CSC498U, Fall 2018
Proxy Lab: Writing a Caching Web Proxy
Assigned: Tue, Oct 2
Due: Tue, Oct 23, 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 that caches web 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. This will introduce you to dealing with concurrency, a crucial systems concept. In the third and last part, you will add caching to your proxy using a simple main memory cache of recently accessed web content.

2 Logistics

You are allowed to work on this project in a group of two students.

3 Handout instructions

The handout file contains a starter code with the necessary data structures needed to implement a working web proxy. The source code of the Tiny web server that we have seen in class is also included. While not as powerful as `thttpd`, the 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.

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://example.com/path/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://example.com/path/index.html HTTP/1.1

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

    GET /path/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.

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://example.com/path/index.html`, your proxy would send the following header:

      Host: example.com

  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) AppleWebKit/537.36
      (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36

  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. 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.

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

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://example.com:8080/path/index.html`, in which case your proxy should connect to the host `example.com` 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 1,024 and less than 65,536) as long as it is not used by other processes.

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. 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.

6 Part III: Caching web objects

For the final part of the lab, you will add a cache to your proxy that stores recently-used Web objects in memory. HTTP actually defines a fairly complex model by which web servers can give instructions as to how the objects they serve should be cached and clients can specify how caches should be used on their behalf. However, your proxy will adopt a simplified approach.

4

When your proxy receives a web object from a server, it should cache it in memory as it transmits the object to the client. If another client requests the same object from the same server, your proxy need not reconnect to the server; it can simply resend the cached object.

Obviously, if your proxy were to cache every object that is ever requested, it would require an unlimited amount of memory. Moreover, because some web objects are larger than others, it might be the case that one giant object will consume the entire cache, preventing other objects from being cached at all. To avoid those problems, your proxy should have both a maximum cache size and a maximum cache object size.

6.1 Maximum cache size

The entirety of your proxy's cache should have the following maximum size:

```
MAX_CACHE_SIZE = 1 MiB
```

When calculating the size of its cache, your proxy must only count bytes used to store the actual web objects; any extraneous bytes, including metadata, should be ignored.

6.2 Maximum object size

Your proxy should only cache web objects that do not exceed the following maximum size:

```
MAX_OBJECT_SIZE = 100 KiB
```

For your convenience, both size limits are provided as macros in `cache.h`.

The easiest way to implement a correct cache is to allocate a buffer for each active connection and accumulate data as it is received from the server. If the size of the buffer ever exceeds the maximum object size, the buffer can be discarded. If the entirety of the web server's response is read before the maximum object size is exceeded, then the object can be cached. Using this scheme, the maximum amount of data your proxy will ever use for web objects is the following, where `T` is the maximum number of active connections:

```
MAX_CACHE_SIZE + T * MAX_OBJECT_SIZE
```

6.3 Eviction policy

Your proxy's cache should employ an eviction policy that approximates a least-recently-used (LRU) eviction policy. It doesn't have to be strictly LRU, but it should be something reasonably close. Note that both reading an object and writing it count as using the object.

6.4 Synchronization

Accesses to the cache must be thread-safe, and ensuring that cache access is free of race conditions will likely be the more interesting aspect of this part of the lab. As a matter of fact, there is a special requirement

that multiple threads must be able to simultaneously read from the cache. Of course, only one thread should be permitted to write to the cache at a time, but that restriction must not exist for readers.

As such, protecting accesses to the cache with one large exclusive lock is not an acceptable solution. You may want to explore options such as partitioning the cache, using Pthreads readers-writers locks, or using semaphores to implement your own readers-writers solution. In either case, the fact that you don't have to implement a strictly LRU eviction policy will give you some flexibility in supporting multiple readers.

7 Evaluation

This assignment will be graded out of a total of 100 points:

- *BasicCorrectness*: 60 points for basic proxy operation

- *Concurrency*: 20 points for handling concurrent requests

- *Cache*: 20 points for a working cache

7.1 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.

8 Testing and debugging

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.

8.1 `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.

8.2 `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
```

8.3 `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 `yourproxyserver.com` using port 12345 you can do something like the following to manually test your proxy:

```
sh> nc yourproxyserver.com 12345
GET http://example.com/path/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:

7

```
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`.

8.4 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.

An important caveat is that you must be very careful when testing caching using a Web browser. All modern Web browsers have caches of their own, which you should disable before attempting to test your proxy's cache.

9 Handin instructions

Submit your code using the following link:

```
https://docs.google.com/forms/d/e/1FAIpQLSdveawj7xaqJ70oyVb304ob3D1usvm48XhIWJMM
-Sm-B-G7w/viewform
```

10 Hints

- As discussed in class, 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.

- 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!