SE101 Spring 2020
Lab Assignment 9: Web Proxy
Assigned: May 18, Due: June 1, 11:59PM

Nian Liu (nianliu@sjtu.edu.cn) is responsible for lab.

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.

Logistics

As always, you should work on your own for this lab. Any clarification and revision to the assignment will be posted on the course Web page.

Hand Out Instructions

The lab is distributed to you in lab10 directory in svn. The handout contains several files:

- `proxy.c`: This is the skeleton, though it's very simple. 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 CS:APP textbook. It contains error handling wrappers and helper functions such as the RIO (Robust I/O) package (CS:APP 10.5), `open_clientfd` (CS:APP 11.4.8), and `open_listenfd` (CS:APP 11.4.8).
- `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`.
- `grade.py`: The grade script of this lab, written in Python. Further information will be given in evalution part
- `client`, `server`: The helper that grade.py will use. Don't delete or modify them.

Your `proxy.c` file may call any function in the `csapp.c` file. Feel free to modify any thing include Makefile and grade.py. However, When we grade your lab, we will replace your grade.py with standard version, so you can't cheat in the grade script.

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.

Logging

Your proxy should keep track of all requests by printing log on screen. Each log entry should be a line 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:

```
Thu 16 May 2019 16:58:24 UTC: 127.0.0.1 http://127.0.0.1:45659/ 581
```

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` 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.

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 (CSAPP 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

We grade your lab using `grade.py`. Running this Python script requires you to install a Python library called pexpect. In most cases, `sudo apt-get update` then `sudo apt-get install python-pexpect` will do it for you. You are allowed to inspect or modify the grade script. If you found any bug or improvement, you can discuss with TA about it.

- Basic proxy functionality (35 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. We test this part by issue proxy request from `client` and try to access `server` via your proxy.

  To get a full mark, you need to handle various requests correctly, including abnormal ones.

- Handling concurrent requests (30 points).

  Your proxy should be able to handle multiple concurrent connections. We test it in two cases:

  - Download a huge file and access lots of small files simultaneously. In the huge access, server will send the content part by part, you should redirect the received parts to client immediately, instead of buffering them content until all parts are received. If you buffer them, the user will think his downloading is stuck.
  - Access lots of small files simultaneously. In this test, the small accesses are no longer sequential. We randomly choose one of the HTTP connection and send several bytes to simulate segmentation and out-of-order.

  As `printf` is not thread-safe, you are also asked to add an lock to protect it. We will check this requirement by inspecting your code.

Hints

- The best way to get going on your proxy is to start with the basic echo server (CS:APP 11.4.9) and then gradually add functionality that turns the server into a proxy.
- You should debug your proxy using telnet as the client (CS:APP 11.5.3), and target your access to some HTTP-only website, such as `http://www.sjtu.edu.cn/`. Notice that you may get `HTTP/1.1 301 Moved permanently` since most websites have already moved to more secure `https`.
- Since we want you to focus on network programming issues for this lab, we have provided you with two additional 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.
- 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 (CS:APP 12.3.6).
- Since the log is being printed by multiple threads, you must protect it with mutual exclusion semaphores whenever you use `printf` to it (CS:APP 12.5.2 and 12.5.3). You do not have to protect error printing.
- Use the RIO (Robust I/O) package (CS:APP 10.5) 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. 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 (CS:APP 8.5.3) to explicitly ignore these SIGPIPE signals
- Wireshark and strace is helpful to monitoring what did your program do on network. You can Google or Baidu the tutorial of them.

Handin Instructions

Simply commit your code to svn as you did in previous lab. Don't forget any modification you did. If your code fails to compile on TA's computer, you will get zero.