CS105 – Computer Systems Fall 2019

Assignment 7: Proxy Lab

Due: Tuesday, December 10, 2019 at 11:59pm

Recall from class that a proxy is a program that acts as a go-between between a *web browser* and an *end server.* Instead of contacting the end server directly to get a webpage, 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. 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 back 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. In the third and last part, you will add an interesting new feature to your proxy.

Handout Instructions

As usual, you will download the file `proxylab.tar` from the course web page.

Copy the handout file to a protected directory on server, and then issue the following command:

```
linux> tar xvf proxylab.tar
```

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

1 Implementing a sequential web proxy (45 points)

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.

1.1 HTTP/1.0 GET Requests

When an end user enters a URL such as `http://www.cs.pomona.edu/classes/cs105/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.cs.pomona.edu/classes/cs105/index.html HTTP/1.1
```

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

```
GET /classes/cs105/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'`.

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.

1.2 Request Headers

Here are the important request headers for this lab.

- 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.cs.pomona.edu/classes/cs105/index.html`, your proxy would send the following header:
  ```
  Host: www.cs.pomona.edu
  ```
  It is possible that web browsers will attach their own `Host` headers to their HTTP requests. If that is the case, your proxy should use the same `Host` header as the browser.
- You *may* choose to always send the following `User-Agent` header:
  ```
  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3)
      Gecko/20120305 Firefox/10.0.3
  ```
  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.
For convenience, the values of the described `User-Agent` header is provided to you as a string constant in `proxy.c`.

- 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. It is perfectly acceptable (and suggested) to have your proxy open a new connection for each request. Specifying `close` as the value of these headers alerts web servers that your proxy intends to close connections after the first request/response exchange.

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

1.3 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.cs.pomona.edu:4747/classes/cs105/index.html`, in which case your proxy should connect to the host `www.cs.pomona.edu` on port 4747 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 91711:

```
linux> ./proxy 91711
```

You may select any non-privileged listening port (greater than 1,024 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 do not pick your own random port. If you do, you run the risk of interfering with another user.

2 Multiple Concurrent Requests (5 points)

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. Other designs are also possible, such as the prethreaded server described in Section 12.5.5 of your textbook.

- Note that your threads should run in detached mode to avoid memory leaks.
- The `open_clientfd` and `open_listenfd` functions described in the CS:APP3e textbook are based on the modern and protocol-independent `getaddrinfo` function, and thus are thread safe.

3 Feedback (2 points)

Create a file called `feedback.txt` that answers the following questions:

1. How long did each of you spend on this assignment?
2. Any comments on this assignment?

As always, how you answer these questions **will not affect your grade**, but whether you answer them will.

4 Submission and Evaluation

4.1 Submission

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 `../proxylab-handin.tar`, which you can then submit.

4.2 Evaluation

This assignment will be graded out of a total of 52 points:

- *BasicCorrectness*: 45 points for basic proxy operation
- *Concurrency*: 5 points for handling concurrent requests
- *Feedback*: 2 points for submitting a feedback file

4.3 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.

4.4 Demonstration

We will use our last laboratory session, on December 11, to demo your work for the course staff.

.1 General References

Chapters 10-12 of the textbook contains useful information on system-level I/O, network programming, HTTP protocols, and concurrent programming.

RFC 1945 (`http://www.ietf.org/rfc/rfc1945.txt`) is the complete specification for HTTP/1.0.

A Notes on Testing and Debugging

We recommend that you start by testing your proxy with the tiny web server (provided) and simple text-based client (e.g., `telnet` or `curl`). However, a completely correct implementation should be able to handle http requests from a browser to an arbitrary webpage. Your implementation will only be tested with http requests; you are not required to support https.

A.1 Tiny Web Server

Your handout directory the source code for the CS:APP Tiny web server. While not as powerful as `httpd`, 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 your proxy code. And it is the server that the driver code uses to fetch pages.

A.2 `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.

A.3 `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
```

A.4 Web Browsers

Mozilla Firefox works well for testing your proxy. To configure Firefox to work with a proxy, visit

```
Preferences > General > 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.

Be careful if you decide to implement caching in your proxy. All modern Web browsers have caches of their own, which you should disable before attempting to test your proxy's cache.

A.5 Hints

- As discussed in Section 10.11 of your textbook, using standard I/O functions for socket input and output is a problem. Instead, we recommend that you use the Robust I/O (RIO) package, which is provided in the `csapp.c` file in the handout directory.
- The error-handling functions provide in `csapp.c` are not appropriate for your proxy because once a server begins accepting connections, it is not supposed to terminate. You'll need to modify them or write your own.
- You are free to modify the files in the handout directory any way you like. For example, for the sake of good modularity, you might implement your cache functions as a library in files called `cache.c` and `cache.h`. Of course, adding new files will require you to update the provided Makefile.
- 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.
- 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.
- Forward all requests as HTTP/1.0 even if the original request was HTTP/1.1.

Good luck!