Computer Science & Engineering
UNIVERSITY of WASHINGTON

CSE461 Project: HTTP Proxy

Out: Monday January 26, 2015
Due: Monday February 09, 2015 by **11:59pm**.
Teams Allowed: Yes
Teams Encouraged: Yes
Ideal Team Size: 2

Summary

You'll write an HTTP proxy, capable of both relaying HTTP requests and HTTP CONNECT tunneling. You'll point a browser at your proxy, so that it sends all page requests to your code instead of directly to the page's origin server. For non-CONNECT HTTP requests, you'll slightly edit the HTTP request header and send it and any payload the request might carry to the origin server, and then slightly edit the HTTP response header and send it and any response payload back to the browser. If the browser sends a CONNECT HTTP request, you'll establish a TCP connection to the server named in the request, send an HTTP success response to the browser, and then simply pass through any data sent by the browser or the remote server to the other end of the communication.

Your proxy should be capable of handling the traffic caused by real user browsing. A small portion of that traffic is generated by the user's typing URLs or clicking on links. Much of the traffic is caused by the contents of the pages the user has asked for - both elements embedded in those pages (e.g., images) and Javascript loaded with it can result in many additional HTTP transactions.

We're writing this proxy both for that experience and because we'll use something very similar in a future project - a protocol tunnel. The protocol tunnel will encapsulate HTTP traffic in the Tor61 protocol of the next project, convey the data across the Tor61 network, unencapsulate and send the recovered HTTP to the web origin server, and then do the reverse with the response. This allows us to use browser-generated traffic as the workload in the Tor61 project. (So, this means you should try to build your proxy in a way compatible with adapting it for this future use.)

Implementation Overview

Solution Restrictions

As usual, our goal isn't to have dozens of HTTP proxy implementations, but rather to provide you with a reasonably specific development experience. For that reason, your implementation must conform to these restrictions:

- We'd like you to build your proxy directly on TCP sockets. The language (or libraries available for the language) you use may offer you much higher level functionality - some form of HTTP server is often available, for instance - but you should not use it.
- Your code may buffer entire HTTP headers, in either direction, before sending any portion of the (edited) header on to its destination, but you must not try to buffer the entire request or response. This means your code must *stream* at least the payload portion of the request and response - send it on as you receive it, rather than accumulate it until you have it all.
- You are free to use any implementation approach you'd like (e.g., threads or event-loop). However, your implementation must be sufficiently concurrent that the handling of any single client request cannot substantially delay the handling of other, concurrent requests. (Additionally, it would be nice if your proxy didn't completely collapse if the face of a temporary, very high request rate, but that isn't required.)

Background: TCP

HTTP runs over TCP, and you're required to use standard TCP sockets as your implementation base, so you'll need to know something about TCP. The odds are you already do, from previous courses, but here's a brief refresher. (If you do a web search for "`tcp socket` *`yourlanguage`*" you'll probably find dozens of pages that all contain essentially the same information. Those pages are useful, and you should use them. Note that they often simplify beyond what is acceptable for us, though - they don't usually address timeouts, for instance.)

TCP distinguishes between the "server side" and the "client side" of a connection. The server must come up first, creating a socket to which the client can connect. Imagining for the moment that the server side is up, the client does this:

- create a STREAM socket (the terminology in your language may differ)
- connect it to the server's IP and port
- start reading and writing using that socket

The server side is a bit more complicated. The server needs to wait for clients to connect to it. It uses a special kind of socket for this. Each time a client connects to that socket, a new socket is generated on the server side. That socket is a regular old socket, just like the client uses, and is the one actually connected to the client. This simplifies the server code, because it ends up with a 1-1 relationship between (regular old) sockets and clients, making it easy to keep their data streams separated.

Here's what a server does. (Your programming language may combine some of these steps.)

- creates a STREAM socket
- binds it to an IP address and port
- calls `Listen` to indicate that this socket will be used to wait for incoming connections, not for reading or writing data
- waits for a client to connect to the socket. It can do that by calling `accept`, which is a synchronous, blocking call, or by using an event-loop mechanism. When a client connects, a new socket is created and provided to the server code.
- now handles the client interaction by reading and writing on that newly created socket.

At this point each of the client and server can both read can write from/to its connection socket. Data is conveyed as a stream - TCP doesn't impose any boundaries on the data sent or received. If one end writes some data and the other tries to read some data, the read may return fewer bytes than were sent. In that case, the remaining bytes will be returned in response to some subsequent read.

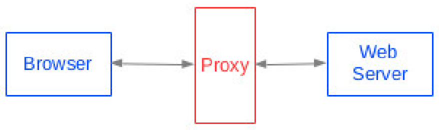
Configuring Firefox

To use the proxy we must configure the browser to send all its requests to the proxy, instead of directly to the web servers. In Firefox you do this using Preferences, then the the Advanced icon, then the Network tab, then the Settings button for "Connection." Configure the proxy manually, giving the host and port your proxy is running on. You should, eventually, allow all types of traffic to pass through your proxy (although we care only about HTTP and HTTPS (SSL)), so you should check the "Use this proxy server for all protocols" check box. It might be easier during initial test to leave it unchecked, though, in which case your proxy will see only `http://` requests.

HTTP Proxying

HTTP is the protocol used to transfer information between browsers and web servers. HTTP is transmitted using TCP as the transport protocol.

An HTTP proxy is a program that can accept and reply to requests that would normally be directed to some web server. Proxies are an example of the use of "interposition" - placing something between two things that communicate using a well-defined interface -- as shown in the figure below. Interposition is a generally useful technique. When possible, it allows new functionality to be injected into existing code with little or no modification to that code. For example, an HTTP proxy might be used for monitoring or debugging (by capturing a log of browser requests and server responses), to improve performance by maintaining a cache of web pages, or to enforce some policy about which sites can be accessed.



The requirements for our proxy are very modest: it merely prints out (at least the initial portion of) the first line of each HTTP request it receives from the browser, then fetches the requested page from the origin web server and returns it to the browser. This means that, for the most part, you don't have to know anything about HTTP; you simply read what the browser sends, print out (only) the first line, and pass that and all subsequent lines on to the web server. On the other side, you read everything the web server sends and pass it back to the browser. You keep forwarding data in this way, in each direction, until you detect that the source has closed the connection.

While that's the basic operation, there are two details that require a bit of processing of the HTTP stream. To make what follows more concrete, here's an example of what Firefox sent when I requested the page `www.my.example.page.com`. (I obtained this by running `nc -l 46103` to set it listening for TCP connections on port 46103, and then configuring Firefox to use a proxy located at `localhost:46103`.)

```
$ nc -l 46103
GET http://www.my.example.page.com/ HTTP/1.1
Host: www.my.example.page.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:26.0) Gecko/20100101 Firefox/26.
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Determining the web server's address

When the browser sends an HTTP request to your proxy, you need to forward it on to the origin web server. You determine the web server by recognizing the `Host` line in the HTTP header. In the example above, the host is `www.my.example.page.com`. You should be insensitive to the case of the keyword `Host`, and you should be tolerant of white space anywhere it might plausibly appear. In general, the host name may be given as `hostname:port` or `ip:port`. If no port is specified, you should look for one in the URI given on the request line (the first line of the header). If there is no port there either, you should use 80 if the transport on the request line is either missing or is (case-insensitive) 'http://' and 443 if the transport is 'https://'.

The HTTP specification says that lines of the header are terminated by CRLF:

```
       CR             = <US-ASCII CR, carriage return (13)>
       LF             = <US-ASCII LF, linefeed (10)>
```

You should be lenient in interpreting this, though. For instance, you might see headers where the lines are terminated by a single LF.

HTTP does not require any particular ordering for the lines of the header, except that the request line (which is always of the general form shown in the example above) must be first.

The HTTP 1.1 specification requires that a `Host` line be provided in an HTTP request (but not in a reply). Your code does not have to work with HTTP 1.0, which doesn't require these lines. (But, I'd guess you'd have a hard time finding a browser that wanted to speak HTTP 1.0 in any case.)

Turning off keep-alive

The HTTP `Connection: keep-alive` line can be used to indicate that the browser (or server) wants to keep the TCP connection open even after the current HTTP request has been fully satisfied. This is a performance optimization: if the browser issues additional requests to the same server within a short time, the overhead of closing the current TCP connection and opening a new one is avoided.

Supporting `keep-alive` greatly complicates the proxy, because it needs to do enough HTTP parsing to understand where one HTTP request ends and the subsequent one begins (and similarly for responses coming from the server). HTTP doesn't have a simple framing mechanism for marking these boundaries. To avoid that, you should filter the request and response streams, removing any `Connection: keep-alive`, inserting a `Connection: close`, and converting any `Proxy-connection: keep-alive` to `Proxy-connection: close`. That should cause the browser and web server to close the TCP connection after each request. Each HTTP request now starts with the creation of a new TCP connection and ends with TCP close, making things simpler for the proxy.

The Tranformed Request Header The final change we make is to lower the HTTP version of the request to HTTP 1.0. This is probably unnecessary, but the more discouragement to using persistent connections we can provide the better.

With that change, the header sent on to `www.my.example.page.com` is this:

```
$ nc -l 46103
GET http://www.my.example.page.com/ HTTP/1.0
Host: www.my.example.page.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:26.0) Gecko/20100101 Firefox/26.
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
```

HTTP CONNECT Tunneling

The HTTP request method CONNECT is used to establish a two-hop TCP connection between the client and some server. HTTP is used only to convey the CONNECT request between the client and the proxy, and to convey a success/failure response from the proxy back to the client. When the proxy receives the request, it determines the destination server (using the technique described above) and tries to open a TCP connection to it. If it succeeds, it returns an HTTP 200 OK response to the client. If the proxy fails to connect to the server, it sends an HTTP 502 Bad Gateway response to the client and closes the connection.

At this point, nothing has yet been sent to the server, all that's happened is that a TCP connection has been established with it (in the success case). None of the HTTP request headers are ever sent to the server. Instead, the proxy simply forwards to the server any bytes it receives after the request header on its connection with the client, and forwards to the client any bytes it receives on its connection with the server. The client may send anything at all it wants on that TCP connection - it could be HTTP messages, or it could be something else completely. HTTPS uses this technique to allow TLS to negotiate a session key between the client and the server. The proxy is simply a conduit for a binary data exchange, and the client and server exchange the same messages over the tunnel as they would over a direct TCP connection with each other.

Sample Output

We show here some sample output. The output is basically a trace of the HTTP request methods and URIs issued by the browser when fetching some page. It is very likely that two requests for the same page will result in different request streams. For one thing, the order of the requests is somewhat random. For another, the components of the page, and so the things fetched, can vary from one page fetch to another. On the other hand, some things must appear in each request trace, for instance, the request for the page itself. The result of this is that it's hard to say exactly what part of these traces your output must include.

The output follows a format that your code also must follow: each HTTP request line output must be preceded by '>>> ' (and your code should print that only for such output, except in the odd case that you're printing some data and the data includes it). Note the trailing space after the '>>>' characters, before the HTTP request line starts. You must print at least the HTTP method and URI given on the request line, but you can also print the entire request line (which additionally includes the HTTP version) if that's easier. The sample output prints only the method and URI.

Finally, you may print anything you want before the '>>> ' tag, and you may print any additional lines you want so long as they don't contain the '>>> ' tag. For example, you may want to print error messages, or even debugging information.

simple.txt	Sample Output
simple.html	Sample Output
fish.jpg *Note that this is moving binary data.*	Sample Output
CSE Winter Quarter 2015 Time Schedule	Sample Output
http://www.cs.washington.edu *Note that you get an unexpected error response, at least when using Firefox. That's expected.*	Sample Output
www.whitehouse.gov	Sample Output
www.cnn.com *Output is truncated to the first approximately 50 seconds.*	Sample Output
http://www.google.com	Sample Output

Turnin

`run` Script

To help us test your code, provide a `run` script that will build and invoke your proxy. The script should take a single argument: the port number the proxy's server socket should bind to, for example:

```
$ ./run 1234
```

Execution is terminated by end-of-file on `stdin` or by ctrl-C.

Files

When you're ready to turn in your assignment, do the following:

1. The files you submit should be placed in a directory named `projProxy`. There should be no other files in that directory.
2. Create a README.TXT file that contains the names and e-mail addresses of the member(s) of your team.
3. The first two lines of the readme file should be formatted like this:

```
Nathaniel Guy natguy@cs.washington.edu
John Zahorjan zahorjan@cs.washington.edu
```

Those lines may be followed by any number of lines giving additional comments or information.

4. Put the README.TXT file, your HTTP proxy solution source code, and your `run` script in the `projProxy` directory.
5. While in the directory that is the parent of `projProxy/`, issue the command `tar czf projProxy.tar.gz projProxy`.
6. Verify that the tar file contains the files you intend to submit: `tar tf projProxy.tar.gz`.
7. Submit the `projProxy.tar.gz` file to the course dropbox.