CS 324, Winter 2021
Proxy Lab: Writing a Caching Web Proxy - Threadpool
Due: Monday, Mar. 22, 11:59 PM

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 useful for many purposes. Sometimes proxies are used in firewalls, so that browsers behind a firewall can only contact a server beyond the firewall via the proxy. Proxies can also act as anonymizers: by stripping requests of all identifying information, a proxy can make the browser anonymous to Web servers. Proxies can even be used to cache web objects by storing local copies of objects from servers then responding to future requests by reading them out of its cache rather than by communicating again with remote servers.

In this lab, you will write a simple HTTP proxy objects. For the first part of the lab, you will set up the proxy to accept incoming connections, read and parse requests, forward requests to web servers, read the servers' responses, and forward those responses to the corresponding clients. This first part will involve learning about basic HTTP operation and how to use sockets to write programs that communicate over network connections. In the second part, you will upgrade your proxy to deal with multiple concurrent connections using a simple, thread-based model. This will introduce you to dealing with concurrency, a crucial systems concept. In the the third part, you will modify your concurrency approach to use a threadpool. In a future lab, you will modify your proxy to use I/O multiplexing.

2 Logistics

This is an individual project.

3 Handout instructions

The files for the lab will made available from a link in the "assignment" page for the lab on the course site on Learning Suite, as a single archive file, `proxylab1-handout.tar`.

Start by copying `proxylab1-handout.tar` to a protected Linux directory in which you plan to do your work. Then give the command

```
linux> tar xvf proxylab-handout.tar
```

This will generate a handout directory called `proxylab1-handout`. The README file describes the various files.

4 Part I: Implementing a sequential web proxy

The first step is implementing a basic sequential proxy that handles HTTP/1.0 GET requests. Other requests type, such as POST, are strictly optional.

When started, your proxy should listen for incoming connections on a port whose number will be specified on the command line. Once a connection is established, your proxy should read the entirety of the request from the client and parse the request. It should determine whether the client has sent a valid HTTP request; if so, it can then establish its own connection to the appropriate web server then request the object the client specified. Finally, your proxy should read the server's response and forward it to the client.

4.1 HTTP/1.0 GET requests

When an end user enters a URL such as `http://www-notls.imaal.byu.edu/index.html` into the address bar of a web browser, the browser will send an HTTP request to the proxy that begins with a line that might resemble the following:

```
GET http://www-notls.imaal.byu.edu/index.html HTTP/1.1
```

In that case, the proxy should parse the request into at least the following fields: the hostname, `www-notls.imaal.byu.e` and the path or query and everything following it, `/index.html`. That way, the proxy can determine that it should open a connection to `www-notls.imaal.byu.edu` and send an HTTP request of its own starting with a line of the following form:

```
GET /index.html HTTP/1.0
```

Note that all lines in an HTTP request end with a carriage return, `'\r'`, followed by a newline, `'\n'`. Also important is that every HTTP request is terminated by an empty line: `"\r\n"`. Thus, the entire request ends in the following sequence: `"\r\n\r\n"`.

You should notice in the above example that the web browser's request line ends with `HTTP/1.1`, while the proxy's request line ends with `HTTP/1.0`. Modern web browsers will generate HTTP/1.1 requests, but your proxy should handle them and forward them as HTTP/1.0 requests.

It is important to consider that HTTP requests, even just the subset of HTTP/1.0 GET requests, can be incredibly complicated. The textbook describes certain details of HTTP transactions, but you should refer

to RFC 1945 for the complete HTTP/1.0 specification. Ideally your HTTP request parser will be fully robust according to the relevant sections of RFC 1945, except for one detail: while the specification allows for multiline request fields, your proxy is not required to properly handle them. Of course, your proxy should never prematurely abort due to a malformed request.

4.2 Request headers

The important request headers for this lab are the `Host`, `User-Agent`, `Connection`, and `Proxy-Connection` headers:

- Always send a `Host` header. While this behavior is technically not sanctioned by the HTTP/1.0 specification, it is necessary to coax sensible responses out of certain Web servers, especially those that use virtual hosting.

  The `Host` header describes the hostname of the end server. For example, to access `http://www-notls.imaal.byu.edu/index.html`, your proxy would send the following header:

  ```
  Host: www-notls.imaal.byu.edu
  ```

  It is possible that web browsers will attach their own `Host` headers to their HTTP requests. The value of the `Host` header should match; in the case they do not, your proxy can (for simplicity) use the hostname that is embedded in the URL.

- You *may* choose to always send the following `User-Agent` header:

  ```
  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0)
      Gecko/20100101 Firefox/68.0
  ```

  The header is provided on two separate lines because it does not fit as a single line in the writeup, but your proxy should send the header as a single line.

  The `User-Agent` header identifies the client (in terms of parameters such as the operating system and browser), and web servers often use the identifying information to manipulate the content they serve. Sending this particular User-Agent: string may improve, in content and diversity, the material that you get back during simple telnet-style testing.

- Always send the following `Connection` header:

  ```
  Connection: close
  ```

- Always send the following `Proxy-Connection` header:

  ```
  Proxy-Connection: close
  ```

The `Connection` and `Proxy-Connection` headers are used to specify whether a connection will be kept alive after the first request/response exchange is completed. While it is not the most efficient use of resources, it is perfectly acceptable to have your proxy open a new connection for each request—and it will simplify your proxy implementation. Specifying `close` as the value of these headers alerts web servers that your proxy intends to close connections after the first request/response exchange.

For your convenience, the values of the described `User-Agent` header is provided to you as a string constant in `proxy.c`.

Finally, if a browser sends any additional request headers as part of an HTTP request, your proxy should forward them unchanged.

4.3 Port numbers

There are two significant classes of port numbers for this lab: HTTP request ports and your proxy's listening port.

The HTTP request port is an optional field in the URL of an HTTP request. That is, the URL may be of the form, `http://www-notls.imaal.byu.edu:8080/index.html`, in which case your proxy should connect to the host `www-notls.imaal.byu.edu` on port 8080 instead of the default HTTP port, which is port 80. Your proxy must properly function whether or not the port number is included in the URL.

The listening port is the port on which your proxy should listen for incoming connections. Your proxy should accept a command line argument specifying the listening port number for your proxy. For example, with the following command, your proxy should listen for connections on port 15213:

```
linux> ./proxy 15213
```

You may select any non-privileged listening port (greater than 1,023 and less than 65,536) as long as it is not used by other processes. Since each proxy must use a unique listening port and many people will simultaneously be working on each machine, the script `port-for-user.pl` is provided to help you pick your own personal port number. Use it to generate port number based on your user ID:

```
linux> ./port-for-user.pl droh
droh: 45806
```

The port, p, returned by `port-for-user.pl` is always an even number. So if you need an additional port number, say for the `Tiny server`, you can safely use ports p and $p + 1$.

Please don't pick your own random port. If you do, you run the risk of interfering with another user.

5 Part II: Dealing with multiple concurrent requests

Once you have a working sequential proxy, you should alter it to simultaneously handle multiple requests. The simplest way to implement a concurrent server is to spawn a new thread to handle each new connection request.

Note that with this particular thread paradigm, you should run your threads in *detached* mode to avoid memory leaks. When a new thread is spawned, you can put it in detached mode by calling within the thread routine itself:

```
pthread_detach(pthread_self());
```

5.1 Robustness

As always, you must deliver a program that is robust to errors and even malformed or malicious input. Servers are typically long-running processes, and web proxies are no exception. Think carefully about how long-running processes should react to different types of errors. For many kinds of errors, it is certainly inappropriate for your proxy to immediately exit.

Robustness implies other requirements as well, including invulnerability to error cases like segmentation faults and a lack of memory leaks and file descriptor leaks.

6 Part III: Threadpool

For the final part of the lab, you will change your proxy server to use a pool of threads instead of launching a new thread for each request. This is described in Section 12.5.5 of your textbook.

Launching a thread for each request is costly and could use all your resources. A better approach is to launch several threads that wait for a connection (socket) to be added to a shared queue. The threads will wait on semaphores in a producer/consumer fashion to dequeue a socket from the queue and handle that connection. In this way, you limit the number of resources your server uses.

Note: Access to the queue will need to be protected with semaphores so the operations are thread safe. The queues should use producer/consumer style signalling with the semaphores.

7 Testing and debugging

Besides the simple autograder, you will not have any sample inputs or a test program to test your implementation. You will have to come up with your own tests and perhaps even your own testing harness to help you debug your code and decide when you have a correct implementation. This is a valuable skill in the real world, where exact operating conditions are rarely known and reference solutions are often unavailable.

Fortunately there are many tools you can use to debug and test your proxy. Be sure to exercise all code paths and test a representative set of inputs, including base cases, typical cases, and edge cases.

7.1 telnet

As described in your textbook (11.5.3), you can use `telnet` to open a connection to your proxy and send it HTTP requests.

7.2 curl

You can use `curl` to generate HTTP requests to any server, including your own proxy. It is an extremely useful debugging tool. For example, if your proxy and Tiny are both running on the local machine, Tiny is listening on port 15213, and proxy is listening on port 15214, then you can request a page from Tiny via your proxy using the following `curl` command:

```
linux> curl -v --proxy http://localhost:15214 http://localhost:15213/home.html
* About to connect() to proxy localhost port 15214 (#0)
*   Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 15214 (#0)
> GET http://localhost:15213/home.html HTTP/1.1
> User-Agent: curl/7.19.7 (x86_64-redhat-linux-gnu)...
> Host: localhost:15213
> Accept: */*
> Proxy-Connection: Keep-Alive
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: Tiny Web Server
< Content-length: 120
< Content-type: text/html
<
<html>
<head><title>test</title></head>
<body>
<img align="middle" src="godzilla.gif">
Dave O'Hallaron
</body>
</html>
* Closing connection #0
```

7.3 slow-client.py

`slow-client.py` is a tool included with your handout, with usage similar to that of `curl`. However, it has an additional functionality to test that your proxy is actually reading bytes as a byte *stream*. It does this by sending only chunks of an HTTP request at a time—not the entire request. By adding the option `--sleep-between-send 1`, it sleeps 1 second between the transmission of each chunk. For example, to run a test equivalent to the `curl` command above, using `slow-client.py` with a `--sleep-between-send` of 1 second, use the following:

```
linux> ./slow-client.py --sleep-between-send 1 --proxy http://localhost:15214 http://localh
```

7.4 `netcat`

`netcat`, also known as `nc`, is a versatile network utility. You can use `netcat` just like `telnet`, to open connections to servers. Hence, imagining that your proxy were running on `catshark` using port 12345 you can do something like the following to manually test your proxy:

```
sh> nc www-notls.imaal.byu.edu 12345
GET http://www-notls.imaal.byu.edu/index.html HTTP/1.0

HTTP/1.1 200 OK
...
```

In addition to being able to connect to Web servers, `netcat` can also operate as a server itself. With the following command, you can run `netcat` as a server listening on port 12345:

```
sh> nc -l 12345
```

Once you have set up a `netcat` server, you can generate a request to a phony object on it through your proxy, and you will be able to inspect the exact request that your proxy sent to `netcat`.

7.5 Web browsers

Eventually you should test your proxy using the *most recent version* of Mozilla Firefox. Visiting `About Firefox` will automatically update your browser to the most recent version.

To configure Firefox to work with a proxy, visit

```
Preferences>Advanced>Network>Settings
```

It will be very exciting to see your proxy working through a real Web browser. Although the functionality of your proxy will be limited, you will notice that you are able to browse the vast majority of websites through your proxy.

7.6 Autograding

Your handout materials include an autograder, called `driver.py`, which will evaluate your proxy server. Use the following command line to run the driver for this lab:

```
linux> ./driver.py -b 50 -c 45 threadpool
```

This will provide you a grade and feedback for your assignment. Note that maximum possible points that the script will output is 95. The remaining five points are for no warnings with compilation.

Note that the driver can run with different options to help you troubleshoot. For example:

- **Basic Only.** If you are just testing the basic functionality of your proxy (i.e., without concurrency), just use the `-b` option.

```
linux> ./driver.py -b 50 threadpool
```

- **Increased Verbosity.** If you want more output, including descriptions of each test that is being performed, use `-v`:

```
linux> ./driver.py -v -b 50 -c 45 threadpool
```

  For even more output, including the commands that are being executed, use `-vv`:

```
linux> ./driver.py -vv -b 50 -c 45 threadpool
```

- **Proxy Output.** If you want the output of your proxy to go to a file, which you can inspect either real-time or after-the-fact, use the `-p` option. Use `-p -` for your proxy output to go to standard output.

```
linux> ./driver.py -t myproxyoutput.txt -b 50 -c 45 threadpool
```

- **Downloaded Files.** By default, the downloaded files are normally saved to a temporary directory, which is deleted after the tests finish. If you want to keep these files to inspect them, use the `-k` option.

```
linux> ./driver.py -k -b 50 -c 45 threadpool
```

Any of the above options can be used together.

8 Resources, Hints, and Example Code

8.1 Resources

- Chapters 10-12 of the textbook contains useful information on system-level I/O, network programming, HTTP protocols, and concurrent programming.
- RFC 1945 (`https://tools.ietf.org/html/rfc1945`) is the complete specification for the HTTP/1.0 protocol.

8.2 Tiny web server / CS:APP code

Your handout directory the source code for the CS:APP Tiny web server. While not as powerful as `thttpd`, the CS:APP Tiny web server will be easy for you to modify as you see fit. It's also a reasonable starting point for much of your proxy code. And it's the server that the driver code uses to fetch pages.

That being said, do *not* use CS:APP's code for preparing sockets (e.g., `Open_listenfd`) or its code robust I/O code (i.e., functions starting with `rio_`). See more information and guidance below and setting up sockets and reading and writing robustly from a socket.

Using the error-handling functions provide in `csapp.c` is allowed but not encouraged, as I would rather have you know what you are doing in terms of error handling. Anyway, some of them would need to be modified to behave appropriately for your proxy because once a server begins accepting connections, it is not supposed to terminate (some of the actions in those functions is to simply exit).

8.3 Code from previous homeworks

In previous assignments for this class you have written code to read from a socket using a loop and to write to a socket using a loop. You are also written code to parse an HTTP request. You can use the code already provided to you in your sockets homework assignment (i.e., in `client.c` and `server.c`). Finally, you are free to adapt the code provided to you in the concurrency homework for your producer-consumer queue.

8.4 Other Hints

As discussed in Section 10.11 of your textbook, using standard I/O functions for socket input and output is a problem. Use your own robust `read()` (or `recv()`) loop (i.e., similar to that you created in your sockets homework assignment). The general idea is to keep reading from the socket into a buffer until you have some assurance that you have everything you need in the buffer to move on.

For example, when receiving an HTTP request from a client, you can call `read()` within a loop to keep reading bytes from the socket until you know you have the entire HTTP request (i.e., as indicated with `"\r\n\r\n"` which signal the end of headers). Similarly, when receiving an HTTP response from a server, you can call `read()` within a loop to keep reading bytes from the socket until you know you have received the entire response. Because we are only using HTTP/1.0 in this lab, your signal here that you have received the entire response is when the remote server disconnects, and your `read()` returns 0 (With HTTP/1.1, the server would, by default, keep the TCP connection open for future requests, so you would need to read the `Content-length` header to determine how many bytes to read).

One of the concurrency tests used by the driver runs 10 slow requests followed by 10 fast ones. You will want your proxy server, therefore, to handle *more* than 10 requests concurrently. This translates to having more than 10 worker threads.

You are free to modify the files in the handout directory any way you like. Of course, adding new files will require you to update the provided Makefile, so we can compile it properly.

Remember that not all content on the web is ASCII text. Much of the content on the web is binary data, such as images and video. Ensure that you account for binary data when selecting and using functions for network I/O. For example, you will want to depend heavily on the return value from `read()`, rather than calling (for example) `strlen()` to determine how many bytes have been received.

Finally, forward all requests as HTTP/1.0 even if the original request was HTTP/1.1.

8.5 Error handling

As discussed in the Aside on page 964 of the CS:APP3e text, your proxy must ignore SIGPIPE signals and should deal gracefully with `write` operations that return EPIPE errors.

Sometimes, calling `read` to receive bytes from a socket that has been prematurely closed will cause `read` to return `-1` with `errno` set to `ECONNRESET`. Your proxy should not terminate due to this error either.

Good luck!

9 Grading

The following is the point breakdown:

- - 50 points for basic proxy operation
- - 45 points for handling concurrent requests using a threadpool
- - 5 compiles without any warnings

The maxmimum number of points is 100.

10 Handin instructions

The provided Makefile includes functionality to build your final handin file. Issue the following command from your working directory:

```
linux> make handin
```

The output is the file `../proxylab1-handin.tar`, which you can then handin.

To submit your lab, please upload your `proxylab1-handin.tar` file to the appropriate assignment page on Learning Suite.