Nickolas Gravel
Amy Bui

CS 112
Fall 2022

High Performance HTTPS Proxy

I. Introduction

A proxy is a computer or software system that acts as an intermediary between requests made by client entities and the end host services that contain the requested resources. An HTTPS proxy, in particular, satisfies HTTP(S) requests on behalf of the requesting endpoint by sending the request messages to a web server. The proxy receives the response object from the services and then forwards it back to the requester. Why is a proxy necessary, when the requester could simply make the request to the service themselves? Proxies are more than just a simple forwarding service between endpoints. There are many types of proxies used in networking services today and they include features that protect users' identities, filter content based on a particular network's preferences, encrypt communication packets to provide security, control bandwidth and improve performance, etc. We implement a basic HTTPS proxy that handles forwarding of HTTP request/responses for two kinds of HTTP methods, and then build upon its functionality with additional features. Section 2 details the development environment we worked in and how we divided up the tasks; Section 3 details what features were implemented and which ones made a significant impact on the proxy's performance, if any; Section 4 discusses the results of our performance testing without the proxy, with the basic proxy, with the proxy caching, and with the additional features, and implications of the results; and Section 5 will discuss the process of implementing the proxy and issues that arose.

II. Development

A. Environment

A Docker container was built to provide an environment for the development and the testing of the high performance HTTPS proxy. The Docker container uses the latest build of Ubuntu (22.04) and installs the essential tools needed for the development and testing of the proxy. A container was used in-order to stand-up a virtualized environment on any Linux, Windows, and macOS computer. This helped ensure dependencies remained consistent throughout development and testing.

B. Required Program

OpenSSL 3.1
C99 standard
gcc v11.3

C. *Key and Certificate Management*

To implement support for SSL we must first generate our own *root* certificate and private key. This root certificate is then added to the trusted store on our device so that all self-signed certificates will be trusted by this specific device. Becoming our own CA involved installing openssl to our machine and using the two openssl commands, seen below, to generate a key and certificate. After creating our own root CA, we created a Certificate Signing Request for our proxy and signed it with our root CA's key in order to make our proxy an *intermediate* certificate authority. At this point, the proxy used scripts we wrote to continually update the proxy's certificate with any hostname given now able to ad sign certificate signing requests (CSRs) so that they can be trusted. (1) generate a private key, (2) create a Certificate Signing Request (CSR) and send it to a CA, and (3) install the CA-provided certificate in our proxy server.

Key Generation

Generate an RSA key

```
openssl genpkey -out fd.key -algorithm RSA -pkeyopt rsa_keygen_bits:2048
-aes-128-cbc
```

D. *Division of Work*

The new proxy was built off of Nik's original A1 proxy with proxy caching, and test client and test server. We primarily worked together and peer-programmed to implement the proxy's baseline features and get it to a data flow that we both agreed on. The biggest changes from the original A1 proxy were to the Linked List, HTTP (and Response/Request), Client, Query, and Cache objects, whose interfaces were changed, as well as Proxy, since the A1 proxy did not handle multiple clients. For each subsequent major feature that was added, there was usually an agreed upon plan on how to approach it and build it on current proxy, a primary person implementing it, and we tested and debugged the feature together, as most new features added had to be integrated at one or multiple points of the previous data flow. Nik's primary features were the robust error handling and SSL connection handling; Amy's primary features were cache availability hinting on hyperlinks and efficient client socket selection.

III. Features

A. *Basic Features (Phase 1)*

1. Support of GET method requests
   Proxy handles parsing of clients' GET requests and forwards the request to and response from end hosts. The proxy assumes these requests are non-persistent and closes connections upon serving the response.

2. Support of CONNECT method with HTTP tunneling

The proxy handles parsing of clients' CONNECT requests. On a CONNECT request, the proxy opens a connection with the target host on behalf of the client. From there, any communication received from the client socket will be forwarded to the target host socket by the proxy, and vice versa. The tunnel remains open until one of the ends closes. Responses through this communication method are not cached.

3. Handles multiple clients
   The proxy maintains a list data structure of currently connected clients and their end hosts. It uses select() to monitor multiple client-side and server-side sockets. On a socket-ready-read, the proxy extracts a low number of bytes from the buffer stream (~1 kibibit) at a time, to try and ensure fairness and equal allocation of time for each client. Since the proxy only handles GET and CONNECT requests, the proxy detects when a full request is received from the client using the standard HTTP header format and determines when the request can be forwarded to the end host given in the request header. Likewise, the proxy only caches and forwards responses once it detects the entire response is received; this is signaled by a return value of 0/EOF from recv() or if the number of bytes received match a given Content-Length, if any is given in an unencrypted response header.

4. Error Handling
   The proxy has robust error handling and reports where at any point the program fails in the proxy or where a request/response may be ill-formatted. Under the "Error and Event Handling" section of config.h, there is a list of different internal error flags defined by the proxy and what they mean, including failures from library function calls, failure from a proxy method call, invalid request/response format, premature socket closings, and other intermediate states of our data flow.

B. Additional Features (Phase 2)

1. Caching
   The proxy caches unencrypted resources received from any end hosts it receives before forwarding back to requesting clients. The proxy cache is implemented as a hash table (linear probing) for an amortized O(1) time access, using host, uri, and port as the key. Resources re-requested by a client or previously cached resources requested by a different client are available to send to clients without the overhead of connecting to end hosts and re-requesting popular resources. This significantly reduces the response time to the client and improves the performance. The proxy's cache uses a least-recently-used (LRU) cache replacement policy and staleness replacement policy. As how it is implemented in A1, the idea is that each cache record (we called "Entry") carries time data, namely, the time that it was inserted. When one entry must be evicted, the time

since insertion is calculated as the age of that cache entry and checked against a set max-age in which the entry was allowed to remain “fresh” in the cache; this max-age value can from the response if given, or was a default value selected for all entries. If no entry was found to have been stale, then the LRU policy kicks in; the cache takes from the front of an ordered entry queue, where the most recently used record is at the back of the list, and the least recently used entry is at the front.

2. Optimization: Check Current Clients Only and Establish Client Timeouts
   Client objects are stored with the socket file descriptor that communicates with the proxy. Upon a read file descriptor return from select(), instead of looping through socket numbers up to the fdmax and calling FD_ISSET on each integer, we check only those client sockets that are currently connected to the proxy. Client objects are removed from the proxy if they close or proxy ends communication or they timeout after a set amount of time, so those file descriptors are never checked again even though they are in range [1, fdmax]. However, the time complexity itself is still only improved by a constant amount rather than linearly; so the effect on performance would only be seen as the number of clients *ever connected* to the proxy increases.

3. Hyperlink Coloring Based on Cache Availability
   Responses that can be seen by the proxy are edited in such a way that changes the colors in any hyperlinks in the resource based on whether or not the resource is in the proxy’s cache. The links are red if the proxy does not have it cached, and green if it is. After obtaining a copy of the entire response in a response buffer, the proxy parses it for the open anchor tag followed by an href attribute that starts with “http”, i.e. `<a href=”http`. This would assume most resources with hyperlinks are html and could be colored. There was not enough time to parse other kinds of resources to parse for hyperlinks, e.g. PDF format specification for hyperlinks.

   The program takes a duplicate of the response because in addition to changing the html, it also adds an Age field to the header if the resource is served from the cache; this ensures that the original resource is preserved and ensures that these edits don’t accumulate with each service from the cache. Using the duplicate, the proxy copies each byte to a new buffer until it encounters the anchor tag mentioned above. Using a list of keys that correspond to resources currently in the cache, the program sees if any cache keys match the href value in the anchor tag, i.e. checks if the hyperlink matches a cache key; cache keys are formatted as host followed by the URI from the original request that put it in the cache. The proxy looks for a “perfect prefix” match on the key, because keys could also contain port numbers. If the hyperlink is a prefix of a key, but the key itself has additional elements in the URI, that is not a match because it is a different resource than that of the hyperlink’s, but may come from the same host.

If the hyperlink in the resource matches a cache key, then the proxy inserts a style attribute with green color CSS right after the open anchor tag; it inserts a red one if there was no match. It repeats this algorithm until it reaches the end of the original resource buffer.

As you can imagine, this feature implemented this way, is not very efficient in terms of time to serve the client the resource. The time to complete a request for a resource not in the cache was not that much longer than serving a resource coming from the cache now. While serving from the cache means the proxy did not have to spend time setting up a TCP connection with the end host to retrieve the resource, the impact that color parsing links has could reduce performance than if we did not provide coloring links based on cache availability. Inserting color essentially has the additional time requirement of copying the response buffer one additional time, which has a constant penalty on the time complexity of the proxy, so performance is penalized as size of the resource increases. If performance was a larger concern for clients, the proxy offers the option to turn coloring links on or off.

4. Content Filtering
   For a given list of "flagged" hosts, the proxy will not serve the client's request. The filter list is an existing list that exists in the program director. When the filtering feature is turned on, and the proxy receives a request, the proxy checks the host of the request against the filter list. If the end host matches something on the list, the proxy does not request the resource from the server and does not serve the client. Again, for the way this feature is implemented, the runtime is only linear to the size of the filter list, which, in a real life scenario, is likely not to exceed the size of any of the resources the proxy server. Since the runtime of the proxy is heavily biased towards size of the resources and operations such as establishing connection, the filtering feature does not impede performance.

5. Trusted Proxy/SSL Interception (bug): See Section 5.A. This feature is present but turned off for our proxy for normal use, as it did not pass our tests with consistent results.

IV. Results & Discussion

A. Testing Method

We set up an automated testing script that runs curl and a standard http-client as the request senders to our proxy. Using the system's `time`, we recorded the amount of time in seconds it took for a client to receive a response after it sent a request. We measured the time to receive a response directly from the end hosts (i.e. without proxy), with requesting through a basic proxy,

and through a proxy with additional features turned on, i.e. the caching, optimization, coloring. and SSL handling. Our analysis of the proxy's performance will only be based on how closely we measure the round trip time. Due to time constraints and limited resources, we are not able to control for other effects on performance, such as congestion, packet drops/losses, bandwidth, etc, so we assume they do not factor into our measurements. We tested using end host services that were reachable and transferred data back in TCP stream, i.e. those that responded with a Content-Length header field so that we could normalize the response time from different end hosts. For something like Google.com, which uses chunked transfer-encoding, which our proxy did not implement for unencrypted communication, we did not test with those hosts.

Time to serve a response directly to the requester directly correlates to the size of the response, which is expected. We try to normalize these differences between different end hosts by reporting the rate bytes/sec; however, it should be noted that there were still significant differences between the individual normalized rate in which entire responses from different services returned to clients. It is likely that these variances originated from other factors affecting performance that are not controlled for, including congestion at the end host, which network they operate in relative to the machine we are testing on, and latency. The results in Fig. 1 and Fig. 2 show the average rate across different end hosts for each method of requesting a resource.

Note that results displayed in Fig. 1 for proxy with caching feature are those measured for responses served from the cache; Fig. 2 shows scenarios with caching turned on, but responses from CONNECT http tunneling are not cached.

B. Performance

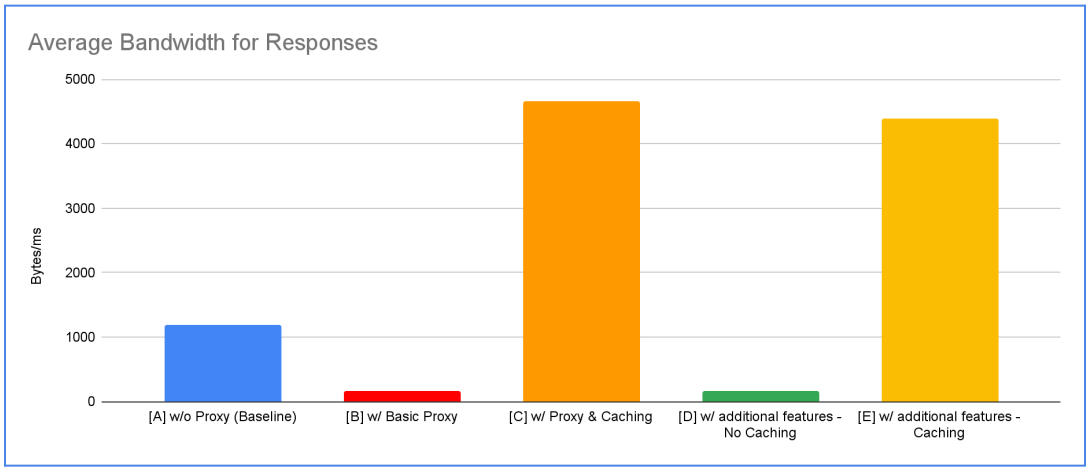

Figure 1: Normalized average rate of responses for client GET requests for multiple end hosts.

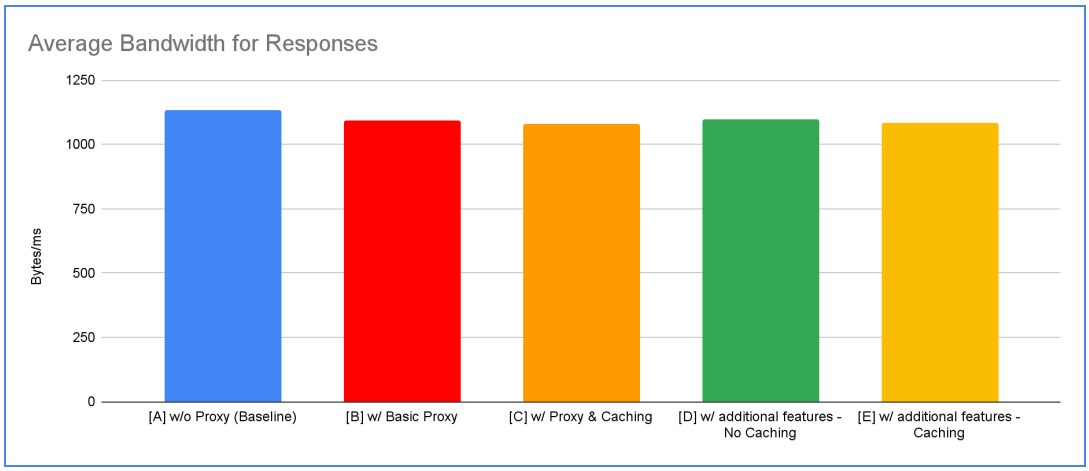

Figure 2: Normalized average rate of responses for client CONNECT requests for multiple end hosts. (Note that encrypted resources were not cached. For the same tests done in Fig. 1, we tested with caching feature turned on and off).

1. Baseline
   Baseline time measurements were recorded using curl. While differences against rate w/proxy was only slightly better on average, the difference was not as significant as the rate when proxy was serving from its cache.

2. Basic Proxy
   As expected, the time to serve response using the proxy with only the basic features was comparable to the baseline response time if the client had made the request without a proxy. The differences were on the order of hundreds of bytes per milliseconds. The small difference may be attributed to the overhead of setting up and running the proxy handling request/response buffers, in addition to the work involved in setting up TCP connections. The work involved in baseline behavior is just setting up TCP connections and transmitting and receiving bytes of data.

3. Proxy with Additional Features
   There is a noticeable improvement in response time when the request is made through the proxy with additional features turned on. The time differences were consistently on the order of hundred of bytes per milliseconds; however, this was only with the caching feature. When the proxy operates with or without the cache, the other additional features discussed above did not make a significant impact on the performance, which can be seen by the comparable rates between the red (basic proxy) and green (basic proxy w/ additional features, no-caching) bars in Figure 1. The corresponding rates in Fig. 2 does not allow for caching of encrypted data, so they are representative of proxy performance without caching.

   The proxy's performance with caching, and with (orange) or without (yellow) the other additional features (Fig. 1) were both also not significantly different from each other, but

were each almost 28 times as fast as the non-caching proxy for serving responses, and 4 times as fast as the baseline. This shows that for our proxy, caching responses was the most significant feature that improved performance of the proxy's response time. Furthermore, this validates the claims about the effects of the different additional features on the proxy's performance stated in section 3; namely, excluding caching, each feature had a O(1) time penalty on the proxy's performance, i.e. no significant effect unless the size of the response and number of clients connected were large enough. Due to limited resources and memory storage, we were not able to approach such limits such that the proxy's performance was observed to be negatively affected.

Whether serving from the cache or serving a response the proxy got from an end host, the common behaviors from the two data flows are receiving, handling, parsing, and writing some buffer. The difference is in one scenario, the proxy is getting a response buffer from a locally implemented cache, while in the other, it must set up a TCP connection in order to communicate with the end host before receiving the response buffer. Since performance was significantly impacted when the proxy does need to request the resource directly from an end service on behalf of the client, this implies that the overhead of establishing a TCP connection to the end host is a significant burden to proxy performance, and finding ways to lessen the responsibility or need for the proxy to open these connections would further improve the response times.

C. Remarks

Analysis of the results concluded that reducing the penalty from setting up a TCP connection to get a resource would significantly enhance performance. In the case of the caching feature, the proxy did not have to re-establish a connection to a server it had already connected to in the past because the resource was already available locally. We do not look to remark on proposals to change the TCP protocol on how connections are established as they are beyond the scope of the current subject. At the level of the proxy's implementation, there are more additional features not implemented here that could target this area of improving performance. For example, the proxy could keep persistent server-side connections so that it doesn't close the connection upon receiving the response it was looking for. The proxy may either pre-establish connections to popular service hosts before it waits to accept clients, or maintain connections from services in which connected clients are actively requesting. This allows current requesting clients as well as other clients to reuse an already established connection to a common target host so that the proxy would not have to consistently connect to the same host over and over again for the same or different resources from some popular host.

V. Project Discussion

A. Bugs

The following are known buggy features in the proxy implementation, some can be turned off:

- Trusted proxy/SSL handling - We used the open source library OpenSSL to incorporate TLS/SSL handling. In our testing, there were inconsistent results of successful connections establishing and responses being received. Another common error that sometimes appears is with wrong TLS versions. A lot of time was spent configuring the work environment to have the correct certificates for the TLS handshake to complete, so the source may be in the implementation on how end of responses are handled over SSL communication.

B. Lessons Learned

1. **Modularity**: We initially set out to make the program as modular as possible in phase 1, anticipating that integrating additional features afterwards would be made more complex if it were not. A feature like coloring response hyperlinks was more easily integrated because we could pin point specifically where in the data flow and in which module that feature would fit. However, integrating a feature like SSL handling was more complex despite our initial efforts to make the basic proxy as modular as possible, because we actually found that SSL communication handling had to be dealt with at multiple different points throughout our program to try and make it work without breaking the existing basic features. Introducing more complex features had us deviate from the original architecture we chose, and we almost anticipated old features breaking with additions of adding new features. We found that an important step was not only committing to the initial design, but also maintaining clear barriers between interfaces even after making changes to the architecture in order to add one new feature. Going forward, we found that a good implementation strategy to make adding new features easier than how we did it would be to be stricter in deciding if the new features had to be integrated in the existing modules, and favor constructing new modules for newer features and limiting the amount of data exchanged between each module to be as minimal as possible, as our current implementation, for example, often passes around a Proxy struct that carries a lot of extra data that most method do not use. Our initial reasoning for doing so had been a “just-in-case” future feature that would need those data.

2. **Time-Constraints**: There were certainly a lot of features we wanted to implement but did not have the time to get to. Two of the main limiting factors were phase 1 taking longer than expected and having to refactor the code as we were adding new features in phase 2, and spending a lot of time and being blocked on the feature for handling SSL connections. We initially thought it would take less than a week to essentially add multi-client handling, error handling, and CONNECT method to the existing A1 proxy,

and that would complete phase 1; however, integrating all the basic features at once to A1 had involved significant changes to the dataflow and affected multiple existing interfaces more than we had anticipated. Debugging it and also fixing new bugs in the basic feature with each new additional feature in phase 2 took a lot of time. While it may not be reflective of real-world situations, if the basic features aspect of the project was released around A1's completion, it would have helped for students to slowly integrate the basic features themselves to their own A1s as multi-client and error handling were introduced in the following assignments; groups that decide on the high performance proxy project would have a version of A1's architecture that somewhat handled basic features without perhaps needing significant changes to the dataflow that we encountered in the beginning of phase 1. Also, if we were to also do this project again, we would have reconsidered starting with SSL handling in phase 2. We knew it would take the most time to learn how to configure the environment so that OpenSSL would work in the docker image, integrate the feature, and debug it, but the the blocks here took up most of the time in the phase 2 week that could have been spent implementing and testing easier additional features.

3. **Collaboration**: This was definitely not a project that could easily be done solo in a short amount of time. Discussing ideas, sharing resources, and ways of implementing different features with groupmates and other groups immensely helped, especially when we were blocked on a problem. Open discussions of ideas also helped pick-up wherever someone might have had a conceptually incorrect idea about a particular feature or implementation and was important in avoiding otherwise time-consuming and frustrating sessions. While there is an array of resources (article, blogs, stackoverflow, etc) on the internet on how to approach ideas or search for answers, we found that more likely than not, we came upon resources that were unorganized, not straight-forward in addressing particular bugs we had, or not robust enough of documentation (ex. the error codes "defined" by the OpenSSL library). As assignments such as these grow in scale and complexity, we feel it is better to be able to see people in person and discuss the different pitfalls different groups found and the viability of potential solutions.

VI. Conclusion

We observed that a limiting factor on performance and therefore user experience in the process of requesting services through a proxy was the overhead of starting the existing transport protocol, TCP. As an established protocol that is widely used, the intuitive approach to increase performance would be to either implement a modified/new protocol or find ways to reduce connection setups. Since the prior is not easily done outside a backend scenario, because the proxy is interfacing for the client with the open internet, the later is a better option to explore for future improvements. We've demonstrated one method of reducing the load of establishing connections (and re-establishing them) by caching previously requested responses. We observed that with this feature, and even with the work done by additional features the proxy has that only have a constant added time complexity, it serves clients resources faster

than a proxy without caching and faster than the baseline. Response time is only one area in which a user would benefit from requesting services through proxies, as proxies as a technology offer a more diverse set of benefits. Performance happens to be one that was explored here as having one of the larger impacts on user experience. This project also demonstrated an appreciation for proxy implementations themselves, as they are required to appear as if users are making these service requests themselves while also yielding the benefits of the added features to a proxy, because a proxy should be able to handle scenarios experienced with the http protocol that are already handled when users are browsing the internet.

VII. Acknowledgements

We would like to acknowledge the support of the Networks & Protocols professor and course staff, Fahad Dogar, Jieling Cai, and Hiba Eltigani, as well as our classmates Matt Zhou and Chami Lamelas, with whom we collaborated.

VIII. Additional References

[1] Kumar, Amlendra. "ssl server client programming using openssl in c". AticleWorld. 2018. Web. December 2022.

[2] OpenSSL. "Simple TLS Server". Wiki. May 2015. Web. December 2022.

[3] Ristic, Ivan. OpenSSL Cookbook. Feisty Duck, 2013. https://www.feistyduck.com/books/openssl-cookbook/