The team was tasked with creating a proxy server capable of running a basic blacklist as well as censoring vulgarity as well being able cache websites already visited. The project was to be completed in the standard C language. The proxy is to receive an HTTP request for an object and replace it with its own request for the same object from a remote server creating a sterile environment to control the traffic.

The program works on an eleven step plan. Initially the server needs to be started before accepting any incoming requests. Following that, the request needs to be checked against the blacklist. If the site is not found on the blacklist, a new request is recreated. Then, a request is sent for the HTML code and the response is stored in a new file. At this point, the files are parsed through for any vulgarity and have the offending words removed from the document. The header is separated from the doc and the header finally sent back before the body.

The initial caching of the entire website is being handled in the proxy part of the program. The proxy goes through the host website filling up the buffer before sending to a textfile that can be referenced using the URL as the file name. If the proxy discovers that the site has already been cached, it will simply bring up the previous cache as opposed to recreating the file.

One of the first challenges we faced was how to properly filter out profanity. While the list from Hyperhero.com was fully fleshed out, a few problems arose from its standard formatting. Certain phrases such as “Ass master” or “Pain in the ass” were in a more traditional written format, others such as “Bugger, silly” had the offending word listed first. Other scenarios arose when trying to differentiate variations of insults, especially those involving the word “ass”.

For the purposes of this project, the team felt it was important to make sure that each of the phrases was censored in full. In order to ensure this would happen, the list was stored in reverse alphabet order. With this in mind, the parser function would hit all variations of a phrase before hitting its root word. This is the most efficient way of making sure that entire phrases are eliminated

In a real world scenario, numerous things would be addressed differently. First, the list of vulgarity to be censored would need to be truncated. The list from Hyperhero.com may have had the best of intentions, however multiple words simply don’t belong. While they may have been included with the best of intentions, attempting to censor the words “cow” and “dinosaur” from the internet serves no practical purpose. On top of obfuscating websites that would legitimately use the words in question, adding more words to the list of vulgarities to be filtered slows down the entire process. By limiting the reference text file to only pertinent vulgarities, the runtime of the proxy can be diminished.

The blacklist came with its own set of challenges. The team decided the best plan was create another text file featuring the websites to be blacklisted. When the website is originally cached, the URL is set aside in a position where the parsing function can find it and compare it as a string against the URLS in the text file. If the string matches one from the text file, the site redirects to a pre cached page stating that the page is blacklisted.

The problem with this approach is that in its current iteration the website is only being blocked at the exact URL referenced in the text file. This is trivially easy to bypass in practice, and something that would need to be addressed before pushing out a final version of the program.

As with most proxies, this will generally only stop low level users from accessing the content. A dedicated user or one of working knowledge of networking could easily bypass it by using another proxy. In our situation, since the proxy needs to be run manually, it would be even less efficient in its current state. While it can be forced to run through certain browser settings, this can easily be bypassed by using a different browser.

The challenges faced proved to be frustrating for the team, however offered valuable lessons for the team down the road. Being able to completely tare out the HTML of a website and parse for certain phrases not only would be crucial to developing and maintaining proxy servers in real world applications, but these skills could prove invaluable if a site would need to be checked for malevolent code harboring injections or other security risks.