

CSC 2405: Concurrent Web Proxy

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 assignment, you will write a concurrent Web proxy that logs requests. In the first part of the assignment, 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. This part will help you understand basics about network programming and the HTTP protocol.

In the second part of the assignment, you will extend your proxy from the first part to keep a log of requests in a disk file. This part will help you learn how to manipulate files in C.

In the third part of the assignment, you will upgrade your proxy from the first part 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

In your `csc2405` directory, copy `proxy-handout.tar` from `/tmp`:

```
cp /tmp/proxy-handout.tar ~/csc2405
```

Next use the command

```
tar xvf proxy-handout.tar
```

This will cause a number of files to be unpacked in the directory `proxy-handout`:

- `proxy.c`: This is the only file you will be modifying and handing in. It contains a few helper functions for your proxy.
- `echoservert.c`: A complete concurrent echo server example that uses threads.

- `csapp.c`: This file 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 a C source file and `csapp.c` into the corresponding executable.

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 need different versions of functions in `csapp.c` (as suggested in the Hints section), write the new functions in the `proxy.c` file.

Testing the Echo Server

Build the `echoservert` executable on tanner by typing in `make echoservert`, then test the server using `telnet` as follows. Pick a random 5-digit integer `N` no greater than 64000 and pass it to the server as an argument:

```
./echoservert N
```

Open a second terminal window on tanner and invoke `telnet` at the shell prompt, passing the port number `N` as a second argument:

```
telnet tanner N
```

The `telnet` application opens a TCP connection to port `N` on tanner. Now anything that you type into the `telnet` window will be sent over this connection and echoed back to you by the server.

Part I: Implementing a Sequential Web Proxy

In this part you will implement a sequential 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 the request (altered as described in subsequent sections), receive the reply, and forward the reply (unaltered) to the browser, if the request is not blocked. Check the Hints section for details on error handling. You know that your program works if the browser displays the information correctly.

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. Chapter 12 in our textbook is a very good reference material – use it as needed.

HTTP Requests to Servers

HTTP requests are of many types, but we will only concern ourselves with the GET request in this assignment. Suppose that you type the URL `http://www.google.com/index.html` into your Web browser. The Web browser will open a TCP connection to `www.google.com` on port 80 and send the request (followed by a number of lines not shown here):

```
GET /index.html HTTP/1.1\r\n
```

The `\r\n` sequence represents the ASCII character 13 (`'\r'`) followed by ASCII character 10 (`'\n'`). In response, the web server will locate the file called `/index.html` and transmit its contents to the web client. Try out the following:

1. Telnet to your favorite Web server, for example

```
telnet www.google.com 80
```

The telnet application opens a TCP connection to port 80 (default HTTP port) on `www.google.com`. Anything that you type in next will be sent over this connection.

2. Type in an HTTP GET request:

```
GET /index.html HTTP/1.1
```

and hit Enter TWICE. In this way you send this minimal (but complete) GET request to the HTTP server. It is possible that you won't see your GET request on the screen as you type it in. If you make a typing error, you will get a "Bad Request" response from the server (in that case, start over).

3. Look at the response send by the HTTP server! It should be identical to the text you see when selecting "View/Source" on the top bar menu of your browser.

HTTP Requests to Proxies

Web browsers can be configured to use a proxy server (see the Hints section) instead of sending requests directly to Web servers. In this case, when you try to load a page such as

```
http://www.google.com/index.html
```

the browser contacts the designated proxy server and asks it for the page as follows:

```
GET http://www.google.com/index.html HTTP/1.1
```

Notice the difference: instead of asking just for the resource `/index.html`, the web browser asks for the full URL `http://www.google.com/index.html`. This is to give the proxy enough information to fetch the file `/index.html` from `www.yahoo.com` on behalf of the web browser and deliver it as a result, or do whatever it wants to do.

To fetch the file, the proxy extracts the file name from the GET line and forwards the request

```
GET /index.html HTTP/1.1\r\n
```

along with subsequent lines, as they came from the browser; the last line is always empty (`\r\n`).

Port Numbers

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

```
unix> ./proxy 12345
```

A unique port number has been assigned to you – please check the class website to find out the port number your server is supposed to use. This is to ensure that your port number is not currently being used by other students or system services (see `/etc/services` for a list of the port numbers reserved by system services).

PART II: Logging Requests

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

```
Sun 16 Feb 2009 02:51:02 EST: 128.2.111.38 http://www.xxx.com/ 34056
```

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. See

```
http://www.csc.villanova.edu/~mdamian/C/c-files.htm
```

for instructions on manipulating files in C.

Part III: Implementing Concurrency with Threads

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. Use `echoserv.c` as a guiding example.

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. Check the Hints section for details on thread-safe functions.

Evaluation

Each student will be evaluated on the basis of a demo to your instructor. Evaluation criteria include:

- Basic proxy functionality. Your sequential proxy should correctly accept connections, forward the requests to the end server, and pass the response back to the browser. Your program should be able to proxy browser requests to the following Web sites (and, for Part II, correctly log the requests):

- `http://www.yahoo.com`
- `http://www.aol.com`

- Handling concurrent requests using threads.
- Style. 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.

Hints

- The best way to get going on your proxy is to start with the basic echo server from your handout 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.
- Since the focus of this assignment is on concurrency and network programming, we provide you with two helper routines: `parse_url`, which extracts the hostname, path, and port components from a URL, 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.
- 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. Such functions compute the result in a `static` structure and return a pointer to that structure, making it possible for the result used by one thread to be silently overwritten by another thread. In particular, the `open_clientfd` function in `csapp.c` is thread-unsafe because it calls `gethostbyaddr`. You will need to write a thread-safe version of `open_clientfd`, called `open_clientfd_ts`. This function should lock a mutex, call the thread-unsafe function, copy the result to local memory, then unlocks the mutex.
- Use the RIO (Robust I/O) package 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. When a system call fails, it sets the external system variable `errno`. 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

Handin Instructions

- Remove any extraneous print statements.
- Make sure that you have included your identifying information in `proxy.c`.
- Hand in a printout of your code and a README file that lists any known debugs or deviations from the specifications.
- Schedule a demo time with me.