CSCE 3530 Proxy Server Group 4

1 OBJECTIVES

To create a proxy server, written in C++, which meets the following criteria:

- Must cache previously requested webpages
- Must block websites found on a blacklist
- Must filter out profane language

2 CHALLENGES

2.1 KEEP-ALIVE HTTP

Nearly all modern web hosts keep their HTTP connections open for streaming. Because the corresponding sockets are not closed after an entire message has been sent, the recv() function does not automatically return zero to signal the end of a message. Therefore, the only way to know when you've receive a complete HTTP response is to parse the response message's length indicators as you receive them.

In order to avoid this technicality, our proxy includes the **`Connection: close`** HTTP header field in all of its requests to outside servers. It also uses the **`MSG_WAITALL`** flag in its recv() calls. This forces recv() to either retrieve the entire message or to fill the entire buffer.

2.2 INCOMPLETE SENDS

If you attempt to close a socket immediately after sending a large amount of data through it, there is a significant chance that the receiving host will detect the closed connection before receiving all of the data. Google Chrome (and possibly other browsers) responds to this by aborting the page download. Ideally, our proxy should wait for the browser to close the client's socket before it closes its own. We took a simpler path, though, and just added a one second delay between the calls to send() and close().

2.3 CHUNKED ENCODING

There are two types of HTTP response length indicators. The HTTP response header could contain a **`Content-Length`** field whose value indicates the total number of bytes in the message. Alternatively, the header could have a **`Transfer-Encoding`** field. This field almost always has a value of **`chunked`**. In this case, a hexadecimal number follows immediately after the HTTP header, then that many bytes of message data, then another hexadecimal number, etc.

Because our proxy modifies the page's length, it cannot leave the page's original chunk data unmodified. Rather than hassle with chunk computations, our proxy simply removes chunk data and converts all pages to the simpler **`Content-Length`** form.

2.4 SUBSEQUENT HTTP REQUESTS

Once the client web browser receives the html document, it will begin requesting all the additional resources that page links to. Most of these links are relative and so would be sent back to our proxy by default. In order to avoid these additional requests, our proxy adds a **<base>** html element into each webpage's **<head>** html element. This **<base>** element redirects all relative links in the document to the document's original source server, rather than our proxy.

Unfortunately, client web browsers will send some additional requests to our proxy even before receiving the html document. The most notable of these additional requests is for the website's favicon. Our proxy uses the following regular expression to filter out only the requests for actual root web pages:

```
([alnum]+.)+[alnum]+.((com)|(edu)|(org)|(net))
```

Note that this expression only allows root webpage addresses ending in .com, .edu, .net, or .org. Any other kind of request is discarded.

3 PROGRAM OVERVIEW

3.1 CACHING

Our program uses an SQLite3 database to hold a list of previously visited web addresses and their corresponding HTTP **Last-Modified** time stamps. The cached pages themselves are stored as text files in the proxy's working directory.

The database is checked each time a web page is requested. If a website is in the database, a **Get-If-Modified** request is sent to the page's origin server to make sure the cached copy is up to date. If it is, our proxy simply sends this file to the client. If it isn't, the proxy removes the new pages header and transfer encoding data, adds a **<base>** html element, filters out foul language, and adds its own header to the page. It then updates the cache with this new modified page (HTTP header included) and sends the page to the client.

3.2 BLACKLISTING

Blacklisting is accomplished through the same database used for caching. If a website is to be blacklisted, an entry is created for it in the database with the **Last-Modified** field set to the text string '**BLACKLISTED**'. When our proxy checks for the page's **Last-Modified** date/existence in the cache table, it also checks for the **BLACKLISTED** string. If that is found, the proxy returns a simple HTTP Forbidden page.

3.3 FILTERING PROFANITY

Our proxy uses a simple html-aware text filter to overwrite foul language. This filter replaces any occurrence of words in the profanity.txt file with a series of Xs. It is designed to only modify text outside of the html tags, as modifying the tag contents themselves could easily break the page.

4 RESULTS

Some web pages return unusual results. An example is www.cse.unt.edu, which returns a single html **<META>** element instead of a full webpage. More advanced programs (like web browsers) can detect these peculiarities and respond to them. In the case of the CSE webpage, the **<META>** element instructs the recipient to load a new relative URL. We could fix this particular issue by checking for a **<META>** element instead of a full document, and modifying its address to point back to the original server. However, this and many other enhancements, would take too much time.

Other than these types of issues, our proxy seems to work correctly.