COMP112 Final Project Report

Group Members: Keren Zhou, Ruiyuan Gu

Modules and Contributions

Keren's Contributions

Cache module (cache.h/.c): LRU cache for full server responses.
Each cache item uses hostname + url as the key and uses the corresponding full server response (header + body) as the value. Cache items are stored in a linked list.
Newly inserted or updated items will be put/moved to the front of the linked list. Then, the least recently used item is at the back of the list. When the cache is full and there is no stale item in the cache, we simply pop the last item in the linked list, since the last item is the least recently used.
Codes are mostly reused from A0.

HTTP parsers (http_utils.h/.c): A collection of HTTP parsers and helper functions.
This module contains:

- a function to extract the head and the body from a request/response (parse_body_head()),
- functions to parse head of a request/response (parse_request_head(), parse_request_line(), parse_response_head(), parse_status_line()),
- functions to parse certain fields in header line (parse_cache_control(), parse_host_field())
- and functions to extract the first completed request/response from buffer (extract_first_request(), extract_first_response()).

Codes are mostly reused from A1.

Logger (logger.h/.c): Log facilities to print debug info.
Logger can print custom debug info with the filename and the line number where it is called.
Codes are mostly reused from A3.

Ruiyuan's Contributions

Socket buffer (sock_buf.h/.c): Message buffer for each opened socket.
Each opened socket has its own buffer. Each socket buffer stores partial messages from its socket. For a client socket, we can pop the first completed request to send it to the corresponding server. For a server socket, we can pop the first completed response to cache or send it to the corresponding client. After disconnecting a socket, we clear its buffer.
Each socket buffer also keeps track of key information of its socket, e.g. whether the socket is for a client or a server, when the last input came, whether the socket uses SSL or not, FD for

the socket on the other side of the connection, etc. These key informations help us handle each socket correctly.
We keep the last input timestamp in socket buffers to find whether the socket is idle for too much time. If the socket is idle for more than 10min, we manually disconnect it and clear its buffer, as the client/server that the socket represents may have disconnected unintentionally. We don't want an idle client/server to use up too much resource.

Peer Programming Part

Main driver (proxy.c): Main driver for the proxy.

Advance Features

Supporting Chunked Transfer Encoding

Chunked transfer encoding (https://en.wikipedia.org/wiki/Chunked_transfer_encoding) is a streaming data transfer mechanism available in HTTP/1.1. For a response using chunked encoding, there is no "Content-Length" in its header; instead, there is a header line "Transfer-Encoding: chunked". The response body in this case is transferred in several non-overlapping chunks. Each chunk starts with a hexadecimal byte size of the data to transfer followed by a newline ("\r\n"). Then, it follows the actual data and also followed by a newline("\r\n"). At the end of the response, the server will send a chunk with 0 byte of data ("0\r\n\r\n") as the marker of ending.

We support chunked transfer encoding since we found HTTP and HTTPS pages using chunked transfer encoding. If the proxy solely relies on the content length when parsing a server response, it probably won't regard any response using chunked transfer encoding as completed, so it won't cache it. This will cause a problem.

To support chunked transfer encoding, we first check whether the response header contains a header line of "Transfer-Encoding: chunked". If not, we simply use content length as the indicator of whether the response body is completed or not. If so, we check the response body in chunked mode. We first check if the body ends with "0\r\n\r\n", which indicates the end of chunked transfer encoding. If so, we then validate the size and format of each chunk. Only if all the chunks are valid and the body ends with "0\r\n\r\n", we regard this response as completed.

SSL Interception

In a normal SSL link between the client and the server, the communication is encrypted and the proxy cannot get the plaintext. This guarantees the security of data transfer. But, to cache server responses, the proxy needs to know the plaintext of the content. In this case, SSL interception is a good solution.

In SSL interception mode, once our proxy receives the CONNECT request, it will first create an SSL connection to the server, then reply "connection established" to the client and accept the SSL connect request from it. As we have two separate SSL connections, the proxy can read the plaintext when it forwards the messages between the client and the server. Then, the proxy can effectively cache server responses.

Performance Benefits

In this project, the performance is mainly about how quickly the client receives the response.

The main performance benefit in our proxy is from the cache. The proxy will save the most recent several responses from servers and record their URL. When the client requests the same URL in the cache, the proxy will pick the saved response from memory and send it to the client directly. In this case, the proxy doesn't need to forward the request to the server and wait for the response. As a result, the client can get the response much faster.

But for the proxy without SSL interception, caching the response in SSL connection is impossible. The proxy cannot encrypt or decrypt the message between clients and servers, which means it can only access the encrypted responses. Even if the proxy saved encrypted responses, they will not be accepted by other clients.

We implemented SSL interception, so caching HTTPS responses is possible in our proxy. As HTTPS are used by most websites on the Internet, such an advanced feature can improve the performance of our proxy largely.

Supporting chunked transfer encoding enables our proxy to handle the response in this format and cache them. Similarly, more websites can be cached, higher performance our proxy will have.

Evaluation Section

To evaluate the performance of our proxy, we compared the page load time of direct access, access via uncached proxy and access via cached proxy.

Method

We run our proxy on a local machine. Our local machine runs Windows 10. The proxy source code is compiled and run in a WSL2 environment of Ubuntu 20.04.

We use curl to access websites directly or via our proxy.
To access websites directly, we use the command `curl “<url>” --get`.

To access websites via our proxy, we use the command
`curl “<url>” --get --proxy “localhost:<port>”` to the command.
To access HTTPS websites via our proxy in SSL interception mode, we use command
`curl “<url>” --get --proxy “localhost:<port>” --insecure` to let curl trust our proxy.
To get page size, we use command
`curl “<url>” --get -so "/dev/null" -w "%{size_download}\n"`.
When we access a website via proxy for the first time, we believe the page is not cached and the load time is the page load time of an uncached proxy.
When we consecutively access a website a second time, we believe the page is cached and the load time is the page load time of a cached proxy.

We also write python scripts “bench_proxy_default.py” and “bench_proxy_ssl_interception.py” to automate our benchmark, which benchmark proxy in SSL tunnel (default) mode and SSL interception mode respectively. For each website, we firstly access it using curl via proxy twice consecutively, then directly access it using curl, then measure the page size using curl. Elapsed time for each access is timed using Python time library.

We benchmark our proxy with 50 the most visited websites. This list of websites refers to wiki: https://en.wikipedia.org/wiki/List_of_most_visited_websites. The ranking data updated according to November 1, 2021.
Since these 50 websites all use HTTPS, we also add HTTP websites to evaluate proxy performance on HTTP. However, HTTP websites are not common in popular websites, so we only found 9 HTTP websites to test with.

Result

HTTP Page Load Time

In this section, we measure and compare page load time for 9 HTTP websites.
When we access a HTTP website for the first time, access via proxy has no benefit compared to direct access.
When we access the website for the second time, the page load time decreases significantly. Since the page has been cached in proxy after the first proxy access, the page is loaded from cache on the localhost during the second access. Then, it saves both transmission delay and propagation delay.
If the proxy is deployed on some remote machine, the benefits of loading from cache depends on the bandwidth and the distance between the client and proxy. It may not show such a significant decrease in page load time.

Table 1. HTTP Page Load Time

URL	page size (Byte)	uncached proxy (s)	cached proxy (s)	direct access (s)

http://www.cs.cmu.edu/~prs/bio.html	8458	0.115	0.008	0.115
http://www.cs.cmu.edu/~dga/dga-headshot.jpg	2925958	4.582	0.016	6.337
http://info.cern.ch	646	0.366	0.008	0.315
http://brightbeautifulshiningsunrise.neverssl.com/online	2238	0.215	0.008	0.115
http://www.testingmcafeesites.com/	28634	0.466	0.008	0.516
http://www.softwareqatest.com	10445	0.265	0.008	0.315
http://www.http2demo.io/	105216	0.918	0.008	0.265
http://scp-wiki-cn.wikidot.com/	68721	0.165	0.008	0.115
http://scp-jp.wikidot.com/	85275	0.416	0.008	0.115

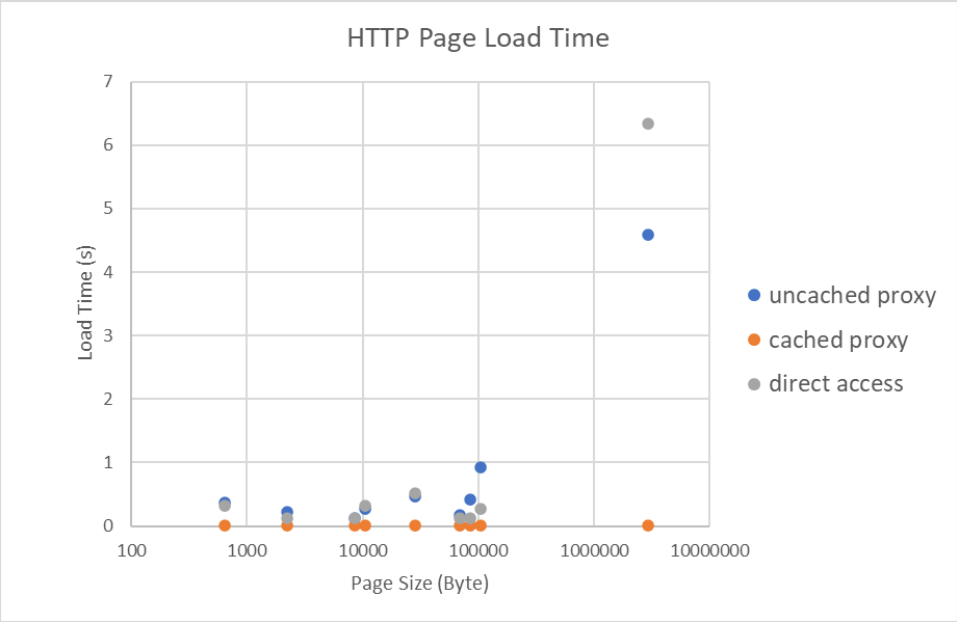

Figure 1. HTTP Page Load Time

HTTPS Page Load Time for SSL Tunnel Mode

Using SSL tunnel mode, our proxy simply forwards the encrypted data between the corresponding client and server. Since the proxy cannot get the plaintext of requests and responses, it has no idea what to cache in this case.

Without caching, there is no significant difference in page load time between the first and the second proxy access, since the proxy has to fetch the page from the server in all cases.

Theoretically, HTTPS page load time should be a little slower in SSL tunnel mode compared to direct access, since proxy access takes extra steps such as forwarding data between client and

server. However, in practice, page load time is not stable due to unpredictable path and traffic on the path between client and server. So, in some cases, proxy has shorter page load time.

Table 2. HTTPS Page Load Time for SSL Tunnel Mode

URL	page size (Bytes)	first proxy access (s)	second proxy access (s)	direct access (s)
https://www.google.com/	15307	0.165	0.316	0.165
https://www.youtube.com/	590457	0.366	0.366	0.416
https://www.facebook.com/	82927	1.32	0.767	0.516
https://twitter.com/	88823	0.868	0.667	0.617
https://www.instagram.com/	69860	0.566	0.617	2.022
https://www.baidu.com/	2443	0.867	0.868	0.918
https://www.wikipedia.org/	79455	0.667	0.516	0.466
https://yandex.ru/	109928	1.119	1.118	0.868
https://www.yahoo.com/	620503	1.62	0.968	0.717
https://www.xvideos.com/	112267	0.918	0.767	0.767
https://www.whatsapp.com/	164791	0.316	0.416	0.316
https://www.amazon.com/	1203	0.165	0.215	0.115
https://www.xnxx.com/	96455	0.867	0.717	0.667
https://www.netflix.com/	431580	0.466	0.466	0.416
https://outlook.live.com/owa/	37064	0.365	0.215	0.265
https://www.yahoo.co.jp/	40046	1.018	1.018	1.118
https://www.pornhub.com/	598809	0.466	0.567	0.416
https://zoom.us/	102518	0.315	0.215	0.215
https://www.office.com/	119460	0.265	0.215	0.315
https://www.reddit.com/	753112	1.32	2.674	1.922
https://vk.com/	0	0.616	0.667	0.466
https://www.tiktok.com/	0	0.265	0.115	0.064
https://xhamster.com/	193287	0.616	0.516	0.466
https://www.linkedin.com/	109236	0.366	0.265	0.366
https://discord.com/	41533	0.466	0.566	0.165
https://www.naver.com/	230167	0.466	1.62	1.47
https://www.twitch.tv/	113514	0.265	0.115	0.115
https://www.bing.com/	88529	0.265	0.215	0.165

https://www.microsoft.com/en-us/	217082	0.366	0.265	0.366
https://mail.ru/	310432	4.08	5.837	5.435
https://www.roblox.com/	60940	0.315	0.165	0.165
https://duckduckgo.com/	5722	0.165	0.165	0.115
https://www.pinterest.com/	85414	0.315	0.316	0.316
https://www.samsung.com/us/	438629	0.366	0.366	0.366
https://www.qq.com/	132388	0.165	0.115	0.115
https://www.msn.com/	51033	0.265	0.266	0.667
https://news.yahoo.co.jp/	426643	2.974	3.125	2.723
https://www.bilibili.com/	98921	0.918	1.269	1.269
https://www.ebay.com/	412085	0.567	0.566	0.567
https://www.google.com.br/	16050	0.215	0.165	0.566
https://www.globo.com/	649526	2.775	3.928	3.527
https://www.fandom.com/	110557	0.165	0.165	0.115
https://ok.ru/	201778	1.369	1.369	1.169
https://docomo.ne.jp/	9	0.817	0.767	0.867
realsrv.com	N.A.	N.A.	N.A.	N.A.
https://www.bbc.com/	247886	0.265	0.165	0.165
https://www.accuweather.com/	269	0.115	0.064	0.065
https://www.amazon.co.jp/	76079	0.667	0.517	0.617
https://www.walmart.com/	21	0.265	0.215	0.215

We fail to access realsrv.com using curl or browser. It doesn't respond after 20s timeout. vk.com and tiktok.com are 2 special cases. Their page size or content length is 0 when accessing with curl.

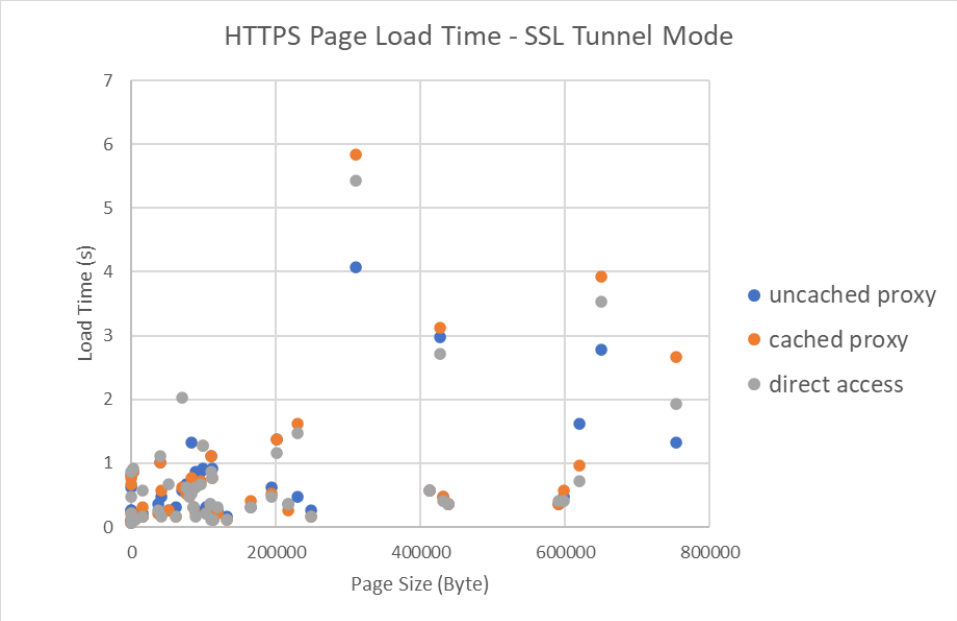

Figure 2. HTTPS Page Load Time for SSL Tunnel Mode

HTTPS Page Load Time for SSL Interception Mode

Using SSL Interception mode, the proxy establishes 2 seperate SSL connections, proxy-server and proxy-client. So, the proxy can access the plaintext of requests and responses. Then, the proxy can effectively cache those pages and decrease the page load time by loading pages from cache when it receives repeated requests.

Due to the limitation of our implementation, there are 15 HTTPS websites that our proxy cannot access in SSL interception mode.

According to table 3 and figure 3, page load time of some pages is decreased in orders of magnitude when our proxy accesses them for a second time. For example, facebook.com, instagram.com, wikipedia.org, yahoo.com, whatsapp.com, etc. By loading page from cache, it saves the time for SSL handshake between the proxy and the server, also the propagation delay and the transmission delay.

Table3. HTTPS Page Load Time for SSL Interception Mode

URL	page size (Bytes)	uncached proxy (s)	cached proxy (s)	direct access (s)
https://www.google.com/	15321	0.165	0.215	0.115
https://www.youtube.com/	594691	0.416	0.366	0.415
https://www.facebook.com/	82922	0.215	0.032	0.215
https://twitter.com/	N.A.	N.A.	N.A.	N.A.
https://www.instagram.com/	69859	0.265	0.032	1.218

https://www.baidu.com/	2443	0.516	0.266	0.516
https://www.wikipedia.org/	79455	0.215	0.065	0.165
https://yandex.ru/	0	0.516	0.466	0.366
https://www.yahoo.com/	623540	0.717	0.064	0.666
https://www.xvideos.com/	111621	0.867	0.265	0.817
https://www.whatsapp.com/	164939	0.366	0.032	1.169
https://www.amazon.com/	2671	0.115	0.115	0.065
https://www.xnxx.com/	96455	0.817	0.316	0.617
https://www.netflix.com/	N.A.	N.A.	N.A.	N.A.
https://outlook.live.com/owa/	37064	0.265	0.115	0.265
https://www.yahoo.co.jp/	40047	1.168	0.516	1.018
https://www.pornhub.com/	N.A.	N.A.	N.A.	N.A.
https://zoom.us/	102518	0.265	0.215	0.165
https://www.office.com/	119459	0.265	0.064	0.115
https://www.reddit.com/	N.A.	N.A.	N.A.	N.A.
https://vk.com/	0	0.617	0.516	0.467
https://www.tiktok.com/	0	0.265	0.115	0.115
https://xhamster.com/	189747	0.566	0.065	0.567
https://www.linkedin.com/	109236	0.316	0.033	0.265
https://discord.com/	41533	0.215	0.065	0.165
https://www.naver.com/	N.A.	N.A.	N.A.	N.A.
https://www.twitch.tv/	N.A.	N.A.	N.A.	N.A.
https://www.bing.com/	89021	0.215	0.065	0.165
https://www.microsoft.com/en-us/	N.A.	N.A.	N.A.	N.A.
https://mail.ru/	310484	5.385	0.617	2.373
https://www.roblox.com/	N.A.	N.A.	N.A.	N.A.
https://duckduckgo.com/	5722	0.165	0.065	0.115
https://www.pinterest.com/	N.A.	N.A.	N.A.	N.A.
https://www.samsung.com/us/	N.A.	N.A.	N.A.	N.A.
https://www.qq.com/	N.A.	N.A.	N.A.	N.A.
https://www.msn.com/	50839	0.215	0.064	0.215
https://news.yahoo.co.jp/	426859	2.524	0.366	2.323

https://www.bilibili.com/	96291	1.32	0.165	1.27
https://www.ebay.com/	N.A.	N.A.	N.A.	N.A.
https://www.google.com.br/	16065	0.215	0.215	0.165
https://www.globo.com/	649217	3.929	0.315	3.025
https://www.fandom.com/	N.A.	N.A.	N.A.	N.A.
https://ok.ru/	152425	1.169	0.466	1.319
https://docomo.ne.jp/	9	0.667	0.667	0.818
realsrv.com	N.A.	N.A.	N.A.	N.A.
https://www.bbc.com/	247952	0.165	0.033	0.165
https://www.accuweather.com/	N.A.	N.A.	N.A.	N.A.
https://www.amazon.co.jp/	76860	0.617	0.065	0.516
https://www.walmart.com/	21	0.616	0.365	0.516

Due to the limitation of our SSL interception implementation, multiple websites can't be accessed via proxy. Their page load time and page size are all set to N.A.
The page size for the same website may be different between table 2 and 3, since the contents of some websites are dynamic and variable. We may get content each time we access it.
vk.com and tiktok.com are 2 special cases. Their page size or content length is 0 when accessing with curl.

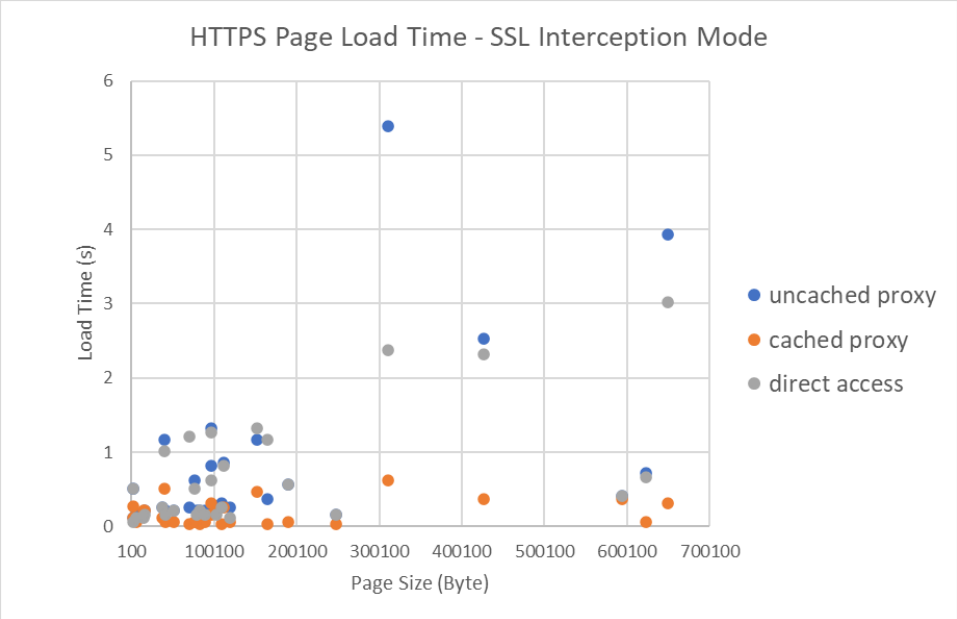

Figure 3. HTTPS Page Load Time for SSL Interception Mode

Limitations

The main limitation of our proxy is that it cannot let the browser trust it in SSL interception mode. We failed to make the browser completely trust the certificate of our proxy and start SSL connect with proxy without warning.

We attempted to make Firefox trust our browser. In the certificate settings of Firefox, we added our proxy certificate to authorities and set it to identify websites and mail users. In about:config of Firefox, we set security.enterprise_roots.enabled to true as Piazza post (https://piazza.com/class/kt1srs7oyk45ck?cid=280) pointed out. We also set network.stricttransportsecurity.preloadlist in about:config to false. We cleared the cache and restarted Firefox. However, the proxy still could work properly with Firefox.

When accessing websites like Google and YouTube via proxy, Firefox shows “Did Not Connect: Potential Security Issue” and shows error code MOZILLA_PKIX_ERROR_CA_CERT_USED_AS_END_ENTITY under the advanced option. And we cannot access it by force under the advanced option.

When accessing websites like tufts.edu and cplusplus.com via proxy, Firefox shows that “Warning: Potential Security Risk Ahead” and shows error code MOZILLA_PKIX_ERROR_CA_CERT_USED_AS_END_ENTITY. But we can access it by force by clicking the “Accept the Risk and Continue” button under the advanced option.

Although our proxy cannot work properly with browsers in SSL interception mode. It can handle HTTPS requests from curl if we add “--insecure” option in the curl command. For example, we run our proxy on localhost:9999, then run command `curl --proxy "localhost:9999" --insecure --verbose "https://www.google.com"`. We can get “HTTP/1.1 200 OK” from the server. Also, our proxy works smoothly with browsers in SSL tunnel mode.