Lab 3

Building a web proxy

INTRODUCTION

We have explored the idea of a *caching* web proxy in class and through textbook readings. We found these were the heart of how CDNs work, allowing cached services to be deployed closer to the consumer and increasing network responsiveness. We also learned that caching web proxies reduced traffic intensity at access network connections, and could increase performance and/or lower costs.

In this assignment, we will build a simple network proxy: a web proxy. This service needs to contain both client and server logic. It will need to generate valid application-level error messages in some situations. Otherwise, it will need to interpret the application-level request messages, modify these appropriately, and proxy them.

Building this service will require socket programming, using the forking-server pattern, understanding the format of both HTTP request and responses, and will let us learn about the nuances of proxying connections successfully.

PROXY BEHAVIOR

Your web proxy will have to translate between requests that the client makes (the one that starts with `GET http://remoteserver`) into requests that the remote server understands. Your web proxy will listen on a port the user specifies on startup, to avoid conflicts with other servers.

Generally, once the request line has been received, the web proxy should continue reading the input from the client until it encounters the CRLF. The proxy should then fetch the URL from the remote server specified in that request, forward its response back to the client, and then close the connection. The proxy should forward response data as it arrives, rather than buffering the entire response; this allows the proxy to handle huge responses without running out of memory.

To get a sense of how a proxy works, lets use different utilities to see the view from the client's and server's perspectives. In the examples that follow, commands on the client-side, proxy-side and server-side have, respectively, the following prompts:

```
[client] $
[proxy] $
[server] $
```

You may be able to adapt and run these examples yourself with some effort. For example, the squid proxy config file (squid.conf) has been released with the other materials for the project (see the Section “Resources”). We use netcat (nc) as a stand-in for the proxy and server in these examples, to see the response before and after it has been proxied.

Example 1a.

Server Perspective (Non-proxied request)

Here, nc is acting like a webserver, and showing us what a standard request looks like.

```
[server] $ nc -lp 3333
[client] $ lynx http://blue.cs.sonoma.edu:3333/page_name.html
```

The request looks like:

```
GET /page_name.html HTTP/1.0
Host: blue.cs.sonoma.edu:3333
Accept: text/html, text/plain, text/sgml, text/css,
application/xhtml+xml, */*;q=0.01
Accept-Encoding: gzip, bzip2
Accept-Language: en
User-Agent: Lynx/2.8.9dev.4 libwww-FM/2.14 SSL-MM/1.4.1
OpenSSL/1.0.1k-fips
```

Example 1b.

Server Perspective (Non-proxied request)

Same as the prior example, with a different client.

```
[server] $ nc -lp 3333
[client] $ wget http://blue.cs.sonoma.edu:3333/page_name.html
```

The request looks like:

```
GET /page_name.html HTTP/1.1
User-Agent: Wget/1.18 (linux-gnu)
Accept: */*
Accept-Encoding: identity
Host: blue.cs.sonoma.edu:3333
Connection: Keep-Alive
```

Example 2a.

Proxy Perspective (Proxied request)

Here, nc is acting like the proxy, and showing us what a standard request via a proxy looks like on the proxy side.

```
[proxy] $ nc -lp 3128
[client] $ env http_proxy=http://blue.cs.sonoma.edu:3128 lynx
http://blue.cs.sonoma.edu:3333/page_name.html
```

The request looks like:

```
GET http://blue.cs.sonoma.edu:3333/page_name.html HTTP/1.0
Host: blue.cs.sonoma.edu:3333
Accept: text/html, text/plain, text/sgml, text/css,
application/xhtml+xml, */*;q=0.01
Accept-Encoding: gzip, bzip2
Accept-Language: en
User-Agent: Lynx/2.8.9dev.4 libwww-FM/2.14 SSL-MM/1.4.1
OpenSSL/1.0.1k-fips
```

Example 2b.

Proxy Perspective (Proxied request)

```
[proxy] $ nc -lp 3128
[client] $ env http_proxy=http://blue.cs.sonoma.edu:3128 wget
http://blue.cs.sonoma.edu:3333/page_name.html
```

The request looks like:

```
GET http://blue.cs.sonoma.edu:3333/page_name.html HTTP/1.1
User-Agent: Wget/1.18 (linux-gnu)
Accept: */*
Accept-Encoding: identity
Host: blue.cs.sonoma.edu:3333
Connection: Keep-Alive
Proxy-Connection: Keep-Alive
```

Example 3a.

Server Perspective (Proxied request)

Here, nc is acting like the server, and showing us what a request looks like after it has been proxied by squid, a popular proxy.

```
[server] $ nc -lp 3333
[proxy] $ squid -f squid.conf -N
[client] $ env http_proxy=http://blue.cs.sonoma.edu:3128 wget
http://blue.cs.sonoma.edu:3333/page_name.html
```

The request looks like:

```
GET /page_name.html HTTP/1.1
User-Agent: Wget/1.18 (linux-gnu)
Accept: */*
Accept-Encoding: identity
Host: blue.cs.sonoma.edu:3333
Via: 1.1 blue.cs.sonoma.edu (squid/3.5.10)
X-Forwarded-For: 130.157.166.29
Cache-Control: max-age=259200
Connection: keep-alive
```

Example 3b.

Server Perspective (Proxied request)

```
[server] $ nc -lp 3333
[proxy] $ squid -f squid.conf -N
[client] $ env http_proxy=http://blue.cs.sonoma.edu:3128 lynx
http://blue.cs.sonoma.edu:3333/page_name.html
```

The request looks like:

```
GET /page_name.html HTTP/1.1
Host: blue.cs.sonoma.edu:3333
Accept: text/html, text/plain, text/sgml, text/css,
application/xhtml+xml, */*;q=0.01
Accept-Encoding: gzip, bzip2
Accept-Language: en
User-Agent: Lynx/2.8.9dev.4 libwww-FM/2.14 SSL-MM/1.4.1
OpenSSL/1.0.1k-fips
Via: 1.0 blue.cs.sonoma.edu (squid/3.5.10)
X-Forwarded-For: 130.157.166.29
Cache-Control: max-age=259200
Connection: keep-alive
```

Observations

We can notice a few things from these examples:

- At the proxy, the request holds a URI that includes the full URL:
  `GET **http://blue.cs.sonoma.edu:3333/page_name.html** HTTP/1.1`
  however, when this request is forwarded along, the URI is shortened to only include the path:
  `GET **/page_name.html** HTTP/1.1`
- Our squid proxy has added some fields (Via, X-Forwarded-For, Cache-Control). Your proxy is *not* responsible for doing this.
- The squid proxy has modified some client request fields (Proxy-Connection) to be relevant to the proxied connection (Connection). Your proxy is responsible for this.

SOCKET PROGRAMMING

The general flow of the server is the following:

1. Create a listening socket for the server
    - Call socket()
    - Use setsockopt() to set the SO_REUSEADDR flag (see man 2 setsockopt) to simplify development; this will reduce the bind expiry timeout.
    - Call bind() and listen()
2. Detect read activity on the listen socket, e.g., using select()
3. If there is activity, accept() the connection and fork()
4. In the *server*:
    - Close the resources we do not need, like the connected fd the child will service
    - Occasionally and opportunistically reap zombie children using waitpid()
    - Go back to (2) and service more clients.
5. In the *child*:
    - Close the resources we don't need, like the listen fd
    - Handle the client's connection using read(), write(), fread(), fwrite(), etc.
    - Be careful using some functions, like fgets() which only work on ASCII data (like client requests) but may not work properly on other communication (like server responses holding large binary files).

The general flow of how the *child* will service the request is:

1. Read the request to find the URL
2. Figure out the identity of the remote server from the URL using getaddrinfo()
3. Use socket() and connect() to connect to the remote server as a client
4. Send the client's request to the remote server, line-by-line, adjusting certain lines of the request, as appropriate.

5. Receive the remote server's response and send this to the client.
    - Do not receive the full response before sending to the client, or your client may run out of memory when large files are being transmitted.
6. Close the connection.

RESOURCES

Here is an extremely simple skeleton available you may use for your project:

- `git clone https://github.com/gondree/webproxy-assignment.git`

I've made several example client/servers as models of how to use important system calls:

- http://blue.cs.sonoma.edu/~gondree/sample-client-server-ex.tgz
- http://blue.cs.sonoma.edu/~gondree/sample-webserver-ex.tgz

Finally, here is Beej's socket programming guide and the HTTP/1.0 RFC

- Beej's Guide: http://beej.us/guide/bgnet/
- RFC 1945: https://www.w3.org/Protocols/rfc1945/rfc1945

SPECIFICATION

Your code submission should have the following properties:

- **Builds.** Your assignment should create a binary named proxy that can be compiled and runs on blue. It should build using a Makefile via the target 'all.' You can complete the assignment in either ANSI C or C++.
- **Command Line Arguments.** Your proxy must support the following command-line arguments:

```
./proxy          # output usage message when too few/many arguments
./proxy --help    # output usage message
./proxy 12345    # proxy will listen to the port in the first arg
./proxy 54321 --verbose  # outputs debug, if needed
```

- **Silent.** Your proxy should run silently — any status messages or diagnostic output should be *off* by default. If you need to include diagnostic messages, they should only be shown on stdout when the --verbose flag is activated. Error messages may be send to stderr, whether the --verbose flag has been set or not.
- **Server Architecture**. Your proxy should fork on each new client connection and fully handle each client's request. It should leave no zombie processes. Every client should

receive a response, either the response from the remote server or an error message from the proxy. No client request should go unanswered.

- **Client Support.** Your proxy should support at least the clients wget and lynx.
- **Basic Functionality**. At least, your proxy must able to handle:
    - Proxy valid HTTP/1.0 GET requests to the remote server
    - Proxy responses from the remote server back to the client
    - Respond to the client with HTTP/1.0 error messages, e.g., on malformed requests
    - See remarks on testing Basic Functionality, below.
- **Error checking**. At a minimum, all system calls are error-checked.
    - A fail-fast philosophy is preferred usually; one exception is when an error is encountered while handling a client request, where providing the client an application-level error message is preferred.

TESTING

- **Testing**. Your README documents the manual tests that you ran. Alternatively, if you decide to use automated testing, instead, you should document how to build and run those tests and the tests should be included when you submit your project. Organizationally, each test should be numbered (or in its own function) and documented re: what it is testing (in the README, or in code comments for automated tests).
- **Testing via Valgrind**. Valgrind should report no errors when your proxy is run under valgrind. Valgrind will help you discover many pointer errors and problems with dynamic memory.

Here are some things you should consider:

- Your proxy should handle Full Requests; it *does not* need to handle “Simple Requests” (see RFC 1945, Section 4.1).
- Your proxy *should* handle GET requests for HTML files
- Your proxy *should* handle GET requests for binary files / images
- Your proxy *should* handle GET requests in the following situations:
    - Proxying to a different port on the same machine
    - Proxying a request to remote machine on a non-standard port
    - Proxying a request to a remote machine on a standard port
- It should handle: large requests, large response, split requests, zero length response.
- It should also handle: premature client close, abusive client requests (malformed requests, infinitely long requests, clients that hold connections open without finishing the request)
- Your proxy *does not* need to handle *persistent connections*. If you receive a request for a persistent connection, you can handle it as if a non-persistent connection was requested.
- Your proxy *does not* need to handle POST or HEAD requests, or *https*.

DIVISION OF LABOR

It is up to your group to figure out a fair division of labor and how to share code. I would suggest that git may be useful, if that is already in your skillset — or a more lightweight solution like putting the tarball into your group's Google Drive to hand it off to teammates. Pair programming during lab is a realistic option, as is a scrum where your group decides how to divide the project into functions that are separately implemented.

One useful role may be to design tests for the proxy. Another useful role may be to configure and stand-up a proxy to have a reference model for behavior. Another useful role may be to review the group's code while carefully reading man pages, with the goal of improving error checking.

RUBRIC

The below scoring rubric helps you identify the expectations and point breakdown for the lab.

Item	Pts
The elements of the specification are satisfied and implemented correctly.	35
Functions are used and documented following a standard convention, e.g. Google Style Guide	15
Testing: You have some description of tests that you either manually set-up to run or are written as a series of unit tests. Valgrind reports no errors.	20
Passes provided tests: Tests will be released that you can run on your own code. You get a number of points proportional to the tests you pass.	20
Passes extra tests: Some tests (aligned with the specifications) that were not released will also be run. You get a number of points proportional to the tests you pass.	10

SUBMISSION

There is no written report. This lab is due 4/1`3/18 at 11:55pm via Moodle. Your group should submit a single tarball (.tar, or .tgz) holding:

- All of the source code for your proxy
- A Makefile that builds your proxy
- A README.md file written in Markdown, describing your code, describing the design decisions that you made, and describing your testing (or how to build and run the unit tests that you've provided).