**Project 1**

80 points for 325, 100 points for 425

(20 extra credit points for 325 students doing the 425 portion)

EECS325/425.

Due: 11:59pm on March 22, 2016.

Enterprise networks often employ a Web proxy as an intermediary between the employee browsers and the rest of the Internet. The proxy can serve multiple roles: block inappropriate websites, monitor employee web surfing, screen incoming pages for malware, and (as mentioned in class) provide a shared cache for Web objects. In this project, you will develop a simplified Web proxy that accepts HTTP requests from browsers, generates the corresponding HTTP requests for the same objects to the origin servers and forwards the responses to the browsers. Some instructions (**make sure to also read tips at the end**):

1. While in reality a proxy usually runs on a separate host, you can develop it on your own machine. To test your proxy, you can run both the browser and the proxy on the same machine (e.g., your laptop). Once everything seems to work, you **must deploy and test your proxy on a departmental machine.** **We will be testing your proxy on the departmental machine. Do make sure it works there.**
2. The machine is a linux server named [eecslinab1.engineering.cwru.edu](http://eecslinab1.engineering.cwru.edu/); you should have an account on it already (use your Case network ID and password). You will need to pick a port on which your proxy will listen for browser requests. Since everyone is sharing the same machine, we need to avoid port collisions. Please refer to the 325 and 425 student lists, find your order number N, and then pick port 5000+N if you are in the 325 list, and 5500+N if you are in the 425 list.
3. Please use Java for this project (to make grading feasible). Please use TCP sockets directly, do not use high-level methods and classes such as URLConnection. (You do not need to use sockets explicitly for DNS resolutions – built-in Java mechanisms are fine).
4. (This is a frequent source of bugs!) Note that when you configure your browser to use a proxy, the HTTP requests the browser sends to the proxy are a bit different than when the browser sends requests directly to origin websites. For example, if the browser requests a URL <http://cnn.com/headlines.html>, the request the browser would send without a proxy would contain the request line “GET /headlines.html HTTP1.1” while the request the browser would send if configured to use a proxy would contain the full URL including the hostname part. This is for historical reasons, from the days when the “Host: “ HTTP header field was not mandatory, to let the proxy know from which website it must fetch “headlines.html” (indeed, perhaps nyt.com also has the headlines.html resource). At the same time, some (not all) websites expect the relative URL (without the hostname) in their HTTP requests, so your proxy **must** replace the full absolute URL with the relative one without the hostname.
5. Your proxy must be able to process two HTTP requests: GET and POST.
6. You do not need to cache Web responses (this is a non-caching proxy that would presumably be used not for performance improvement but other reasons mentioned above). You can assume you only need to support one browser at a time. You can also assume the browser does NOT use pipelining but please allow for the possibility of it using persistent connections (this is actually the default behavior of most browsers). Further, keep in mind that browsers often open several parallel connections – your proxy must be able to handle that. (Tip: use multi-threaded implementation for the proxy.)
7. Please put all necessary files into your home\_directory/project1 on named [eecslinab1.engineering.cwru.edu](http://eecslinab1.engineering.cwru.edu/). We will start your proxy server by first compiling it by invoking “javac proxyd.java” in the “home\_directory/project1/src” directory and then executing the command “java proxyd –port 50025” (for student # 25) from the same directory. **Make sure your program has the above name and accepts the above argument, (and also compiles!).**

**425 only:**

1. To save on DNS resolutions, implement internal DNS caching within the proxy: save your DNS resolutions for future use and before you need one, check if you have the required resolution in the cache. For simplicity, please ignore TTL – just reuse each resolution for the default of up to 30 seconds.

Tips:

* Keep in mind that the browsers often send requests to the proxy over the same TCP connection, no matter which sites they are accessing. The proxy, obviously, will need to open different connections to different sites. But also keep in mind that the browsers often open parallel connections to your proxy.
* Do not use Char[] or String buffers when shuffling bytes between the server and browser sides. Use byte buffers. Remember, the server may be sending binary data (e.g., jpg files). Thus, unlike our example of Java TCP client and server in the slides to lecture 8, I recommend just reading bytes directly from the input stream you get from the socket (without resorting to InputStreamReader), and using the read method that reads some number of bytes into a byte array and returns the number of bytes read or (-1) if the connection has been closed by the other party. See <https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html> and search for “read(byte[] b)” there. Similarly, you can write raw bytes into the OutputStream associated with the socket.
* Since you will be using the standard socket API, your reads are blocking[[1]](#footnote-1): each read call on the associated input stream will only return if the socket has the requested number of bytes or if the socket has been closed (in which case the read will return (-1)). This will require care to avoid your proxy from hanging. When your proxy reads from the socket connecting it to the origin server, the easiest way to handle blocking reads is if you force the server to not use persistent connection. To this end, when your proxy forwards a request from the client to the origin server, make sure your proxy looks for a "Connection" header field in the request and replaces "Connection: keep-alive" with "Connection: close" or, if there is no "Connection" header field in the request, insert "Connection: close".  (Note: some browsers actually insert “Proxy-connection” header field instead of “Connection”. The “Proxy-connection is a non-standard header but your proxy should replace either “Connection: keep-alive” or “Proxy-connection: keep-alive” header field (whichever field is contained in the request) with the "Connection: close" header field. This way, you force the server to always close the connection after processing the request, which simplifies things significantly. In particular, you can now detect the end of response by reading “end-of-file” from the socket (i.e., testing if your read call returned -1).
* However, when your proxy reads from the client’s connection, the above method is not useful because the client may use persistent connection. Thus, you must either check after each read operation if your proxy has obtained a complete request (in which case your proxy must be careful to not read anything else from this socket until the proxy write out the response to the client – else the socket will never have any new bytes to deliver and the read call will be blocked) or you must use multi-threaded implementation, so your proxy can make progress while the read call is blocked.
* Furthermore, remember that when you write something to the output stream, this data does not necessarily go out to the wire immediately: the kernel can buffer it to assemble a larger chunk. This often creates a “hanging” effect, where for instance your proxy sends the last portion of the response to the client but the client never receives it until your proxy closes the socket (which you may want to keep open to support persistent connections). The same effect may occur when your proxy writes a request to the server. The easiest solution, since we do not worry about performance, is to always flush your sockets after every write call (by calling flush() on the output steam associated with the socket).
* To test your proxy, configure your browser to use it (on Safari open the preferences dialog, go to “advanced” tab, then to “Proxies: Change settings”, then check the “Web proxy” button and enter the proxy’s IP address and port number). Make sure you do not check any other buttons (in particular, leave “SSL Proxy” unchecked), otherwise your browser will try to use your proxy for HTTPS connections and your proxy will not be able to handle them. Again, make sure you do this testing using the proxy deployed on [eecslinab1.engineering.cwru.edu](http://eecslinab1.engineering.cwru.edu/).

Deliverables: Well-commented (within source code, no separate class documentation needed) code + README with (a) which port you are using (b) instructions to operate the proxy (c) which browser you used to test it with and (d) which web sites you tested on.

We will be testing your proxy by running it on [eecslinab1.engineering.cwru.edu](http://eecslinab1.engineering.cwru.edu/) and configuring the browser to send its requests to this proxy, and then accessing some web sites.  We will also read your code.

We will use the following grading instructions:

* Proxy handles a sequence of client requests (does not get stuck after the first one) 20
* Proxy transfers text data 20
* Proxy transfers binary data (e.g., images) 10
* Proxy converts absolute URLs to relative URLs 5
* Proxy can load a complex web site, e.g., [case.edu](http://cnn.com/) 15
* Code is logical and understandable, with inlined comments 10
* (425) Implementation of DNS cache 20

1. There is a way to use non-blocking socket channels and “poll” the socket for available data to avoid blocking. However, this is beyond the scope of this class and I would like you to not use these facilities. [↑](#footnote-ref-1)