Lynx sent:

```
% nc -lp 8888
GET http://www.yahoo.com/ HTTP/1.0
Host: www.yahoo.com
Accept: text/html, text/plain, application/vnd.rn-rn_music_package,
application/x-freeamp-theme, audio/mp3, audio/mpeg, audio/mpegurl,
audio/scpls, audio/x-mp3, audio/x-mpeg, audio/x-mpegurl, audio/x-scpls,
audio/mod, image/*, video/mpeg, video/*
Accept: application/pgp, application/pdf, application/postscript,
message/partial, message/external-body, x-be2, application/andrew-
inset, text/richtext, text/enriched, x-sun-attachment, audio-file,
postscript-file, default, mail-file
Accept: sun-deskset-message, application/x-metamail-patch,
application/msword, text/sgml, */*;q=0.01
Accept-Encoding: gzip, compress
Accept-Language: en
User-Agent: Lynx/2.8.4rel.1 libwww-FM/2.14 SSL-MM/1.4.1 OpenSSL/0.9.6b
```

The GET request on the first tells the proxy to get file `http://www.yahoo.com` using HTTP version 1.0. Notice how this request is quite different from the example without a web proxy! The protocol and machine name (`http://www.yahoo.com`) are now part of the request. In the previous example this part was omitted. Look in RFC 1945 for details on the remaining lines. (It's effective reading material if you really can't sleep and Dostoevsky didn't do the trick.)

HTTP (reply) example with a web proxy

The previous example shows the HTTP request. Now we'll try to see what a real web proxy (sure.lcs.mit.edu port 3128) sends to a web server. To achieve this we use `nc` to be a fake web server. Start the ``fake server'' on `pain.lcs.mit.edu` with the following command:

```
% nc -lp 8888
```

Again, you may have to choose a different number if 8888 turns out to be taken by someone else.

```
% env http_proxy=http://sure.lcs.mit.edu:3128/ lynx -source
http://pain.lcs.mit.edu:8888
```

Needless to say, you should replace 8888 by whatever port you chose to run `nc` on. `nc` will show the following request:

```
% nc -lp 8888
GET / HTTP/1.0
Accept: text/html, text/plain, text/sgml, video/mpeg, image/jpeg,
image/tiff, image/x-rgb, image/png, image/x-xbitmap, image/x-xbm,
image/gif, application/postscript, */*;q=0.01
Accept-Encoding: gzip, compress
Accept-Language: en
User-Agent: Lynx/2.8.4rel.1 libwww-FM/2.14
Via: 1.0 supervised-residence.lcs.mit.edu:3128 (Squid/2.4.STABLE4)
X-Forwarded-For: 18.26.4.76
Host: pain.lcs.mit.edu:8888
Cache-Control: max-age=259200
Connection: keep-alive
```

Notice how the web proxy stripped away the `http://pain.lcs.mit.edu:8888` part from the request!

Your web proxy

Your web proxy will have to translate between requests that the client makes (the one that starts with ``GET http://machinename") into requests that the server understands. So far for the bad news. The good news is that we provide you with some helpful code that will make this very easy to do.

Your web proxy will listen on a port other than port 80, so as to avoid conflicts with regular web servers.

Once the request line has been received, the web proxy should continue reading the input from the client until it encounters a blank line. The proxy should then fetch the URL from the appropriate server, forward the response back to the client, and close the connection. The proxy should forward response data as it arrives, rather than buffering the entire response; this allows the proxy to handle huge responses without running out of memory.

Your web proxy has to support the `GET` method only [3]. A GET method takes two arguments: the file to be retrieved and the HTTP version. Additional headers may follow the request.

Getting Started

Enough talking. Now do something.

We have provided a skeleton webproxy directory. It is available in `/home/6824/labs/webproxy1.tgz`. The following sequence of commands should yield a compiled version of the server you should extend to pass the tests.

```
% tar xzvf /home/6824/labs/webproxy1.tgz
% cd webproxy1
% gmake
```

The tarball contains `http.C`, `http.h`, `Makefile`, and `webproxy1.C`. The first two files will help you parse HTTP requests. The `Makefile` is, as its meaningful name implies, a Makefile. `Webproxy1.C` is a pretty useless web server that, nonetheless, should help you on your way.

http.C and http.h : a HTTP parser

We have provided a parser for proxy-style HTTP requests. It is implemented in the files `http.C` and `http.h` that are included in the tarball.

http.h defines the class `httpreq` that inherits from the class `httpparse` (if you are unfamiliar with C++ inheritance, consult the Stroustrup C++ language guide referenced in the course information page. Don't drop this book on someone's face. It's a pretty hefty book.)

To parse a request, first create a `httpreq` object. Then, parse the (potentially incomplete) HTTP request by feeding it to `int parse (char *buf, ssize_t len)` until it returns 1, indicating that the headers are complete. `buf` should be the buffer that contains the (potentially incomplete) HTTP request. `len` is the length of the HTTP request fragment in `buf`. Notice that `parse` needs to see the **whole** request you have read so far.

`parse` returns 1 if the HTTP request is complete, 0 if it needs more data to complete, or -1 on a parse error. `parse` does not modify the contents of `buf`. Once `parse` returns 1, you can call---amongst others---the following methods on the calling `httpreq`.

- **char* method()** The 'type' of request (POST, GET, HEAD)
- **char* host()** The destination host
- **short port()** The destination port
- **char* path()** The filename part of the requested URL
- **char* url()** The requested URL

Here's a simple program that illustrates the use of `httpreq`.

```
#include <stdio.h>
#include "http.h"

int
main()
{
  httpreq *r = new httpreq();
  char buf[512];
  int ret;

  // incomplete header
  strcpy(buf, "GET http://web.mit.edu/index.html");
  ret = r->parse(buf, strlen(buf));
  printf("ret %d file %s\n",
         ret,
         ret == 1 ? r->path() : "(none)");

  // complete header
  strcat(buf, " HTTP/1.0\r\n\r\n");
  ret = r->parse(buf, strlen(buf));
  printf("ret %d file %s\n",
         ret,
         ret == 1 ? r->path() : "(none)");

  delete r;
  exit(0);
}
```

Documentation

You may want to read "Using TCP through sockets" to learn about socket programming in C/C++. Also, take a look at the references at the bottom of this page.

Running and testing the proxy

Your proxy program should take exactly one argument, a port number on which to listen. For example, to run the proxy on port 2000:

```
% ./webproxy1 2000
```

As a first test of the proxy you should attempt to use it to browse the web. Set up your web browswer to use one of the class machines running your proxy as a proxy and experiment with a variety of different pages.

When you think your proxy is ready, you can run it against the test program `webproxy1-test`, our tester, the source of which is in `~6824/labs/webproxy1/webproxy1-test.C`. Run the tester with your proxy as an argument:

```
% /home/6824/labs/webproxy1/webproxy1-test ./webproxy1
```

Note that this may take several minutes to complete. The test program runs the following tests:

Ordinary fetch

This test is the "normal case". We send a normal HTTP 1.0 GET request and expect the correct web page.

Split request

This tests splits the HTTP request in two chunks. The first chunk contains a partial HTTP request. The second chunk completes the first after which the tester expects the correct web page contents to come back.

Large request

The tester does a request of exactly 65535 bytes.

Large response

The tester fetches a web page larger than the maximum amount of memory available to your web proxy.

Zero-size response

The tester fetches a web page without a body.

Recover after bad connect

The tester sends a request with a URL that specifies a false port. Your proxy will attempt to make a connection to a bogus port. Soon thereafter, the tester tries to fetch a valid page to see if your proxy is still doing ok.

Malformed request

The tester sends an HTTP request that is not syntactically correct. After that, it tries to fetch a valid page to see if it your proxy is still doing ok.

Premature client close()

The tester sends a partial HTTP request and then closes the connection. After that, it tries to fetch a valid page to see if it your proxy is still doing ok.

Infinitely long request

The tester swamps your proxy with a request larger than 65535 bytes. The tester expects your proxy to close the connection. After that, it tries to fetch a valid page to see if it your proxy is still doing ok.

Stress test

The tester stress tests your web proxy with a ruthless combination of ordinary fetches, split requests, malformed requests, and large responses. This may expose memory leaks, unclosed connections, and random other bugs.

Collaboration policy

You must write all the code you hand in for the programming assignments, except for code that we give you as part of the assigment. You are not allowed to look at anyone else's solution (and you're not allowed to look at solutions from previous years). You may discuss the assignments with other students, but you may not look at or copy each others' code.

Handin procedure

You should hand in a gzipped tarball `webproxy1-handin.tgz` produced by `gmake dist`. Copy this file to `~/handin/webproxy1-handin.tgz`. Do **not** make this file world readable! We will use the **first** copy of the file that we can find after the deadline---we try every few minutes. Don't bother to copy a new version over the old one hoping that we will use it instead. We won't.

References

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

2 T. Berners-Lee, et al. *RFC 1945: Hypertext Transfer Protocol - HTTP/1.0*, May 1996.

3 *CERN Web Proxy*. http://www.w3.org/Daemon/User/Proxies/Proxies.html.

4 Netcat. http://www.atstake.com/research/tools/ .