CS 112 - A1: HTTP Proxy

Due: Wednesday, September 24 - 5:00pm EST

Objectives:

- Introduction to HTTP and Proxy Caching
- Introduction to Socket Programming

Introduction

HTTP is one of the most popular application layer protocols; it runs on top of TCP, which is the standard transport protocol used by most applications. This assignment will give you insight into how HTTP — and, more generally, the application layer — makes use of TCP through the socket interface. It will also introduce you to the potential benefits and challenges of using HTTP proxies.

Description

Your proxy will relay HTTP messages between a client and server, and cache the response to facilitate faster load times. HTTP messages typically have two parts: the header and the content/body, separated by an empty line. After the first line which specifies HTTP method and server path, the header consists of various fields, each on their own line and in no particular order, that describe the HTTP request and content it is providing. Because this is a simple proxy, most header fields should be ignored. For request headers from the client you should focus on these fields:

```
GET http://www.example.com/some/path HTTP/1.1
...
Host: www.example.com
...
```

If a non-standard port is used to connect to the server it is appended to the domain in the `Host` field (ex: Host: www.example.com:8000), making it a good resource for making a connection to the server. The headers in the server response will contain:

```
HTTP/1.1 200 OK
...
Cache-Control: max-age=N
...
Content-Length: 500
```

Of course, if there is no caching policy the `Cache-Control` header will not be present (more details on this in Part II). You do not need to do any special handling for any of the numerous HTTP status codes. For example, if the resource the client is requesting is not available, the first line of the response will probably read, `HTTP/1.1 404 Not Found,` but your proxy does not need to handle this any differently.

Part I: Simple Proxy

- Your proxy will listen on a port for client connection requests -- the port will be passed as the first and only command line argument.
- It will handle one client at a time: it will serve a client request, disconnect, and then wait for another connection request. Once a client establishes a connection on that port, your proxy will wait for an HTTP request to forward to a server (or serve from the cache, as in Part II). After the request has been forwarded and the server's response returned to the client, your proxy should close the connection with both the client and the server. (This is called a non-persistent connection, and while it is not the default in HTTP 1.1 it will simplify things for this assignment, so you should assume that connections are non-persistent)
- Your proxy is only expected to handle GET requests.
- If the port number for the server is included in the HTTP request it should be respected, otherwise port 80 can be assumed. (Note that this feature is essential for your proxy because we will be testing it with our own servers, which will be running on different ports, so your proxy MUST support this feature in order to pass any automated tests that we do).
- Your proxy should run indefinitely. It should keep waiting for more connection requests from clients once done with a previous request.

Part II: Caching

Caching allows for recurring data to be returned to the client faster and for decreased load on both the network and the origin server. This is especially true of what you are building, a *Shared Proxy Cache*. For example, if this proxy serviced the entire Tufts network, it could serve one cached resource to thousands of users, dramatically reducing the overall traffic on the network and freeing up bandwidth to the users requesting new data.

The requirements for your proxy's cache are:

- HTTP 1.1 introduces a new HTTP caching field in the header of server responses called `Cache-Control`. The `max-age=N` directive of `Cache-Control` provides the `N` number of seconds from now that the data being served remains fresh. You need not implement other values for `Cache-Control` (`no-cache`, `no-store`, `private`, `etc`).
- Server responses that specify no `Cache-Control` or when `max-age` is not specified should be cacheable and the content is considered fresh for **one hour**.

- Your cache should also modify the HTTP header in the response to the client to include the current `Age` of the data, as it will be one of the ways that we test your cache. `Age` is the number of seconds since the data was first stored in the cache. If `0 < Age < max-age`, the data is considered fresh. If `Age >= max-age`, the data is considered stale. `Age` should be present in the HTTP response header only when the data is served from the cache. `Age` could be put anywhere in the HTTP response header as long as it is after the first line.
- Your cache should store the responses of the 10 most recently GET requested URLs.
  - If the data for a new request is in the cache and is fresh, the proxy should return the cached data to the client without passing the request to the server.
  - If the data for a new request is in the cache but is stale, your proxy should refetch the data from the server, forward the response to the client and also update it in the cache.
  - If the data is not in the cache the request should be forwarded to the server, and the response should be added to the cache and forwarded to the client.
  - If the cache already has 10 other objects stored, it should prioritize purging Any single stale item. If none are stale then it should remove the least recently accessed one. (Note: in this assignment, FIFO policy is not applicable since all the existing items in the cache are considered retrieved at least once. This is because in this assignment, every "PUT" in the cache originates from a "GET" request from a client.)
- You can assume that the name of the object being cached will be less than 100 characters. Similarly, the maximum size of the object being cached will be less than 10MB.
- Your cache should consider different ports on the same domain to contain different content and cache them separately (ex: GET www.example.com:8001 != GET www.example.com:8000). As in Part I, if a port is not specified in the HTTP request then port 80 can be assumed.
- You do not need to worry about any query parameters at the end of your URL. Assume for this assignment that any data the server needs to serve a GET request will be included as part of the URL path.
- You should cache the server's response in its entirety. Do NOT only cache the headers or the content (Normally proxies don't cache headers, but caching headers will make it easier for you).

Building and running your code

Your code should eventually run on the servers designated for this course, although you may use other environments to build/test your code as well. The course servers on which you will write your code are protected from the outside world by a firewall. Your proxy will not be visible from the public internet. To test your proxy, you must do so from a point within the Halligan network.

Your code can be divided into multiple files. Please use the provided Makefile to compile your code. To get the Makefile, use the following command.

```
cp /comp/112/uploads/Makefile .
```

This will copy the Makefile in your current working directory. The Makefile will help link your code against the network services library (Please do not use any other library as we will be using the same Makefile to compile your code). To compile your code, simply put the Makefile in your code directory and issue the make command

```
make
```

The make command will generate `a.out` executable. Start your program as follows, replacing 9020 with one of your port numbers:

```
./a.out 9020
```

At this point, your proxy service is running.

If you are connecting remotely, you can also use the command-line wget and curl to test your proxy.

Testing your proxy

Testing GET via your proxy can be largely done using cURL's `-x` argument on the command line. Because HTTPS connections won't work, you should use websites that do not require a secure connection. Unfortunately for us (but luckily for security), finding non-HTTPS sites in 2022 can be difficult.

Here are some non-SSL web pages:

http://www.cs.tufts.edu/comp/112/index.html

http://www.cs.cmu.edu/~prs/bio.html

http://www.cs.cmu.edu/~dga/dga-headshot.jpg

(Your proxy should be able to handle simple web pages/responses like the above -- we won't test it with more complex web pages or servers that support other features, such as chunked encoding or scenarios where web pages may not be available or have been relocated).

We also suggest that you write a simple web server to test out various features of your proxy. Your server should be able to reuse a lot of the code you would write for the proxy, so it won't be too much additional work. Note that you won't be required to submit your server.

A good way to verify that your proxy is transferring the objects properly is to take a `diff` between the object transferred via your proxy with the original object (transferred via the direct Internet).

Where to Work

We have six servers dedicated for COMP 112. Inside the departmental firewall (i.e. from a machine in Halligan, or from a public-facing server), please ssh to one of the following machines:

- comp112-07.cs.tufts.edu
- comp112-08.cs.tufts.edu
- comp112-09.cs.tufts.edu
- comp112-10.cs.tufts.edu
- comp112-11.cs.tufts.edu
- comp112-12.cs.tufts.edu

You may only work on your proxy from these machines.

As you know, every service runs on a port–a number between 1 and 32767. You are assigned ports that are yours alone. Please see the Piazza post for your assigned ports.

When you test your services, you must stay within your own port range. Two services cannot share a port, so if you steal someone else's port, their service will behave strangely. Please be kind to your classmates, and do not steal their ports!

Submission and Lateness Policy

Bundle all your code files using tar. For example, if you have three code files (main.c, proxy.c, proxy.h), you can use the following command on linux to bundle.

```
tar -cvf a1.tar.gz main.c proxy.c proxy.h
```

Submit your code by using provide:

```
provide comp112 a1 a1.tar.gz
```

We have prepared a few simple test cases to automatically test your program when you submit via "provide". The intent is to help you check if the basic functions are performed correctly. You should still do more extensive tests by yourself to make sure every requirement in the descriptions is met. Unlimited submissions are allowed, but we will only keep and grade the last submission.

Please see the course policies on late submission. For circumstances that you believe warrant an additional extension, speak to Prof. Dogar.

Useful References

- http://www.linuxhowtos.org/C_C++/socket.htm
- http://www.cs.cmu.edu/~dga/15-441/S08/lectures/03-socket.pdf
- **Simple TCP Client:** https://www.cs.cmu.edu/afs/cs/academic/class/15213-f99/www/class26/tcpclient.c
- **Simple TCP Server:** https://www.cs.cmu.edu/afs/cs/academic/class/15213-f99/www/class26/tcpserver.c
- https://linux.die.net/man/1/curl
- https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
- Cache Control: https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching#Controlling_caching
- A common issue with sockets ("address already in use"): https://stackoverflow.com/questions/15198834/bind-failed-address-already-in-use
- Book chapter on HTTP

Things to Pay Attention To:

- CURL is your friend. I would recommend using it at the beginning without the proxy just to get familiarized with the request-response syntax (headers and bodies).
- Pay attention to the way HTTP headers are structured. The '\n', ':', and '\r' characters are important delimiters.
- Make sure your memory allocation is working correctly and that you are not serving unrequested content nor leaving some bytes behind. Requests going through your proxy should produce the exact same response as direct requests (except for the age header, if necessary).

Clarifications From Previous Semesters:

The following questions and answers were copied from the Piazza forum of a previous year's CS 112 cohort.

I am having trouble with buffer inconsistencies, printing "52 http\0" when only trying to print a single character at an index (i.e. buffer[j]).

C strings are defined as char arrays with a null terminator at the end. It depends on how you're printing, but it's likely that buffer[j] is "5", and the rest of the string is "2 http\0", so your function is reading until it hits "\0".

Adding current Age to response header

You should add a new line. Check out the following link for more info on how the line should look like: Age - HTTP | MDN

read() only getting partial data

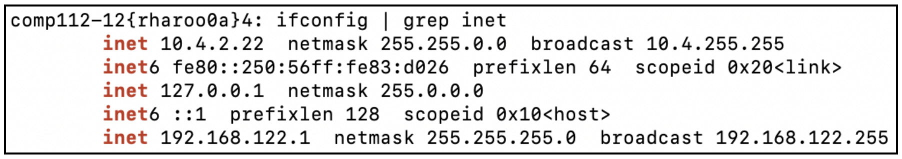
a single read() will most likely not receive the full response. As mentioned in class, you need to call it multiple times, until there is nothing more to be read.

How to figure out my IP address?

You can use the following terminal command on the homework servers to check your IP address:

```
ifconfig | grep inet
```

If you are connected to a Tufts network, the first two digits of your IP address will start with `10`. In the example below, `10.4.2.22` is my IP address.



```
comp112-12{rharoo0a}4: ifconfig | grep inet
        inet 10.4.2.22  netmask 255.255.0.0  broadcast 10.4.255.255
        inet6 fe80::250:56ff:fe83:d026  prefixlen 64  scopeid 0x20<link>
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        inet 192.168.122.1  netmask 255.255.255.0  broadcast 192.168.122.255
```