Heigh performance proxy

Kailin Zhang
Tufts University
Email: Kailin.Zhang@tufts.edu

Jieling Cai
Tufts University
Email: Jieling.cai@tufts.edu

Xiaoxiong Huang
Tufts University
Email: xiaoxiong.huang@tufts.edu

***Abstract*—Building a proxy from scratch and implementing some of popular advance functionality, is a enormous step in learning network constitution and socket program in c.**

Dec 17, 2021

1. Introduction

This proxy have all the baseline features of a simple HTTP proxy. It support basic GET and CONNECT method of HTTP request, can handle multiple clients sent different request at the same time. There is a HTTP cache to make sure our proxy is worst to use compare to the baseline.

We also implemented some of additional functionality on this proxy. With user's permission we are able to interception the request and content for a HTTPs request and serve the client with the content we get from encrypted server. User can pass in a list of host which they want to block and we can block those host for them. Also user can limiting the bandwidth by pass in the bandwidth they want to have.

We optimized the performance by relay the whole program on a single SELECT method and some effective data struct, we will talk more about that in advance features section.

2. Advance features

2.1. Trusted proxy

We have successfully implemented trusted proxy by using OpenSSL. We used a self signed certificate to intercept content send by client. We have access to the messages send by client by decrypt it. Then we make can sent content or make connection to client's target server, encrypt it and pass it to the server. This goes other way around, we also have access to the information passed by server to client. This is a huge step in building a proxy. With the ability to look at the content, we can have so many additional features build on top of this and you just can not have them with out using a trusted proxy.

2.2. Performance Optimization

As motioned before the whole proxy is built on a signal SELECT. This will give us the ability to handle multiple client at the same time and not slow by some client that have a really slow bottleneck. Also, this will make proxy robust enough to make sure any bad request will not shut our proxy down. On top of that, we have a one to relationship between client, client's socket, server and server's socket. We store them in a list data struct with access using pointer. Any query and modify are done in linear time. This is also a very important features in a proxy, the reason is simple enough. Who does not want to fast internet connection.

2.3. Bandwidth limiting

Within the client and server data struct, we add a bandwidth variable. Which you can pass in when you lunch the proxy. Every time we read, we decrease the bandwidth variable for that client or server. Once it reach to 0, we stop read. Bandwidth will restore every second. This features will come in handy when you want to download a file and witch a video, you dont care how fast you will get that file but you do want your video not to be freeze every second. You can limit the bandwidth of the download using this proxy. Some other usefully scenario can be you want to prank your room mate.

2.4. HTTP & HTTPs Caching

We reused the cache in previous assignment. We can set timeout for each cache unit, we can set cache size, and we can also update cache unit with the same key. For HTTP & HTTPs Caching, we use $[host\_url]$ as the key. Each time we access the same website with same URL and host, we can get content from cache. Specifically, for HTTPs, when we access some website, there could be several TCP connections. We save each http response in a separate cache unit. So, if there is another request to the same website but some URL changes, we can still retrieve from our cache for those unchanged URL.

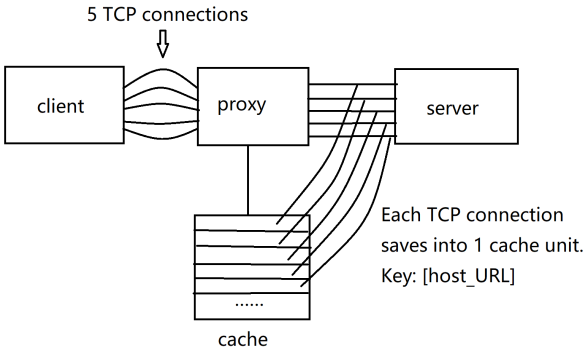


Figure 1. The cache saves each response in different cache unit

2.5. Cooperative Caching

We can set different proxy servers in configuration file (cache _ server.txt). At the beginning of our program, the proxy tries to connect other proxies through TCP, and keep the successful connections on while it's running. When there're some http requests, the proxy will send cache request (CACHEREQ) to other proxies through our own protocol. If some other proxy has the cache we need, it will send it back and our proxy will save it to cache and send it back to the client. If no other proxy has cached the content we need, our proxy will connect to server to fetch it.
The format of our own cache request is $CACHEREQ[URL]\backslash r\backslash nHost : [host : port]\backslash r\backslash n\backslash r\backslash n.$ It's like GET request, so that we can reuse the http request analysis function.

CACHEREQ [URL]
Host: [host:port]

For example:
CACHEREQ https://www.google.com/
Host: www.google.com:80

Figure 2. Cache Request – our own protocol

local > cache_server.txt

```
1  [localhost:8080]
2  [localhost:9150]
3  [localhost:0909]
4
```

Figure 3. cooperative proxies configuration file

2.6. Content filtering

We use host blocking as a form of content filtering. Proxy can specify a list of or single hostname(s) in the third input argument that it wants to block. If one wants to specify multiple hosts, different hostnames should be separated by a comma (e.g.: "www.tufts.edu,www.youtube.com"). The proxy can also block any hostname that contains specific keywords (e.g., input "tufts" as the third argument will block any host of which the hostname contains "tufts" keyword). For any host that is blocked, the proxy will refuse to forward the request to the destination server and send notification saying that the request has been blocked to client. In this case, the corresponding client will not receive any HTML pages, but proxy can still keep working with other clients' requests successfully. For any webpage that is blocked, client will see the following page on the browser:

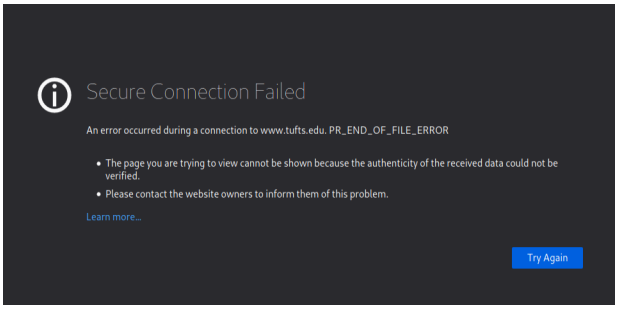


Figure 4. Content filtering

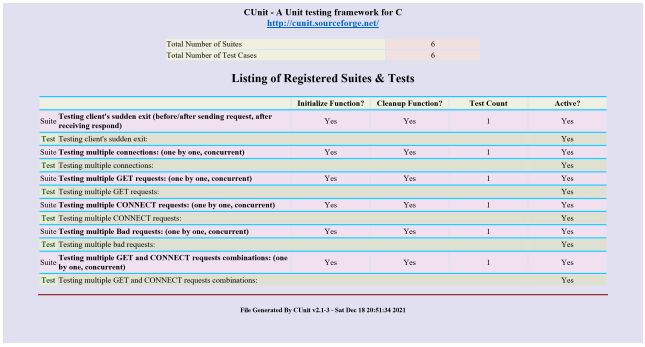
3. Evaluation

3.1. Error testing

To begin with, it will be good to introduce what we have done toward our error handlings. When any of the following bad cases happened, we closed the corresponding TCP connections and TLS connections on both sides, while still preserving other clients'/servers' connections and processing their requests (specifically, for the case that client sends bad requests, the proxy will send "Bad Request!" prompt back to client):
1. Any client or server suddenly and unexpectedly exit
2. Proxy failed to forward message or read responds from server
3. Proxy failed to make TCP connections/TLS connections
4. The destination server that client specified does not exist
5. Client sends malicious messages or bad requests to proxy

To test if our error handlings work and whether our proxy keep running under any scenario, we perform unit testing for testing whether our proxy will be brought down by any above misbehavior of the client or server. We depend on the CUnit (http://cunit.sourceforge.net/), a library for C

unit testing, to perform automatic unit testing and output results (with a format of ".xml" file). Associated codes are under a separate directory called "test" in our project. The basic design of unit testing is structured as follows:
1. $client.c$ includes all the possible behaviors of clients, which are making connections to the destination servers (options: multiple clients make connections one by one/at the same time), clients exiting (options: exit before or after sending the requests/receiving messages), and sending requests (options: multiple clients send requests one by one/at the same time). To send requests or make connections at the same time, we implement multi-threads to test the concurrency.
2. $test\_fun.c$ uses CUnit framework to build up test cases and test suites, inputs different parameters into the functions and uses CUnit Assertion to check whether the expected output is the same as the real output. We have included 18 tests in this C file already, which simulates the clients' different behaviors in the context of both HTTP (GET) and HTTPS (CONNECT) requests. Particularly, we also pass different kinds of malicious requests for testing (e.g.: some are incomplete or in a random format/some are without an existing host)
3. $run\_test.c$ is the main program to start unit testing
4. After finishing automatic testing, the output results will be included in the "results" directory. Specifically, the results are the two files named "Output-Listing.xml" and "Output-Results.xml". Our example test cases output the results as follows:



CUnit - A Unit testing framework for C
http://cunit.sourceforge.net/

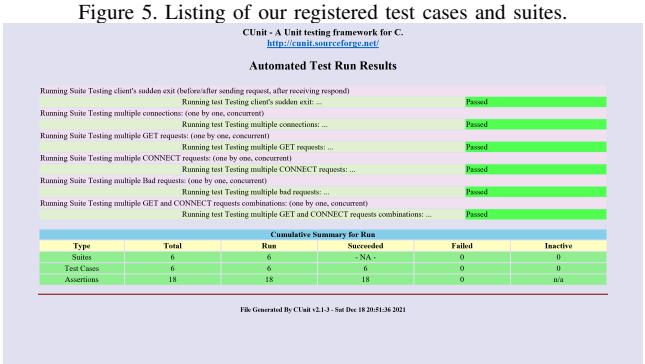
Total Number of Suites	6
Total Number of Test Cases	6

Listing of Registered Suites & Tests

		Initialize Function?	Cleanup Function?	Test Count	Active?
Suite	Testing client's sudden exit (before/after sending request, after receiving respond)	Yes	Yes	1	Yes
Test	Testing client's sudden exit:				Yes
Suite	Testing multiple connections: (one by one, concurrent)	Yes	Yes	1	Yes
Test	Testing multiple connections:				Yes
Suite	Testing multiple GET requests: (one by one, concurrent)	Yes	Yes	1	Yes
Test	Testing multiple GET requests:				Yes
Suite	Testing multiple CONNECT requests: (one by one, concurrent)	Yes	Yes	1	Yes
Test	Testing multiple CONNECT requests:				Yes
Suite	Testing multiple Bad requests: (one by one, concurrent)	Yes	Yes	1	Yes
Test	Testing multiple bad requests:				Yes
Suite	Testing multiple GET and CONNECT requests combinations: (one by one, concurrent)	Yes	Yes	1	Yes
Test	Testing multiple GET and CONNECT requests combinations:				Yes

File Generated By CUnit v2.1-3 - Sat Dec 18 20:51:34 2021

Figure 5. Listing of our registered test cases and suites.



CUnit - A Unit testing framework for C.
http://cunit.sourceforge.net/

Automated Test Run Results

Running Suite Testing client's sudden exit (before/after sending request, after receiving respond)
Running test Testing client's sudden exit: ... Passed
Running Suite Testing multiple connections: (one by one, concurrent)
Running test Testing multiple connections: ... Passed
Running Suite Testing multiple GET requests: (one by one, concurrent)
Running test Testing multiple GET requests: ... Passed
Running Suite Testing multiple CONNECT requests: (one by one, concurrent)
Running test Testing multiple CONNECT requests: ... Passed
Running Suite Testing multiple Bad requests: (one by one, concurrent)
Running test Testing multiple bad requests: ... Passed
Running Suite Testing multiple GET and CONNECT requests combinations: (one by one, concurrent)
Running test Testing multiple GET and CONNECT requests combinations: ... Passed

Cumulative Summary for Run

Type	Total	Run	Succeeded	Failed	Inactive
Suites	6	6	- NA -	0	0
Test Cases	6	6	6	0	0
Assertions	18	18	18	0	n/a

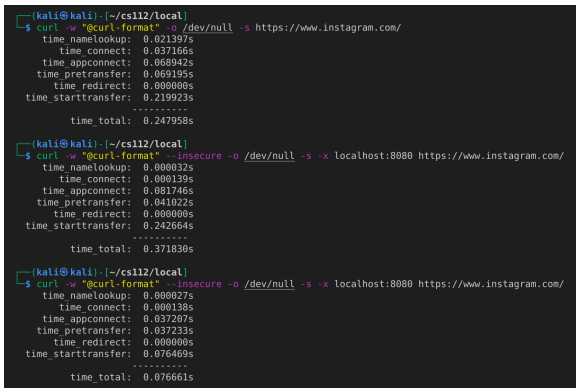
File Generated By CUnit v2.1-3 - Sat Dec 18 20:51:36 2021

Figure 6. we have successfully passed through all the 18 test cases.

3.2. Performance Test

We compared responding time of direct connections and proxy connections. We also tested how much time can cache save.

1) $curl\ -w\ "@curl-format"\ -o\ /dev/null\ -s\ [URL]$ – connect to server directly and show consuming time.
2) $curl\ -w\ "@curl-format"\ -o\ /dev/null\ -s\ -x\ [proxy\_host]\ [URL]$ – connect through proxy and show consuming time.



```
┌──(kali㉿kali)-[~/cs112/local]
└─$ curl -w "@curl-format" -o /dev/null -s https://www.instagram.com/
     time_namelookup:  0.021397s
        time_connect:  0.037166s
     time_appconnect:  0.068942s
    time_pretransfer:  0.069195s
       time_redirect:  0.000000s
  time_starttransfer:  0.219923s
                     ----------
          time_total:  0.247958s

┌──(kali㉿kali)-[~/cs112/local]
└─$ curl -w "@curl-format" --insecure -o /dev/null -s -x localhost:8080 https://www.instagram.com/
     time_namelookup:  0.000032s
        time_connect:  0.000139s
     time_appconnect:  0.081746s
    time_pretransfer:  0.041022s
       time_redirect:  0.000000s
  time_starttransfer:  0.242664s
                     ----------
          time_total:  0.371830s

┌──(kali㉿kali)-[~/cs112/local]
└─$ curl -w "@curl-format" --insecure -o /dev/null -s -x localhost:8080 https://www.instagram.com/
     time_namelookup:  0.000027s
        time_connect:  0.000138s
     time_appconnect:  0.037207s
    time_pretransfer:  0.037233s
       time_redirect:  0.000000s
  time_starttransfer:  0.076469s
                     ----------
          time_total:  0.076661s
```

Figure 7. curl commands and results

The first result tells us the time of direct connection. The second result shows the time we connect to the server through our proxy (the first time we request the URL or the cache is timeout). The third one is the second time we request to the same URL, so we get response from cache. As you can see, the responding time decreases repidly (from 0.37s to 0.07s).
We've tested 6 websites:

1) http://www.cs.tufts.edu/comp/112/index.html
2) http://www.cs.cmu.edu/~prs/bio.html
3) http://www.cs.cmu.edu/~dga/dga-headshot.jpg
4) https://www.google.com/search
5) https://www.instagram.com/
6) https://www.youtube.com/

	URL	Direct Connection	Proxy without Cache	Proxy with Cache
1	http://www.cs.tufts.edu/comp/112/index.html	0.020841	0.018359	0.010437
2	http://www.cs.cmu.edu/~prs/bio.html	0.061796	0.088141	0.01485
3	http://www.cs.cmu.edu/~dga/dga-headshot.jpg	0.69669	0.759197	0.01319
4	https://www.google.com/search\?q=landscape\&	1.188809	1.224466	0.098275
5	https://www.instagram.com/	0.2477958	0.37183	0.076661
6	https://www.youtube.com/	0.440885	0.53405	0.103056

Figure 8. Result from above URLs

As you can see in figure 9, there's a little overhead when accessing through the proxy. The responding time is a little slower than direct connection. But overall, the overhead is not so much and the proxy is efficient. When using cache, the responding time decreases drastically – some website has

less than 10% of original time (1.22s to 0.09s). Therefore, the cache works well.

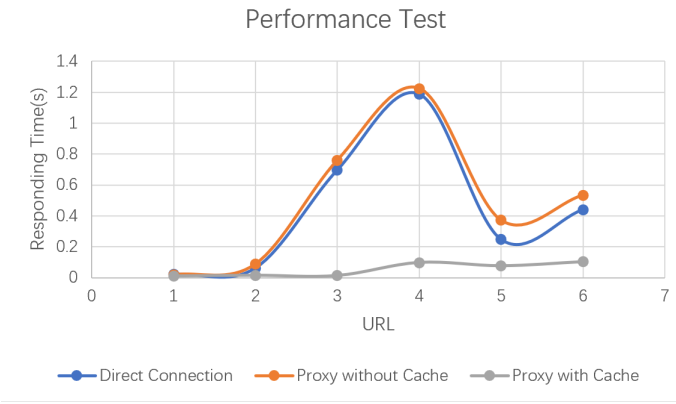


Figure 9. responding time chart

Finally, we did concurrency test with ApacheBench which send 5 request each to 3 clients at the same time.

```
┌──(kali㉿kali)-[~/cs112]
└─$ ab -n 5 -c 3 -X 127.0.0.1:9150 http://www.cs.tufts.edu/comp/112/index.html
This is ApacheBench, Version 2.3 <$Revision: 1879490 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking www.cs.tufts.edu [through 127.0.0.1:9150] (be patient).....done


Server Software:        Apache/2.4.6
Server Hostname:        www.cs.tufts.edu
Server Port:            80

Document Path:          /comp/112/index.html
Document Length:        6321 bytes

Concurrency Level:      3
Time taken for tests:   0.092 seconds
Complete requests:      5
Failed requests:        0
Total transferred:      33460 bytes
HTML transferred:       31605 bytes
Requests per second:    54.58 [#/sec] (mean)
Time per request:       54.965 [ms] (mean)
Time per request:       18.322 [ms] (mean, across all concurrent requests)
Transfer rate:          356.69 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       0
Processing:    13   38  23.5     47      64
Waiting:       11   35  23.4     44      62
Total:         13   38  23.5     47      64

Percentage of the requests served within a certain time (ms)
  50%     37
  66%     58
  75%     58
```

Figure 10.

4. Limitation

Only part of the cached content is used. For a connection to a specific website, the server establishes several TCP connections to the server. We cache all the http responses from different TCP connection separately. But during the testing, we found that only a few of cache units can be reused. This is because when user refreshes the website, most request URL changes, and we don't know if they match the previous cached content. For example, https://www.google.com/webhp?hl=en&ictx=2&sa=X&ved=0ahUKEwjZyKWM4u70AhWyct8KHSOGB1EQPQgI is a URL connecting to Google. Next time when user connects to Google in the same way, the latter part (after ".com/") might change. Fortunately, some picture URL stay unchanged, and they are normally the main part (size) of the page. So, if the picture cache works, we can save a lot of time when refreshing.

The proxy cannot find URL in cache when user restarts the browser and access to the previous accessed website. This is because when user restarts the Brower and access to the same website, the request URL changes. Because we use URL as the key of cache, we cannot reuse cache in this case.

Cooperative Caching: the proxy is hard to get cached content from other proxies. The reason is similar to the previous problem – different users use different URL, so the cache key doesn't work.

5. Contribution

5.1. Kailin Zhang

HTTP proxy framework, request/response analysis (information extraction), trusted proxy optimization, HTTP caching, HTTPs caching, cooperative caching, responding time test with curl, concurrency test with ApacheBenchmark.

5.2. Jieling Cai

Write some main functions to combine functionalities; Error handling; Fix bugs of GET (solved: failed to request multiple times across different HTTP pages on browser) CONNECT (solved: proxy error writing or reading or suddenly shut down) SSL (solved: failed to load complex HTML framework such as youtube, proxy shut down after a few requests); Unit testing (for error testing) using CUnit; Host filtering (blocking) as a firewall

5.3. Xiaoxiong Huang

Base line functionality, break large piece of code to smaller pieces. Data struct to store client and server. Generate certificate and key for SSL connection. Make proxy to support SSL interception and ability to connect to HTTPS server. Bandwidth limiting and performance optimization.

Acknowledgments

Many thanks to Dr. Fahad Dogar and TA Hafiz Mohsin for all the support and advise on this project.

References

[1] Openssl : https://www.openssl.org/

[2] Socket : https://www.chilkatsoft.com/refdoc/c_CkSocketRef.html