Jason Byrd
Benjamin Garside
Juan Hernandez

CSCE 3530 Proxy Server Report

Problem

In this assignment, our group was tasked with creating a functional HTTP proxy server to act as a middle-man in all web requests a user direct towards it. The proxy server was to first receive a request from a user behind a browser, assess the request and pull meaningful data from it, forward the request to the user's destination, grab the response, analyze it, and finally send it back to the user's browser after a little bit of modification. The server was to be built with multi-connectivity in mind - so we were tasked with implementing multiple threading to allow support for simultaneous users on the server. It was also to incorporate a profanity and insult filter, a caching system to save previously accessed websites, and a black-list to completely block users from accessing certain sites through the proxy server.

Approach

This project was more difficult to split into separate workloads than the first was. We needed the core functionality of the HTTP proxy server working before we could have a solid understanding of where to edit the code in order to implement the black-list, the insult-filter, and any functionality related to caching. Because of this, we started by implementing a simple and very basic version of the proxy first. It received requests, forwarded them, received a corresponding response, and then sent the response back to the original requesting machine. All of these interactions were supported by sockets: one to connect the user's machine to the proxy server, and another that connects the proxy server to the HTTP destination. The latter is formed by parsing the HTTP request sent to the proxy, pulling the destination IP from it, and using it to initialize a socket. Each initial request to the server spawns a new thread of execution, so multiple users and simultaneous requests are supported.

One problem we ran into with the implementation was signaling to our response-receiving code that the destination was finished sending data through the socket. If we didn't specify a byte-size to receive from the host, we would receive the response in small chunks, and the program would hang. To combat this, we implemented a timer system within a while loop. The timer calculates a 'difference' variable to represent the time elapsed since our last bytes were received. Combining this with a buffer that receives responses in chunks of 512 bytes allows for the program to receive bits of the response piece-by-piece. Upon each iteration of the loop, the difference variable is recalculated and compared against a threshold, and the loop is broken if the difference exceeds a maximum timeout value set by us. This allowed for us to gather the entire response from the destination server, and to close the socket upon breaking out of the receive loop.

Once the very basic portions of the server were working, we needed to implement the additional functionality laid out in the project description. We began with the insult filter, because it seemed to be the most simple. Upon the program's execution, an array of insults is formed in the system. This array is to be used for comparisons when filtering out insults in the text we receive from the destination servers. We decided to make the actual comparisons and perform the text substitution after the entire response has been received from the server and concatenated into one long string. The string is passed through a function that inspects it for insults. If any are found, they are replaced. This process was done character-by-character to ensure that the original response does not have it's formatting modified heavily. After searching through the entire response and replacing insults, it is forwarded onwards to be sent to the original requesting machine.

The blacklist was simple to implement. In a method similar to initializing the insult list, an array is formed and populated with a list of websites that we do not want to allow user access to. When a request is received from the user's machine, the server parses it and pulls and saves the new destination host (as mentioned earlier). Before forwarding the request onwards to the new host, we compare the cannibalized hostname ('google.com', 'yahoo.com', etc.) to each entry in the blacklisted sites array we generated at execution. If a match is found, we stop the proxy from forwarding the request, and inform the user that access is blocked to the site they attempted to navigate to.

Implementing the cache was a little bit more difficult than the blacklist and insult filter. We needed to create and make use of a database in order to save and load pages that we have already accessed. We made use of a SQL database and corresponding C++ functions to handle all of our insertions and selects. In order to check whether or not the site being accessed has been accessed before, we run the the destination host name against a database table containing our previously requested hosts and their corresponding responses. If the hostname matches a hostname present in the database, we select and pull the associated response data and store it in a variable in code. This string is then sent to the original requesting machine as a response. If the requested hostname is NOT found in the database, a flag is activated. After regular operations continue in the program (a request is forwarded and a response is received), this flag is checked. If it was activated earlier, the hostname and the response received from it are both stored as new entries in the database's cache table. This allows users to effectively skip the second round of request/response interaction, and load previously accessed websites straight from the database.

Solution

After implementing the functionality described above, the proxy server sends and receives most HTTP requests and responses normally. The flow of data through the system is modeled correctly, and all checks and modifications to the response code occur in the correct places. In theory, everything implemented in the code should work near perfectly. In practice, we receive different and varied results. It seems that at some point in execution, usually after a few requests have been sent and received, some of our data streams becomes corrupt, and

some of the string variables we create from that point onwards have garbage memory data appended to them. This throws off any forwarded requests, because the destination server won't recognize the garbage text near the end of the requested data. We thought it might be related to null terminators, a leak in data between threads, something going wrong with the socket stream, or memory initialization errors; but after hours of experimentation and investigation into each potential problem, we were never able to discover what causes this text corruption to occur. Before the corruption happens, however, the system handles sending and receiving the packets perfectly well, with the exception of image files, which can not easily be stored and passed as character arrays, because they exist as binary data.