COS 316 Final Project - "HTTP/HTTPS Web Proxy and Cache"

Collaborators: Aylin Hadzhieva(ah4068) and Kok-Wei Pua (kp7662)

Inspired by Stanford's CS110 on "**HTTP Web Proxy and Cache**," our team has implemented a HTTP web proxy and cache in GoLang.

1. **Motivation:**
   We were inspired by our lectures on network protocols and have a keen interest in building a practical application in this area. We wish to use the knowledge we learned in lectures, including networking concepts and caching mechanisms, to implement a useful application.

2. **Project Goals:**
   The goal of our final project is to create a HTTP/HTTPS web proxy and cache in GoLang. The proxy will act as an intermediary for HTTP/HTTPS requests, controlling the flow and content of data between clients and servers.

   We aim to challenge ourselves by implementing a system not extensively covered in lectures. By conducting independent research and being creative in making design decisions for our system, we hope to enhance our skills and expand our knowledge on this topic. We spent a considerable amount of time understanding the working principles of proxy, the handling of HTTP requests and response headers, caching of HTTP responses, setting up proxy servers and clients in Go, etc.

3. **Design and Implementation:**

   2.1 Forward Proxy: We implemented a forward proxy that intercepts HTTP/HTTPS requests, modifies them as needed (e.g., removing hop-and-hop headers, appending X-forwarded headers), and forwarding them to the intended servers. Our proxy supports basic HTTP/HTTPS methods – GET, POST and CONNECT.

   3.1.1 Handling HTTPS CONNECT requests
   In our original project proposal, we did not set out to handle HTTPS requests. However, during our internal testing, we noticed that many HTTP sites have embedded HTTPS requests to retrieve contents and styles from the destination servers. For instance, the http://go.com/ site fetches its stylesheets over HTTPS, so its layout may not look correct. Without properly handling the HTTPS CONNECT requests, we were unable to load the sites properly. Therefore, we felt that it is necessary to expand our proxy server's functionalities to support HTTPS CONNECT requests so that we can load the full web page for inspection, even though this feature wasn't included in our project proposal. We do this by setting up a tunnel consisting of TCP connections that requires handling of multiple goroutines running concurrently.

3.1.2 Hop-by-hop Transport & X-Forwarded-For

Inspired by this blog, we decided to align our implementation of proxy server with the HTTP protocol by handling Hop-by-hop headers as in standard practice, the proxy should remove them before sending them to the next address. You may learn more about the purpose of these headers here. The X-Forwarded-For header identifies the originating IP address of the client, and we have decided to include it in our proxy requests as it is important to enhance security – it helps in identifying the source of traffic in case of malicious activities or troubleshooting. More information on this topic can be found here.

3.2 Blockset and Caching: Then, we implemented blocking mechanisms to certain websites and caching mechanisms for static resources. We modified the ServeHTTP function to handle domain blocking and implemented a cache module to store responses. The aim is to use caching in reducing network traffic and enhancing performance.

3.2.1 Blocking Mechanism

We implemented the blocking mechanism by defining a BlockedSet structure to store a set of blocked domain patterns. These patterns are represented as compiled regular expressions that are stored in blocked-domains.txt. The NewBlockedSet function initializes this set by reading domain patterns from a specified file, compiling each pattern into a regular expression, and storing them in the blockedDomains slice of the BlockedSet struct. The IsBlocked method enables checking whether a given domain is blocked by iterating through the compiled regular expressions in the set. If a match is found between the input domain and any of the stored patterns, the method returns true, indicating that the domain is blocked; otherwise, it returns false. This design allows us to maintain and update the blocking criteria easily through the input file of regular expressions.

3.2.2: Caching

We implemented an HTTP/HTTPS caching mechanism by storing the cache files in the file system using a directory structure. When a HTTP/HTTPS GET response is cacheable[1], we generate a unique key by hashing the request URL using SHA-1. We then use the key to create a file path within the cache directory 'http_cache'.

We serialize the cached data (including the response body, headers, and status_code) into binary format using the encoding/gob package as this way it is faster, compact and does not require additional parsing. We then write the binary data to the cache file. When retrieving a cached response, the key is used to reconstruct the file path, and the data is read from the

[1]We define a HTTP response to be cached only when it includes a "Cache-control" header set to either "public," "no-cache" or "max-age".

corresponding file. Stale cache[2] entries are removed by deleting the associated file and eventually replacing it with the updated HTTP/HTTPS cache.

We implemented an enhanced caching system named 'cache_lru,' which utilizes the Least Recently Used (LRU) method for cache eviction. It not only eliminates stale cache but also prioritizes the removal of the least frequently used files when the cache reaches its maximum capacity. We measure this capacity by the number of files present in the directory. Given that the data is stored in a compact binary format, we've determined that an ideal maximum capacity would range between 150 to 200 files. Further details on the rationale behind our design choices are discussed in the 'Technical Challenges' section below.

4. **Testing**

We tested the functionalities of our forward proxy by visiting the following HTTP sites and inspecting the Network traffic in the browser when we loaded the sites.

To test GET and CONNECT requests: http://www.gnu.org UN.org Gravatar.com Digg.com This website has a compiled list of more HTTP sites.	To test POST requests: http://ecosimulation.com/testWithImages.html http://ecosimulation.com/testpost.html

Our proxy server should respond to all HTTP requests with a placeholder status line consisting of an HTTP/1.1 version string, a status code of 200, and a curt OK reason message.

To validate the functionality of our proxy, we compare the visual layout of HTTP sites in two scenarios: (1) when accessed through a client browser configured to route through our proxy server (by adjusting the connection configuration settings in Mozilla Firefox), and (2) when accessed through an unconfigured Google Chrome that does not route through our proxy server.

We inserted logs or print statements at strategic points in our functions to verify that specific lines of code are executed under certain conditions. It enabled us to track the flow of execution and identify areas that are or aren't being reached during runtime.

[2]We define a cache entry to become stale if its MaxAge is zero or if its age exceeds MaxAge, indicating outdated data. When MaxAge is unspecified, it defaults to -1, necessitating data validation before client delivery. This is also required under a "no-cache" condition.

To validate the functionality of our caching mechanism, we performed multiple accesses to an HTTP site and examined the visual layout upon refreshing. We introduced a log statement to signal when the HTTP response is served from the cache (cache_lru: and when and which cache file is removed in case the data has become stale or max capacity has been reached). A well-functioning caching mechanism should present the HTTP site identically to accessing it directly by sending a request to the destination server. NOTE: if you want to test cache_lru, swap it with cache_without_lru and make sure you delete the http_cache directory if the proxy is restarted. More explanation on this is in the "Technical Challenges" section.

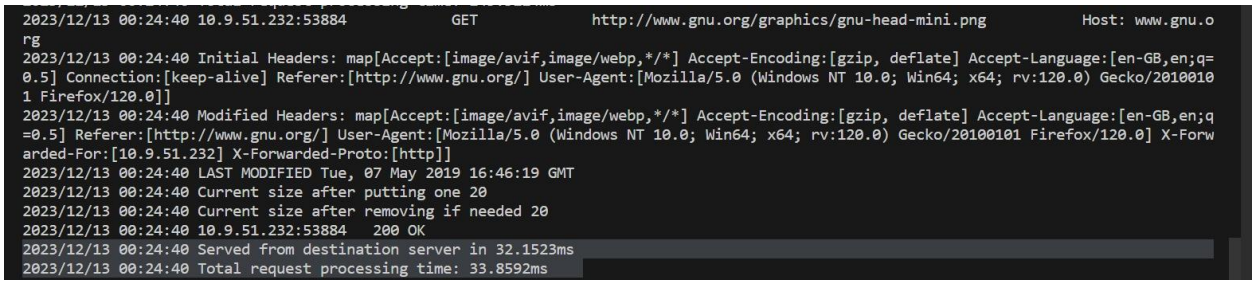
To validate the functionality of our blockset portion, we visit blocked websites specified in the blocked-domains.txt and make sure the output shows "forbidden content." Test this feature with this blocked HTTP site with our proxy server: e.g. gov.bg

5. **Evaluation**

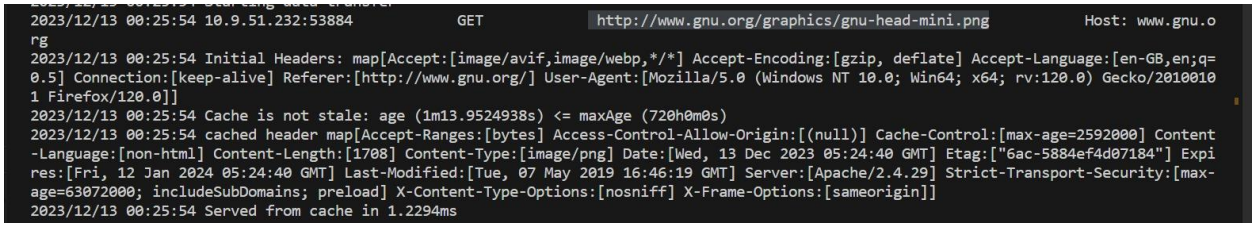
To evaluate the efficiency of implementing a cache mechanism versus not having one, we measure the time it takes to load an HTTP site in two scenarios. First, when a GET request is made, the proxy checks if the response is cached. If found, the cached response is served, and the time taken to complete the process is logged. The second scenario involves sending a fresh HTTP request to the destination server without utilizing the cache. By comparing the processing times in both situations, we can assess the impact of the cache mechanism on reducing response time and enhancing overall efficiency in serving HTTP content.

Results:

a. Fresh HTTP Request (No Cache): The time taken to serve a GET request directly from the destination server is **32.1523ms**.



b. Cached Response: For the same GET request, the time taken to serve the HTTP request from the cache is **1.2294ms**.



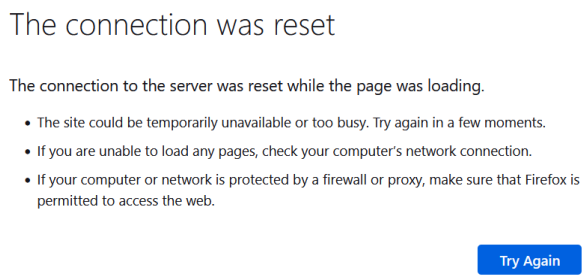
The results prove that our cache mechanism is effective in reducing the response significantly when serving from the cache. It highlights the cache's role in enhancing system efficiency by optimizing repeated content requests and improving overall performance.

6. Technical Challenges & Future Improvements

6.1 Our current cache eviction mechanism only evicts cache files based on the LRU rule without considering if the files are stale. In the future, we plan to enhance this policy by evicting files not only based on their LRU status but with a combination of whether they are stale, ensuring a more effective and efficient cache management.

6.2 Currently, we set the maximum limit of cache equal to the number of files in the directory as a proof-of-concept. However, in the future, we plan to base this limit on memory usage rather than file count. This change is important because it allows for more efficient use of resources, ensuring that the cache size aligns with available memory rather than an arbitrary file count. This modification also makes more sense since cache file sizes can vary significantly.

6.3 After implementing 'cache_lru', we realize that when we restart the proxy server without clearing the existing cache files, our system encounters an issue and fails to load the website from the cache, displaying an error message:



We've identified that this problem arises because restarting the proxy server reinitializes the HTTPCache instance, but doesn't delete the existing cache files in the 'http_cache' directory. This discrepancy creates a mismatch between the HTTPCache's current in-memory state and the outdated state of the cache files stored on the disk. Moving forward, we will focus on finding a solution that keeps the in-memory cache state aligned with the cache's persisted state on the disk to avoid such issues. As a current solution, we default our caching mechanism with 'cache_without_lru', but you may still test the functionality of 'cache_lru' by swapping the two files. 'cache_lru' will still work properly with a continuous proxy connection (without restarting the proxy).

7. **Run the Proxy Server:** Please follow the instructions here.

COS 316 Final Project - "HTTP/HTTPS Web Proxy and Cache"

8. Bibliography

1) Andile, Maximilien. (n.d.). Chapter 35: Build an HTTP Client. In Practical Go Lessons. Retrieved December 10, 2023, from https://www.practical-go-lessons.com/chap-35-build-an-http-client
2) Apache Traffic Server Authors. (n.d.). HTTP Proxy Caching. Apache Traffic Server Documentation. Retrieved December 11, 2023, from https://docs.trafficserver.apache.org/admin-guide/configuration/cache-basics.en.html
3) Bendersky, E. (2022, October 24). Go and Proxy Servers: Part 1 - HTTP Proxies. Eli Bendersky's website. Retrieved December 10, 2023, from https://eli.thegreenplace.net/2022/go-and-proxy-servers-part-1-http-proxies
4) Carson. (2021, March 3). HTTP Cache — How does it work step by step? Medium. Retrieved December 11, 2023, from https://cabulous.medium.com/http-cache-how-does-it-work-step-by-step-6ec8e2f303ec
5) Davidson, Kristin. (2022, April 21). How To Make an HTTP Server in Go. DigitalOcean. Retrieved December 10, 2023, from https://www.digitalocean.com/community/tutorials/how-to-make-an-http-server-in-go
6) Davidson, Kristin. (2022, April 26). How To Make HTTP Requests in Go. DigitalOcean. Retrieved December 10, 2023, from https://www.digitalocean.com/community/tutorials/how-to-make-http-requests-in-go
7) Davidson, Kristin. (2022, January 21). How To Run Multiple Functions Concurrently in Go. DigitalOcean. Retrieved December 10, 2023, from https://www.digitalocean.com/community/tutorials/how-to-run-multiple-functions-concurrently-in-go
8) Fielding, R., Nottingham, M., & Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Caching. Internet Engineering Task Force. Retrieved December 11, 2023, from https://datatracker.ietf.org/doc/html/rfc7234
9) Go Authors. (n.d.). Context. In context package. Go Packages. Retrieved December 10, 2023, from https://pkg.go.dev/context#Context
10) Go Authors. (n.d.). Hijacker. In http package. Go Packages. Retrieved December 9, 2023 website, from https://pkg.go.dev/net/http#Hijacker
11) Go Authors. (n.d.). http package. Go Packages. Retrieved December 8, 2023, from https://pkg.go.dev/net/http
12) Go Authors. (n.d.). NewRequest. In net/http package. Go Packages. Retrieved December 10, 2023, from https://pkg.go.dev/net/http#NewRequest
13) Go Authors. (n.d.). Time. In time package. Go Packages. Retrieved December 10, 2023, from https://pkg.go.dev/time#Time
14) HackTricks Authors. (n.d.). Abusing Hop-by-Hop Headers. HackTricks. Retrieved December 10, 2023, from https://book.hacktricks.xyz/pentesting-web/abusing-hop-by-hop-headers
15) Jacquemin, Léo. (2019, October 24). An in-depth introduction to HTTP caching: Cache-Control & Vary. freeCodeCamp. Retrieved December 11, 2023, from

https://www.freecodecamp.org/news/an-in-depth-introduction-to-http-caching-cache-control-vary/
16) Lowicki, M. (2017, October 28). HTTP(S) proxy in Golang in less than 100 lines of code. Medium. Retrieved December 11, 2023, from https://medium.com/@mlowicki/http-s-proxy-in-golang-in-less-than-100-lines-of-code-6a51c2f2c38c
17) Lowicki, M. (2018). HTTPS Proxies Support in Go 1.10. Medium. Retrieved December 9, 2023, from https://medium.com/@mlowicki/https-proxies-support-in-go-1-10-b956fb501d6b
18) Mozilla Corporation. (n.d.). HTTP Caching. MDN Web Docs. Retrieved December 11, 2023, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching
19) Mozilla Corporation. (n.d.). HTTP Messages. MDN Web Docs. Retrieved December 10, 2023, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages
20) Mozilla Corporation. (n.d.). X-Forwarded-For. MDN Web Docs. Retrieved December 9, 2023], from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-For
21) Posnick, Jeff. (n.d.). Prevent unnecessary network requests with the HTTP Cache. web.dev. Retrieved December 10, 2023, from https://web.dev/articles/http-cache#response-headers
22) REDBot Authors. (n.d.). REDBot. Retrieved December 11, 2023, from https://redbot.org/
23) Tronic, Josh. (2021, June 11). How to Download Files with cURL. DigitalOcean. Retrieved December 10, 2023, from https://www.digitalocean.com/community/tutorials/workflow-downloading-files-curl
24) Wikipedia contributors. (2023, October 10). Hop-by-hop transport. In Wikipedia. Retrieved December 10, 2023, from https://en.wikipedia.org/wiki/Hop-by-hop_transport

Acknowledgements of the Use of ChatGPT

We would like to acknowledge the use of the ChatGPT AI language model, developed by OpenAI, as a reference resource for providing explanations and insights on various topics related to the development of this project. We want to emphasize that ChatGPT was used solely for educational and reference purposes, and all original work and ideas presented in this project are our own. Here are some examples of how ChatGPT was used for this project:

1. We used ChatGPT like any other search engine to answer factual questions and provide explanations for certain concepts.
2. We used ChatGPT to debug our code by referencing its suggestions.
3. We used ChatGPT to provide suggestions on commenting to adhere to professional standards.