# EE324, Fall 2015
# Assignment 2: Web Proxy
# Assigned: Sep 21,
# Part I Due: October 5, 11:59PM
# Part II Due: October 14, 11:59PM
# Part III Due: October 24, 11:59PM

## 1 Introduction

A Web proxy is a program that acts as a middleman between a Web browser and an *end server*. Instead of contacting the end server directly to get a Web page, the browser contacts the proxy, which forwards the request on to the end server. When the end 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 an end server outside. The proxy may do translation on the page, for instance, to make it viewable on a Web-enabled cell phone. Proxies are also used as *anonymizers*. By stripping a request of all identifying information, a proxy can make the browser anonymous to the end 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 in response to future requests rather than going to the end server.

In this lab, you will write a concurrent Web proxy that logs requests. In the first part of the lab, you will write a simple sequential proxy that repeatedly waits for a request, forwards the request to the end server, and returns the result back to the browser, keeping a log of such requests in a disk file. This part will help you understand basics about network programming and the HTTP protocol. In the second part of the lab, you will upgrade your proxy so that it uses threads to deal with multiple clients concurrently. This part will give you some experience with concurrency and synchronization, which are crucial computer systems concepts. Your proxy should spawn a separate thread to deal with each request. This will give you an introduction to dealing with concurren cy, a crucial systems concept. Finally, you will turn your proxy into a proxy cache by adding a simple main memo ry cache of recently accessed web pages

## 2 Logistics

Unlike previous assignments, you can work individually or in a group of two on Part II. The lab is designed to be doable by a single person, so there is no penalty for work ing alone. You are, however, welcome to team up with another student if you wish. Part I and Part III must be done individually. If you work as a

group, you must email the instructor in advance to get a permission. We will let you share a SVN repository after Part I. The two persons must either to pair programming or show evidence of work (e.g., svn logs) for each individual.

# 3 Hand Out Instructions

**`proxylab-handout.tar` is available on the course webpage.**

Start by copying `proxylab-handout.tar` 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:

- `proxy.c`: This is the only file you will be modifying and handing in. It contains the bulk of the logic for your proxy.
- `csapp.c`: This is the file of the same name that is described in the Computer Systems: A Programmer's Perspective (CSAPP) textbook. It contains error handling wrappers and helper functions such as the RIO (Robust I/O) package, `open_clientfd`, and `open_listenfd`.
- `csapp.h`: This file contains a few manifest constants, type definitions, and prototypes for the functions in `csapp.c`.
- `Makefile`: Compiles and links `proxy.c` and `csapp.c` into the executable `proxy`.

Your `proxy.c` file may call any function in the `csapp.c` file. However, since you are only handing in a single `proxy.c` file, please don't modify the `csapp.c` file. If you want different versions of functions in in `csapp.c` (see the Hints section), write new functions in the `proxy.c` file. You do not have to use the functions provided. You may implement your own `readline()` function. We recommend this if you have done some socket programming previously.

# 4 Part I: Implementing a Sequential Web Proxy

In this part you will implement a sequential logging proxy. Your proxy should open a socket and listen for a connection request. When it receives a connection request, it should accept the connection, read the HTTP request, and parse it to determine the name of the end server. It should then open a connection to the end server, send it the request, receive the reply, and forward the reply to the browser if the request is not blocked.

Since your proxy is a middleman between client and end server, it will have elements of both. It will act as a server to the web browser, and as a client to the end 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 connecti on 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 browsers 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.

Please read some articles about HTTP requests to better understand the format of the HTTP requests your proxy should send to a server. Sec 12.5.3 in the CSAPP is a good starting point (text used in the Programming for EE class). **IMPORTANT:** Be sure to parse out the port number from the URL. We will be tes ting this. If the port is not explicitly stated, use the default port of port 80.

## Logging

Your proxy should keep track of all requests in a log file named `proxy.log`. Each log file entry should be a file of the form:

```
Date: browserIP URL size
```

where `browserIP` is the IP address of the browser, `URL` is the URL asked for, `size` is the size in bytes of the object that was returned. For instance:

```
Sun 27 Oct 2002 02:51:02 EST: 128.2.111.38 http://www.cs.cmu.edu/ 34314
```

Note that `size` is essentially the number of bytes received from the end server, from the time the connection is opened to the time it is closed. Only requests that are met by a response from an end server should be logged. We have provided the function `format_log_entry` in `csapp.c` to create a log entry in the required format.

## Port Numbers

You proxy should listen for its connection requests on the port number passed in on the command line:

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

You may use any port number $p$, where $1024 \leq p \leq 65536$, and where $p$ is not currently being used by any other system or user services (including other students' proxies). See `/etc/services` for a list of the port numbers reserved by other system services.

# 5 Part II: Dealing with multiple requests concurrently

Real proxies do not process requests sequentially. They deal with multiple requests concurrently. Once you have a working sequential logging proxy, you should alter it to handle multiple requests concurrently. The simplest approach is to create a new thread to deal with each new connection request that arrives.

With this approach, it is possible for multiple peer threads to access the log file concurrently. Thus, you will need to use a semaphore to synchronize access to the file such that only one peer thread can modify it at a time. If you do not synchronize the threads, the log file might be corrupted. For instance, one line in the file might begin in the middle of another.

# 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 cl ients, you should save the object in memory as you transmit it back to the client. This way if anoth er client requests the same object at some later time, your proxy neednt connect to the server again. It can simply resend the cached object. Obviously, if your proxy stored every object that was ever re quested, it would require an unlimited amount of memory. To avoid this (and to simplify our testing) we will establish a maximum cache size of

```
MAX CACHE SIZE = 5MB
```

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 timestamp. 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 gian t one, therefore we will establish a maximum object size of

```
MAX OBJECT SIZE = 512KB
```

You should stop trying to cache an object once its size grows a bove 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 `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

```
MAX CACHE SIZE+ T  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

Each group will be evaluated on the basis of a demo to your instructors. See the course Web page for instructions on how to sign up for your demos.

- Basic proxy functionality (30 points). Your sequential proxy should correctly accept connections, forward the requests to the end server, and pass the response back to the browser, making a log entry for each request. Your program should be able to proxy browser requests to the following Web sites and correctly log the requests:
  - `http://www.google.com`
  - `http://ina.kaist.ac.kr`
  - `http://www.yahoo.com`
- Handling concurrent requests (30 points).

  Your proxy should be able to handle multiple concurrent connections. We will determine this using the following test: (1) Open a connection to your proxy using `telnet`, and then leave it open without typing in any data. (2) Use a Web browser (pointed at your proxy) to request content from some end server.

  Furthermore, your proxy should be thread-safe, protecting all updates of the log file and protecting calls to any thread unsafe functions such as `gethostbyaddr`. We will determine this by inspection during the demo.
- 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 w hen possible. Your cache must adhere to the cache size limit and the maximum object size lim it, and must not insert duplicate entries into the cache.
  - 7.5 points will be given for a proper implementation of the sp ecified least-recently-used (LRU) eviction policy.
  - 7.5 points will be given for proper use of locks. Your cache mu st be free of race conditions, deadlocks, and excessive locking. Excessive locking inclu des keeping the cache locked across a network system call or not allowing multiple connection thr eads 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 and well commented. Your code should begin with a comment block that describes in a general way how your proxy works. Furthermore, each function should have a comment block describing what that function does. Furthermore, your threads should run detached, and your code should not have any memory leaks. We will determine this by inspection during the demo.

# 8 Debugging and Testing

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 c ode 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 ava ilable. Below are some suggested means by which you can debug your pro xy, to help you get started. Be sure to exercise all code paths and test a representative set of inpu ts, including base cases, typical cases, and edge cases.

- **Telnet:** You can use telnet to send requests to any web server (and/or t o your proxy). For initial debugging purposes, you can use print statements in your pro xy to trace the flow of information between the proxy, clients and web servers. Run your proxy fr om a shark 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 ina.kaist.ac.kr on port 1217:

```
unix> telnet ina.kaist.ac.kr 1217
Trying 143.248.56.236...
Connected to ina.kaist.ac.kr
Escape character is ].
GET http://www.yahoo.com HTTP/1.0
HTTP/1.0 200 OK
...
```

- **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 rea l server to a real browser. Please test prior to demo time that your proxy works Mozilla Firefox; if y ou can test with other browsers, you are encouraged to do so. To setup Firefox to use a proxy, open the Settings window. In t he Advanced pane, there is an option to Configure how Firefox connects to the Internet. Click the S ettings button and set only your HTTP proxy (using manual configuration). The server will be whate ver shark machine your proxy is running on, and the port is the same as the one you passed to the proxy wh en you ran it.

# 9 Hints

- The best way to get going on your proxy is to start with the basic echo server and then gradually add functionality that turns the server into a proxy.
- Initially, you should debug your proxy using telnet as the client.
- Later, 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.
- **VERY IMPORTANT:** Thread-safety. Please be very careful when accessing shared variables from multiple threads. Normally, the cache should be the only sha red 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 freeing 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.

- We have provided you with a template of two helper routines: `parse_uri`, which extracts the hostname, path, and port components from a URI, and `format_log_entry`, which constructs an entry for the log file in the proper format. Start by filling this in.
- Be careful about memory leaks. When the processing for an HTTP request fails for any reason, the thread must close all open socket descriptors and free all memory resources before terminating.
- You will find it very useful to assign each thread a small unique integer ID (such as the current request number) and then pass this ID as one of the arguments to the thread routine. If you display this ID in each of your debugging output statements, then you can accurately track the activity of each thread.
- To avoid a potentially fatal memory leak, your threads should run as detached, not joinable.
- Since the log file is being written to by multiple threads, you must protect it with mutual exclusion semaphores whenever you write to it.
- Be very careful about calling thread-unsafe functions such as `inet_ntoa`, `gethostbyname`, and `gethostbyaddr` inside a thread. In particular, the `open_clientfd` function in `csapp.c` is thread-unsafe because it calls `gethostbyaddr`, a Class-3 thread unsafe function. You will need to write a thread-safe version of `open_clientfd`, called `open_clientfd_ts`, that uses the lock-and-copy technique when it calls `gethostbyaddr`.
- You may use the RIO (Robust I/O) package for all I/O on sockets or the socket API. Do not use standard I/O (e.g., fread) on sockets. You will quickly run into problems if you do. However, standard I/O calls such as `fopen` and `fwrite` are fine for I/O on the log file.
- The `Rio_readn`, `Rio_readlineb`, and `Rio_writen` error checking wrappers in `csapp.c` are not appropriate for a realistic proxy because they terminate the process when they encounter an error. Instead, you should write new wrappers called `Rio_readn_w`, `Rio_readlineb_w`, and `Rio_writen_w` that simply return after printing a warning message when I/O fails. When either of the read wrappers detects an error, it should return 0, as though it encountered EOF on the socket.
- Reads and writes can fail for a variety of reasons. The most common read failure is an `errno = ECONNRESET` error caused by reading from a connection that has already been closed by the peer on the other end, typically an overloaded end server. The most common write failure is an `errno = EPIPE` error caused by writing to a connection that has been closed by its peer on the other end. This can occur for example, when a user hits their browser's Stop button during a long transfer.
- Writing to connection that has been closed by the peer first time elicits an error with errno set to EPIPE. Writing to such a connection a second time elicits a SIGPIPE signal whose default action is to terminate the process. To keep your proxy from crashing you can use the SIG_IGN argument to the signal function to explicitly ignore these SIGPIPE signals
- 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 wont 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.

## 10 Handin Instructions

Use your SVN repository. Handin directories are:

- Part I: `tags/PA2/Part1`
- Part II: `tags/PA2/Part2`
- Part III: `tags/PA2/Part3`