M1522.000800 System Programming, Fall 2016
Proxy Lab: Echo Proxy
Assigned: Tue., Nov.15, Due: Mon., Nov.28, 23:59

Introduction

An echo proxy is a program that acts as a middleman between an echo client and an *echo server*. Instead of contacting the echo server directly, the client contacts the proxy, which forwards the request on to the end of the echo server. When the echo server replies to the proxy, the proxy sends the reply to the client.

Proxies are used for many purposes in the real world. Sometimes proxies are used in firewalls, such that the proxy is the only way for a web browser inside the firewall to contact a web server outside the firewall. The proxy may do translation on the page, for instance, to make a 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, for example, 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, there are two parts: a simple sequential proxy and a concurrent echo proxy.

- In the first part of the lab, you will write a simple sequential proxy that repeatedly waits for a requests, forwards the request to the echo server, and returns the result back to the client, keeping *a log of such requests in a file in disk*. This part will give you some experience with network programming.
- 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 system concepts.

Logistics

The only *hand-in* will be done via email. Any clarifications and revisions to the assignment will be posted on the course Web page. So please check eTL constantly.

Hand Out Instructions

You can obtain the tar file for this lab from eTL under the section Boards → Lab Assignments.

Download a copy the file `proxylab-handout.tgz` to a directory of your choice (the *lab directory*) in which you plan to do your work. After uncompressing it a number of files will be unpacked in the directory:

- `proxy.c`: This is the **only** file you will be modifying and handing in.
- `csapp.c`: This is the file of the same name as the one described in the textbook. It contains error handling wrappers and helper functions such as the RIO (Robust I/O) package (textbook[1] 10.5), `open_clientfd` (textbook 12.4.4), and `open_listenfd` (textbook 12.4.7).
- `csapp.h`: This file contains a few manifest constants, type definitions, and prototypes for the functions in `csapp.c`.
- `echoclient.c`: This is the echo client.
- `echoserver.c`: This is the echo server.
- `Makefile`: Combines 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 **do NOT** modify the `csapp.c` file. If you want different versions of the functions found in `csapp.c`, write a new function in the `proxy.c` file.

Part I: Implementing a Sequential Echo Proxy

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

Since your proxy is a middleman between the echo client and the echo server, it will have elements of both. It will act as a server to the echo client, and as a client to the echo server. Thus you will get experience with both client and server programming.

Logging

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

```
date: clientIP clientPort size echostring
```

[1] All textbook references are based on the second edition of the textbook

Where `clientIP` is the IP address of the echo client, `clientPort` is the port number of the echo client, `size` is the size in bytes of the echo string that was returned from the echo server, and `echostring` is the echo string returned from the server.

```
Sun 13  Nov 2016 12:51:02 KST: 128.2.111.38 38451 11 HELLOWORD!
```

Note that `size` is essentially the number of bytes received from the end server, from the time the connection opened to the time it was closed. **Only requests that are met by a response from an echo server should be logged**.

Port Numbers

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

```
unix> ./proxy 1357
```

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. See `/etc/services` for a list of the port numbers reserved by other services on your system.

Part II: Dealing with Multiple Requests Concurrently

Real proxies do not process requests sequentially. They deal with multiples 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 (textbook 13.3.8)

With this approach, it is not 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.

Evaluation

- (35 points) Basic proxy functionality. Your sequential proxy should correctly accept connections, forward the requests to the end echo server, and pass the response back to the client, making a log entry for each request.
- (35 points) Handling concurrent requests. Your proxy should be able to handle multiple concurrent connections. We will determine this by using the following test: (1) Open a connection to your proxy using `telnet`, and then leave it open without typing any data. (2) Use an echo client (pointed at your proxy) to send a request to some echo 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.

- (10 points) Code style. You can get up to ten points for well written and commented code. Your code should begin with a short block describing how your proxy works. Also, each function should have a comment block describing what that functions does. Moreover, your threads should run detached, and your code should not have any memory leaks. We will test this using by inspection.
- (20 points) Report. In your report you should describe your implementation. Also, you should add a section regarding difficulties and suggestions for this lab, similar to the previous lab (shell lab).

Hand-in Instructions

First, make sure that you have filled in your student ID and name in both files `proxy.c` and `Makefile`. Then, assuming your have named your report file `report.pdf` and have placed it in the hand out directory you can run `make handin` to generate a tar file which you can then send via email to the TA (sysprog@csap.snu.ac.kr).

Use the following subject line format when sending an email: `[proxylab] Name StudentID`, ex: `[proxylab] TA 2016-123456`. If you fail to use this format you might be subject of a ten point deduction of your final score for this lab.

Hints

- The best way to get going on your proxy is to start with the echo server and then gradually add functionality that turns the server into a proxy.
- Be careful about memory leaks. When the processing for an echo 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 (textbook 13.3.6).
- Since the log file is being written to by multiple threads, you must protect it with mutual exclusion semaphores whenever you write to it (textbook 13.5.2 and 13.5.3).
- 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 (textbook 13.7.1). You will need to write a thread-safe version of open `clientfd`, called open `clientfd_ts`, that uses the lock-and-copy technique (textbook 13.7.1) when it `callsgethostbyaddr`

- Use the RIO (Robust I/O) package (textbook 11.4) for all I/O on sockets. Do not use standard I/O 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 `Rioreadn`, `Rioreadlineb`, and `Riowriten` 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.

- 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 (textbook 8.5.3) to explicitly ignore these `SIGPIPE` signals