CS 15-213, Spring 2010
Lab Assignment L6: Writing a Caching Web Proxy
Assigned: Thursday, April 16th, 2010
Due Date: Thursday, April 29th, 2010
Last Possible Date To Submit: Sunday, May 2nd, 2010

1 Introduction

A web proxy is a program that 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 directly in response to future requests rather than going to the server.

In this lab, you will write a simple proxy that caches web objects. In the first part of the lab, you will set up the proxy to accept a request, forward the request to the server, and return the result back to the browser. 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, you will upgrade your proxy to deal with multiple open connections at once. Your proxy should spawn a separate thread to deal with each request. This will give you an introduction to dealing with concurrency, a crucial systems concept. Finally, you will turn your proxy into a proxy cache by adding a simple main memory cache of recently accessed web pages.

2 Logistics

Unlike previous labs, you can work individually or in a group of two on this assignment. The lab is designed to be doable by a single person, so there is no penalty for working alone. You are, however, welcome to team up with another student if you wish.

There is no autograding for this lab. We will grade your labs after the submission deadline. It is up to yourself to make sure you have throughly tested your own lab and have fixed all the bugs. Teams must

register to work together. We will release a team signup page in the upcoming week.

3 Hand Out Instructions

Start by downloading `proxylab-handout.tar` from Autolab to a protected directory in which you plan to do your work. Then give the command `tar xvf proxylab-handout.tar`. This will cause a number of files to be unpacked in the directory. You should start working with `proxy.c` and you may create any other files you like.

NOTE: Transfer the tarball to a fish machine before unpacking it. Some operating systems and file transfer clients wipe out Unix file permission bits.

The `proxy.c` file should eventually contain the bulk of the logic for your proxy.

4 Part I: Implementing a Sequential Web Proxy

The first step is implementing a basic sequential proxy that handles requests one at a time. When started, your proxy should open a socket and listen for connection requests on the port number that is passed in on the command line. (See the section "Port Numbers" below.)

When the proxy receives a connection request from a client (typically a web browser), the proxy should accept the connection, read the request, verify that it is a valid HTTP request, and parse it to determine the server that the request was meant for. It should then open a 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.

Processing HTTP Requests

When an end user enters a URL such as `http://www.yahoo.com/news.html` into the address bar of the browser, the browser sends an HTTP request to the proxy that begins with a line looking something like this:

```
GET http://www.yahoo.com/news.html HTTP/1.0
```

In this case the proxy will parse the request, open a connection to `www.yahoo.com`, and then send an HTTP request starting with a line of the form:

```
GET /news.html HTTP/1.0
```

to the server `www.yahoo.com`. Please note that all lines end with a carriage return `'\r'` followed by a line feed `'\n'`, and that HTTP request headers are terminated with an empty line. Since a port number was

not specified in the browser's request, in this example the proxy connects to the default HTTP port (port 80) on the server. The web browser may specify a port that the web server is listening on, if it is different from the default of 80. This is encoded in a URL as follows: `http://www.example.com:8080/index.html` The proxy, on seeing this URL in a request, should connect to the server `www.example.com` on port 8080.

The proxy then simply forwards the response from the server on to the browser. **IMPORTANT:** Please read the brief section in your textbook (Sec 12.5.3, HTTP transactions) to better understand the format of the HTTP requests your proxy should send to a server.

Port Numbers

Every server on the Internet waits for client connections on a well-known port. The exact port number varies from Internet service to service. The clients of your proxy (your browser for example), will need to be told not just the hostname of the machine running the proxy, but also the port number on which it is listening for connections.

Your proxy should accept a command-line argument that gives the port number on which it should listen for connection requests. For example, the following command runs a proxy listening on port 15213:

```
unix> ./proxy 15213
```

You will need to specify a port each time you wish to test the code you've written. Since there might 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 (greater than 1K and less than 64K) 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). We have provided a sample script (`port_for_user.pl`) that will generate a port number based on your userid:

```
unix> ./port_for_user.pl droh
droh: 45806
```

We strongly advise you to use this port number for you proxy rather than randomly picking one each time you run. This way, you will not trample on other students' ports.

5 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 working sequential proxy, you should alter it to handle multiple requests simultaneously. The approach we suggest is using 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).

6 Part III: Caching Web Objects

In this part you will add a cache to your proxy that will cache recently accessed content in main memory. HTTP actually defines a fairly complex caching model where web servers can give instructions as to how the objects they serve should be cached and clients can specify how caches are used on their behalf. In this lab, however, we will adapt a somewhat simplified approach.

When your proxy queries a web server on behalf of one of your clients, you should save the object in memory as you transmit it back to the client. This way if another client requests the same object at some later time, your proxy needn't connect to the server again. It can simply resend the cached object.

Obviously, if your proxy stored every object that was ever requested, it would require an unlimited amount of memory. To avoid this (and to simplify our testing) we will establish a maximum cache size of

$$\mathrm{MAX\_CACHE\_SIZE} = 1\,\mathrm{MB}$$

and evict objects from the cache when the size exceeds this maximum.

We will require a simple least-recently-used (LRU) cache replacement policy when deciding which objects to evict. One way to achieve this is to mark each cached object with a time-stamp every time it is used. When you need to evict one, choose the one with the oldest time-stamp. Note that reads and writes of a cached object both count as "using" it.

Another problem is that some web objects are much larger than others. It is probably not a good idea to delete all the objects in your cache in order to store one giant one, therefore we will establish a maximum object size of

$$\mathrm{MAX\_OBJECT\_SIZE} = 100\,\mathrm{KB}$$

You should stop trying to cache an object once its size grows above this maximum. The easiest way to implement a correct cache is to allocate a buffer for each active connection and accumulate data as you receive it from the server. If your buffer ever exceeds $\mathrm{MAX\_OBJECT\_SIZE}$, then you can delete the buffer and just finish sending the object to the client. After you receive all the data you can then put the object into the cache. Using this scheme the maximum amount of data you will ever store in memory is actually

$$\mathrm{MAX\_CACHE\_SIZE} + T * \mathrm{MAX\_OBJECT\_SIZE}$$

where T is the maximum number of active connections.

Since this cache is a shared resource amongst your connection threads, you **must** make sure your cache is thread-safe. A simple strategy to make your cache thread-safe is to use a rwlock to ensure that a thread writing to the cache is the only one accessing it.

7 Evaluation

Your lab will be autograded after the submission period is over and your grade will be available on autolab at that point in time. If you would like to contest your grade you may do so in writing to the course staff.

There are a total of 100 points for this assignment. Points will be assigned based on the the following criteria:

- Basic sequential proxy (30 points). Credit will be given for a program that accepts connections, forwards the requests to a server, and sends the reply back to the browser.
  - 25 points will be given for properly accepting connections, forwarding requests to the server, and sending replies back to the browser. Your proxy must be able to handle sequences of requests, with memory and system resources recovered between requests.
  - 5 points will be given for handling the SIGPIPE signal correctly (Please see the hints in Section 10).
- Handling concurrent requests (30 points). Your proxy is required to handle multiple concurrent connections so that one slow web server does not hold up other requests from completing. Memory and system resources must be recovered after servicing requests. Also, your proxy must be free of race conditions.
  - 15 points will be given for accepting and servicing multiple concurrent connections.
  - 5 points will be given for properly recovering memory and system resources.
  - 10 points will be given for eliminating race conditions.
- Caching (30 points). You will receive 30 points for a correct thread-safe cache.
  - 15 points will be given if your proxy returns cached objects when possible. Your cache must adhere to the cache size limit and the maximum object size limit, and must not insert duplicate entries into the cache.
  - 7.5 points will be given for a proper implementation of the specified least-recently-used (LRU) eviction policy.
  - 7.5 points will be given for proper use of locks. Your cache must be free of race conditions, deadlocks, and excessive locking. Excessive locking includes keeping the cache locked across a network system call or not allowing multiple connection threads to read from the cache concurrently. You may lock down the entire cache every time an update (an insert) is performed.
- Style (10 points). Up to 10 points will be awarded for code that is readable, robust and well commented. Define subroutines where necessary to make the code more understandable. Also you should check the return codes of all library functions and handle errors as appropriate. It is NOT appropriate to use the "capital-letter" CSAPP wrappers around system calls. These wrappers will kill the entire proxy if there is an error on a single connection. The correct thing to do is to close that connection and terminate the connection thread. You will lose style points if your proxy simply terminates when it encounters an error during reading and writing.

8 Debugging and Testing Your Proxy

For this lab, you will not have any sample inputs or a driver program to test your implementation. You will have to come up with your own tests to help you debug your code and decide when you have a correct implementation. This is a valuable skill for programming in the real world, where exact operating conditions are rarely known and reference solutions are usually not available.

Below are some suggested means by which you can debug your proxy, to help you get started. Be sure to exercise all code paths and test a representative set of inputs, including base cases, typical cases, and edge cases.

- **Telnet** You can use telnet to send requests to any web server (and/or to your proxy). For initial debugging purposes, you can use print statements in your proxy to trace the flow of information between the proxy, clients and web servers. Run your proxy from a fish machine on an unused port, then connect to your proxy from another xterm window and make requests. The following output is a sample client trace where the proxy is running on tuna.ics.cs.cmu.edu on port 1217:

  ```
  unix> telnet tuna.ics.cs.cmu.edu 1217
  Trying 128.2.206.26...
  Connected to tuna.ics.cs.cmu.edu.
  Escape character is '^]'.
  GET http://www.yahoo.com HTTP/1.0

  HTTP/1.0 200 OK
  ...
  ```

- **Tiny Server** We have provided you with code to the CS:APP Tiny Web Server in the handout directory. A version is also available through the CS:APP student website

  ```
  http://csapp.cs.cmu.edu/public/students.html
  ```

  You may wish to modify the code in order to control any server behavior and/or to debug from the web server's end.

- **Web browsers** After your proxy is working with telnet, then you should test it with a real browser! It is very exciting to see your code serving content from a real server to a real browser.

  To setup Firefox to use a proxy, open the Settings window. In the Advanced pane, there is an option to "Configure how Firefox connects to the Internet". Click the Settings button and set **only** your HTTP proxy (using manual configuration). The server will be whatever fish machine your proxy is running on, and the port is the same as the one you passed to the proxy when you ran it.

- **Suggested Websites** Your proxy should be able to serve any web page. As a first step, however, we recommend that your proxy be able to serve the web pages listed below. The list is by no means exhaustive; we will test your proxy with additional websites during the grading period. The list is in approximately increasing order of how much your proxy will be stressed in serving the website.

- `http://www.cs.cmu.edu/ 213`
- `http://www.cs.cmu.edu/`
- `http://www.newyorktimes.com/`
- `http://www.cnn.com/`
- `http://www.youtube.com/`

9 Handin Instructions

Make sure you have included **all names and Andrew IDs of your group members** in the header comments of each file you hand in.

Hand in `proxylab.tar.gz`, which includes all files you need to compile your proxy with, by uploading them to Autolab.

Steps to tar up your lab:

```
1) make clean
1.5) rm core* 2) cd ..
3) tar czvf proxylab.tar.gz proxy-handout (i.e., tar the directory, not the
files.)
```

Important: Each group member must upload these code files to get credit. (Of course if you are in the same group, you will each be uploading the same code files, but this is OK).

You may submit your solution as many times as you wish up until the due date.

10 Resources and Hints

- Read Chapters 11 - 13 in your textbook. They contain useful information on system-level I/O, network programming, HTTP protocols, and concurrent programming.
- To get started, you can start with the echo server described in your text and then gradually add functionality. The Tiny Web server described in your text will also provide useful hints.
- **VERY IMPORTANT:** Thread-safety. Please be very careful when accessing shared variables from multiple threads. Normally, the cache should be the only shared object accessed by the different threads concurrently. Make sure you have enumerated race-conditions: for example, while one thread is utilizing a cache entry object, another thread might end up free'ing the same entry.

  At the same time, it is important to perform locking only at places where it is necessary because it can cause significant performance degradation.
- When using threads, keep in mind that `gethostbyname` is not thread-safe and should be protected using a lock and copy technique, as described in your text.

- **IMPORTANT:** When using threads to handle connection requests, you must run them as *detached*, not *joinable*, to avoid memory leaks that could crash the machine. To run a thread detached, add the line `pthread_detach(thread_id)` in the parent after calling `pthread_create()`.
- Use the RIO (Robust I/O) package described in your textbook for all I/O on sockets. Do not use standard I/O functions such as `fread` and `fwrite` on sockets. You will quickly run into problems if you do.
- **IMPORTANT:** Be aware that the default error handling wrappers (upper case first letter) supplied for the RIO routines in `csapp.c` are not appropriate for your proxies because the wrappers terminate whenever they encounter any kind of error. Your proxy should be more forgiving. Use the regular RIO routines (lower case first letter) for reading and writing. If you encounter an error reading or writing to a socket, simply close the socket. Here are some examples of the kinds of errors you can expect to encounter:
  - In certain cases, a client closing a connection prematurely results in the kernel delivering a SIGPIPE signal to the proxy. This is particularly the case with Internet Explorer. To prevent your proxy from crashing, you should ignore this signal by adding the following line early in your code:

    ```
    Signal(SIGPIPE, SIG_IGN);
    ```

  - In certain cases, writing to a socket whose connection has been closed prematurely results in the `write` system call returning a -1 and setting `errno` to EPIPE (Broken pipe). Your proxy should not terminate when a `write` elicits an EPIPE error. Simply close the socket, optionally print an error message, and continue.
  - Reading from a socket whose connection has been reset by the kernel at the other end (e.g., if the process that owned the connection dies) can cause the read to return a -1 with `errno` to ECONNRESET (Connection reset by peer). Again, your proxy should not die if it encounters this error.
- Here is how you should forward browser requests to servers so as to achieve the simplest and most predictable behavior from the servers:
  - Always send a "Host: <hostname>" request header to the server. Some servers (like csapp.cs.cmu.edu) require this header because they use virtual hosting. For example, the Host header for csapp.cs.cmu.edu would be "Host: csapp.cs.cmu.edu".
  - Forward all requests to servers as version HTTP/1.0, even if the original request was HTTP/1.1. Since HTTP/1.1 supports persistent connections by default, the server won't close the connection after it responds to an HTTP/1.1 request. If you forward the request as HTTP/1.0, you are asking the server to close the connection after it sends the response. Thus your proxy can reliably use EOF on the server connection to determine the end of the response.
  - Replace any Connection/Proxy-Connection: [connection-token] request headers with Connection/Proxy-Connection: close. Also remove any Keep-Alive: [timeout-interval] request headers. The reason for this is that some misbehaving servers will sometimes use persistent connections even

for HTTP/1.0 requests. You can force the server to close the connection after it has sent the response by sending the Connection: close header.

- Finally, try to keep your code as "object-oriented" and modular as possible. In other words, make sure each data structure is initialized, accessed, changed or freed by a small set of functions.