M1522.000800 System Programming, Spring 2015

Proxy Lab: Echo Proxy
Assigned: Wednesday May. 27, Due: Tuesday June. 9, 11:59PM

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 echo server. When the echo server replies to the proxy, the proxy sends the reply on 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 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, 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 request, forwards the request to the echo server, and returns the result back to the client, keeping *a log of such requests in a disk file*. 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 systems concepts.

Hand Out Instructions

After downloading the `proxylab-handout.tgz` file from eTL, start by copying `proxylab-handout.tgz` to a directory in which you plan to do your work. Then give the command `tar xvfz proxylab-handout.tgz`. 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.

- `csapp.c`: This is the file of the same name that is described in the textbook. It contains error handling wrappers and helper functions such as the RIO (Robust I/O) package (textbook 11.4), `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`: 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`, write new functions 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 listen 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 echo client and 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 requests in a log file named `proxy.log`. Each log file entry should be a file of the form:

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

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 27 Nov 2013 02:51:02 KST: 128.2.111.38 38421 11 HELLOWORLD!
```

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 echo server should be logged.

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.

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 (textbook 13.3.8).

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.

Evaluation

- Basic proxy functionality (30 points). 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. Your program should be able to proxy echo client requests to an echo server sites and correctly log the requests.

- 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 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.

- 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. Fur- thermore, your threads should run detached, and your code should not have any memory leaks. We will determine this by inspection.

Handin Instructions

Make sure that you have filled in your student ID and name in the file proxy.c and submit the file by email to the TA(sysprog@csap.snu.ac.kr).

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 calls `gethostbyaddr`.

- 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 `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.

- 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