15-213/15-513, Summer 2024
Proxy Lab: Writing a Web Proxy

Assigned: Friday, July. 19
Due: Friday, August. 2
Last Possible Handin: **Friday, August. 2**

1 Introduction

A *proxy server* is a computer program that acts as an intermediary between clients making requests to access resources and the servers that satisfy those requests by serving content. A *web proxy* is a special type of proxy server whose clients are typically web browsers and whose servers are web servers providing web content. When a web browser uses a proxy, it contacts the proxy instead of communicating directly with the web server; the proxy forwards the client's request to the web server, reads the server's response, then forwards the response to the client.

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. A proxy may also perform translations on pages, for example, to make them viewable on web-enabled phones. Importantly, proxies are used as anonymizers: by stripping requests of all identifying information, a proxy can make the browser anonymous to web servers.

This lab has two parts. For the first part, you will create a proxy that accepts incoming connections, reads and parses requests, forwards requests to web servers, reads the servers' responses, and forwards the responses to the corresponding clients. The 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. This will introduce you to dealing with concurrency, a crucial systems concept.

You will debug and test your program with PxyDrive, a testing framework we provide, as well as by accessing your proxy via standard tools, including a web browser. The grading of your code will involve automated testing. Your code will also be reviewed for correctness and for style.

2 Logistics

This is an individual project. You are allowed no grace or late days.

3 Handout instructions

Create your GitHub Classroom repository at `https://classroom.github.com/a/FLQsFpj0`. or by clicking the "Download handout" button on the proxylab Autolab page. Then do the following on a Shark machine:

- Clone the repository that you just created using the `git clone` command. *Do not download and extract the zip file from GitHub.*
- Type your name and Andrew ID in the header comment at the top of `proxy.c`.

3.1 Robust I/O package

The handout directory contains the files `csapp.c` and `csapp.h`, which comprise the CS:APP package discussed in the CS:APP3e textbook. The CS:APP package includes the robust I/O (RIO) package. When reading and writing socket data, you should use the RIO package instead of low-level I/O functions, such as `read`, `write`, or standard I/O functions, such as `fread`, and `fwrite`.

The CS:APP package also contains a collection of *wrapper* functions for system calls that check the return code and exit when there's an error. You will find that the set of wrapper functions provided is a subset of those from the textbook and the lecture notes. We have disabled ones for which exiting upon error is not the correct behavior for a server program. For these, you must check the return code and devise ways to handle these errors that minimize their impact.

3.2 HTTP parsing library

The handout directory contains the file `http_parser.h`, which defines the API for a small HTTP string parsing library. The library includes functions for extracting important data fields from HTTP response headers and storing them in a `parser_t` struct. A brief overview of the library is given below. Please refer to the source files in your handout for the full documentation of the types, structs, and functions available for use in the library. Having a good understanding of the library will be very helpful during this lab.

To start using this library, you first create a new instance of a parser struct using `parser_new()`. This instance will parse all of the lines (both request lines and headers) from a single HTTP request and cannot be reused for later requests. You should use the returned pointer to parse the request and receive all data. The parser instance can be passed to other functions to access its parsed results and help your proxy with identifying the fields and headers of the request.

`parser_parse_line()` will parse a line of an HTTP request and store the result in the provided `parser_t` struct. Parsed fields of specified types may be retrieved from the struct by calling `parser_retrieve()` and by providing a string pointer for the function to write to. Particular headers may also be retrieved by name via `parser_lookup_header()`. Headers may instead be accessed in an iterative fashion by successive calls to `parser_retrieve_next_header()`.

Lastly, you should make sure to free all resources allocated by your parser on error or when you are done using it. While using the parser, please be cautious to not corrupt (i.e. modify) parsed header data.

3.3 Modularity

The skeleton file `proxy.c`, provided in the handout, contains a `main` function that does practically nothing. You should fill in that file with your proxy implementation. Modularity, though, should be an important consideration, and it is important for you to separate the individual parts of your implementation into different functions.

3.4 Makefile

You are free to add your own source and header files for this lab. The Makefile will automatically link all `.c` files into the final binary. While you are free to update the provided Makefile (for example to define the `DEBUG` macro), the autograder will use the original Makefile to grade your solution. As such, the entire project should compile without warnings.

3.5 Other provided resources

Included with your starter code, in the `pxy` directory, is a pair of programs PxyDrive and PxyRegress (given as files `pxydrive.py` and `pxyregress.py`, respectively.) PxyDrive is a testing framework for your proxy. PxyRegress provides a way to run a series of standard tests on your proxy using PxyDrive. Both programs are documented in the PxyDrive user manual, available at:

`http://www.cs.cmu.edu/~213/proxylab/pxydrive-manual.pdf`.

Also included, in the `tests` directory, is a series of 35 test files to test various aspects of your proxy. Each of these is a command file for PxyDrive. You will want to learn about the operation of PxyDrive and how each of these tests operate.

Finally, you are provided with a reference implementation of a proxy, named `proxy-ref`. It is compiled to execute on a Linux machine.

4 Part I: Implementing a sequential web proxy

The first step is implementing a basic sequential proxy that handles HTTP/1.0 GET requests. Your proxy need not handle other request types, such as POST requests, but it should respond appropriately, as described below. Your proxy also need not handle HTTPS requests (only HTTP).

When started, your proxy should listen for incoming connections on a port whose number is 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 should 1) establish its own connection to the appropriate web server, 2) request the object the client specified, and 3) read the server's response and forward it to the client.

4.1 HTTP/1.0 GET requests

When an user enters a URL such as `http://www.cmu.edu/hub/index.html` into the address bar of a web browser, the browser will send an HTTP request to the proxy that begins with a **request line** such as the following:

```
GET http://www.cmu.edu:8080/hub/index.html HTTP/1.1\r\n
```

The proxy should parse the request URL into the **host**[1], in this case `www.cmu.edu:8080`, and the **path**[2],

[1]This is also referred to as the *authority*, to avoid confusion with the *hostname*, `www.cmu.edu`, which does not include the *port*, `8080`. See the diagram at `https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Technical_overview`.

consisting of the `/` character and everything following it. That way, the proxy can determine that it should open a connection to hostname `www.cmu.edu` on port `8080` and send an HTTP request of its own, starting with its own request line of the following form:

```
GET /hub/index.html HTTP/1.0\r\n
```

As these examples show, all lines in an HTTP request end with a carriage return (‘`\r`’) followed by a newline (‘`\n`’). Also important is that every HTTP request must be terminated by a line, consisting of just the string “`\r\n`”.

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.

Additionally, in the above example, a port number of `8080` was specified as part of the host. *If no port is specified, the default HTTP port of `80` should be used.*

4.2 Request headers

Request headers are very important elements of an HTTP request. Headers are **constant** key-value pairs provided line-by-line following the first request line of an HTTP request, with the key and value separated by the colon (‘`:`’) character. Of particular importance for this lab are the `Host`, `User-Agent`, `Connection`, and `Proxy-Connection` headers. Your proxy must perform the following operations with regard to the listed HTTP request headers:

- Always send a `Host` header. This header is necessary to coax sensible responses out of many web servers, especially those that use virtual hosting.

  The `Host` header describes the host of the web server your proxy is trying to access. For example, to access `http://www.cmu.edu:8080/hub/index.html`, your proxy would send the following header:

  ```
  Host: www.cmu.edu:8080\r\n
  ```

  It is possible that the client will attach its own `Host` header to its HTTP requests. If that is the case, your proxy should use the same `Host` header as the client.

- You should always send the following `User-Agent` header:

  ```
  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:3.10.0)
      Gecko/20240719 Firefox/63.01\r\n
  ```

  The header shown above on two separate lines because it does not fit as a single line, but your proxy should send the header as a single line. It is provided as a string constant in `proxy.c`.

  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 returned by some web servers.

[2]There is also the query portion, which starts with the `?` character, but distinguishing between the path and the query is not important for our purposes.

- Always send the following `Connection` header:

  ```
  Connection: close\r\n
  ```

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

  ```
  Proxy-Connection: close\r\n
  ```

  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. Specifying `close` as the value of these headers alerts web servers that your proxy intends to close connections after the first request/response exchange.

With the exception of the `Host` header, your proxy should ignore the values of the request headers described above provided by the client. Instead, it should always send the headers this document specifies.

Finally, **if a client sends any additional request headers as part of an HTTP request, your proxy should forward them unchanged.** Forwarding other request headers is especially important when using PxyDrive to test your proxy, because it uses headers to coordinate the actions of its client and servers.

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.cmu.edu:8080/hub/index.html`, in which case your proxy should connect to the host `www.cmu.edu` on port 8080, and it should include the port number in the `Host` header (e.g., `Host: www.cmu.edu:8080`.)

Your proxy must properly function whether or not the port number is included in the URL. *If no port is specified, the default HTTP port number of `80` should be used,* which should not be included in the `Host` header.

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 12345:

```
linux> ./proxy 12345
```

The proxy must be given a port number every time it runs. When using PxyDrive, this will be done automatically, but when you run your proxy on its own, you must provide a port number. You may select any non-privileged port (greater than 1,024 and less than 32,768) as long as it is not used by other processes. Since each proxy must use a unique listening port, and many students may be working simultaneously on each machine, the script `port-for-user.pl` is provided to help you pick your own personal port number. Use it to generate a port number based on your Andrew ID:

```
linux> ./port-for-user.pl bovik
bovik: 5232
```

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

4.4 Error handling

In the case of invalid requests, or valid requests that your proxy is unable to handle, it should try to send the appropriate HTTP status code back to the client. (see `clienterror()` in `proxy.c`). Read more about HTTP status codes at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

In particular, your proxy must be able to respond to a POST request with the `501 Not Implemented` status code. The request line for a POST request will resemble the following:

```
POST http://exams.ugrad.cs.cmu.edu/Shibboleth.sso/SAML2/POST HTTP/1.1\r\n
```

In other cases, it is acceptable for your proxy to simply close the connection to the client when an error occurs, using `close()`. Note that in all error cases, you should always clean up all resources being used to handle a given request, including file descriptors and allocated memory.

Note: Upon normal execution, your proxy should not print anything. However, you should consider having a verbose mode (set with `-v` on the command line) that prints useful information for debugging.

5 Part II: Dealing with multiple concurrent requests

Production web proxies usually do not process requests sequentially; they process multiple requests in parallel. This is particularly important when handling a single request can involve a lengthy delay (as it might when contacting a remote web server). While your proxy waits for a response from the remote web server, it can work on a pending request from another client. Indeed, most web browsers reduce latency by issuing concurrent requests for the multiple URLs embedded in a single web page requested by a single client. Once you have a working sequential proxy, you should alter it to simultaneously handle multiple requests.

5.1 POSIX Threads

You will use the POSIX Threads (Pthreads) library to spawn threads that will execute in parallel to serve multiple simultaneous requests. A simple way to serve concurrent requests is to spawn a new thread to process each request. In this architecture, the main server thread simply accepts connections and spawns off independent worker threads that deal with each request to completion and terminate when they are done. Other designs are also viable: you might alternatively decide to have your proxy create a pool of worker threads from the start. You may use any architecture you wish as long as your proxy exhibits true concurrency, but spawning a new worker thread for each request is the most straightforward approach.

A new thread is spawned in Pthreads by calling `pthread_create` with a start routine as its argument. New threads are by default joinable, which means that another thread must clean up spare resources after the thread exits, similar to how an exited process must be reaped by a call to `wait`. For a server, a better approach is to *detach* each new thread, so that spare resources are automatically reaped upon thread exit. To properly detach threads, the first line of the start routine should be as follows:

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

5.2 Thread safety

The `open_clientfd` and `open_listenfd` functions described in the CS:APP3e textbook (and included in `csapp.c`) are based on the modern and protocol-independent `getaddrinfo` function, and thus are thread

	Series	Count	Points each	Total
A	Basic proxy operation	12	2	24
B	Robustness and compliance	13	1	13
C	Concurrency	10	3	30
Total				**67**

Figure 1: Score Computation

safe.

6 Evaluation

The grading standards for the assignment are shown in Figures 1. Grading will be via a combination of automatic testing and a manual review of your code. The automatic grading will be done by running PxyRegress with the same command files you are provided. There are a total of 35 files, divided into three series, labeled A–C, as described in Section 7 of the user manual.

As shown in Figure 1, your submission will be evaluated using all three series. Series C tests can only be passed by a proxy that supports concurrent requests. You must pass all of the A/B tests in order to obtain any points in the C tests.

Finally, there are a total of 4 style points, also described below.

6.1 Autograding with PxyRegress

The PxyDrive testing framework is included in the handout in the subdirectories `pxy` (code) and `tests` (command files).

A typical invocation of PxyDrive from your code directory would be:

```
linux> pxy/pxydrive.py -p ./proxy -f tests/A01-single-fetch.cmd
```

With this invocation, the proxy will operate *internally*, under the control of PxyDrive. Alternatively, the proxy can be set up and run *externally*:

First, recall from section 4.3 how to generate a unique port.

```
linux> ./port-for-user.pl bovik
bovik: 5232
```

You can now use this port as follows:

```
linux> ./proxy 5232
linux> pxy/pxydrive.py -P localhost:5232 -f tests/A01-single-fetch.cmd
```

(Normally, these two lines would be executed in separate terminal windows.) Running the proxy externally makes it possible to use programs such as `gdb` and `valgrind` to help with debugging. See Section 7 of the user manual for more detailed instructions.

You can perform testing on trace A and trace B with the command:

```
linux> pxy/pxyregress.py -p ./proxy -s AB
```

The final testing will be done with a more comprehensive set of concurrency checks, inserting random delays to exercise possible races, and checking for thread-unsafe functions. You can perform this test with the command

```
linux> pxy/pxyregress.py -p ./proxy -c 4
```

For your convenience, the driver Autolab will use to test your submission is included as the file `driver.sh`.

For testing your submission, it can be invoked as:

```
linux> ./driver.sh
```

6.2 Style Grading

The style points include a combination of points for code correctness and for coding style. Your TAs will examine your code for any correctness issues that weren't detected by the earlier tests. In particular, we will be looking for errors such as thread safety issues, improper error handling, and memory and file descriptor leaks.

Style points will be awarded based on the usual criteria. Proper error handling is as important as ever, and modularity is of particular importance for this lab, as there will be a significant amount of code. You should also strive for portability.

7 Real-World Testing

Along with PxyDrive, you can also test your proxy's operation using real web browsers and servers. You must set up and run these tests using standard tools, including those described here.

7.1 Web browsers

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

To configure Firefox to work with a proxy, visit `Preferences>Network Proxy>Settings` in the Firefox menu and set the manual proxy configuration with your host name of the specific Shark machine you are using and with the port number for your proxy.

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 many websites through your proxy.

8 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 `../proxylab-handin.tar`. Simply upload it to Autolab. Autolab will use your Makefile to rebuild your proxy from scratch, and it will then run tests using PxyRegress.

9 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. The specification is large and complex. You only need to implement the portions of the protocol we describe in this document.
- HTTP/1.0 has been obsolete since the early 2000s. For modern-day HTTP specifications, see `https://developer.mozilla.org/en-US/docs/Web/HTTP/Resources_and_specifications`. However, implementing HTTP/1.1 would require our proxy to implement `Connection: keep-alive`, which would add additional complexity.

10 Advice

- It is important to adopt the proper mindset when writing a server program. Think carefully about how long-running processes should react to different types of errors, including network failures and malformed data. Your server should do its best to continue in the face of errors. For example, it's OK to fail to respond to a malformed request from a client, but this should not cause your program to terminate. Robustness implies a higher level of code quality, as well, avoiding defects such as segmentation faults, memory leaks, and file descriptor leaks.
- Your proxy will have to do something about `SIGPIPE` signals. The kernel will sometimes deliver a `SIGPIPE` to a process when a socket gets disconnected. Although the default action for a process that receives `SIGPIPE` is to terminate, your proxy should not terminate due to that signal. (To generate the `SIGPIPE` signal in Firefox, press `F5` twice in quick succession to reload the page.)
- 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`. The same result can occur when calling `write` to send bytes on a socket that has been prematurely closed. Your proxy should not terminate due to these errors.
- 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. In particular, you should never use string functions (e.g., `strlen` and `strcpy`) on web data, since these will terminate prematurely when then encounter a byte with value 0, which can occur at any position in a binary file. (The preferred way to copy data is to use `memcpy`.)
- You *can* assume that the request and header lines will be ASCII text, and you can assume that there is some reasonable upper bound on their line lengths.
- Similarly, it is inefficient to use line-oriented I/O routines, such as `rio_readlineb`, on binary data. These routines will break the data into chunks according to the placement of newline characters (hex value `0x0A`). With binary data, these can appear in arbitrary positions in a file. Instead, use functions, such `rio_readn` and `rio_readnb`, that operate independently of the data values.

- Forward all requests as HTTP/1.0 even if the original request was HTTP/1.1.
- Firefox (like all modern web browsers) has useful debugging tools you may take advantage of. The network tab allows you to view the status of network requests, which is useful for determining if your threads terminate and if all requests are fulfilled.