Computer Networking Lab 2

Srinjoy Som	**220123074**
Shubham Kumar Jha	**220123081**
Yash Singhal	**220123072**

For question 2 and question 3, there are several constraints for running the code.

Firstly code includes Ws2_32 header file, so it should be run on Windows Operating System only.

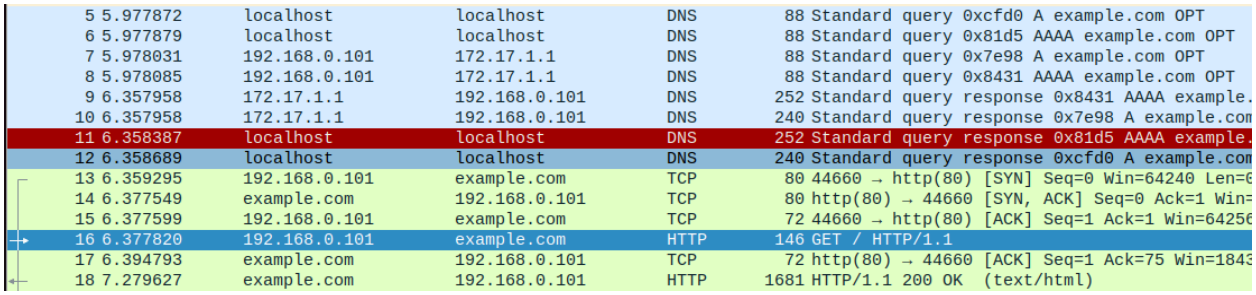
Also in order to run the code see the **readme.md** file as it includes multiple steps.

Question 1.

To have some HTTP traffic, we search for *www.example.com*, and capture the traffic using tcpdump and store it in a file.

Now, we analyze the captured packets, we see that there is initial DNS lookup and TCP handshake.

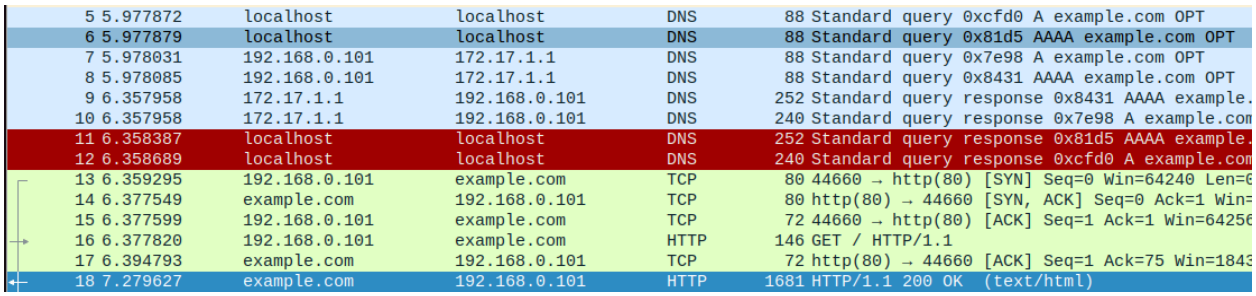
After that HTTP GET request is sent to example.com (*line highlighted in dark blue*)



5	5.977872	localhost	localhost	DNS	88	Standard query 0xcfd0 A example.com OPT
6	5.977879	localhost	localhost	DNS	88	Standard query 0x81d5 AAAA example.com OPT
7	5.978031	192.168.0.101	172.17.1.1	DNS	88	Standard query 0x7e98 A example.com OPT
8	5.978085	192.168.0.101	172.17.1.1	DNS	88	Standard query 0x8431 AAAA example.com OPT
9	6.357958	172.17.1.1	192.168.0.101	DNS	252	Standard query response 0x8431 AAAA example.
10	6.357958	172.17.1.1	192.168.0.101	DNS	240	Standard query response 0x7e98 A example.com
11	6.358387	localhost	localhost	DNS	252	Standard query response 0x81d5 AAAA example.
12	6.358689	localhost	localhost	DNS	240	Standard query response 0xcfd0 A example.com
13	6.359295	192.168.0.101	example.com	TCP	80	44660 → http(80) [SYN] Seq=0 Win=64240 Len=0
14	6.377549	example.com	192.168.0.101	TCP	80	http(80) → 44660 [SYN, ACK] Seq=0 Ack=1 Win=
15	6.377599	192.168.0.101	example.com	TCP	72	44660 → http(80) [ACK] Seq=1 Ack=1 Win=64256
16	6.377820	192.168.0.101	example.com	HTTP	146	GET / HTTP/1.1
17	6.394793	example.com	192.168.0.101	TCP	72	http(80) → 44660 [ACK] Seq=1 Ack=75 Win=1843
18	7.279627	example.com	192.168.0.101	HTTP	1681	HTTP/1.1 200 OK (text/html)

The url requested is example.com and the HTTP version used is HTTP/1.1

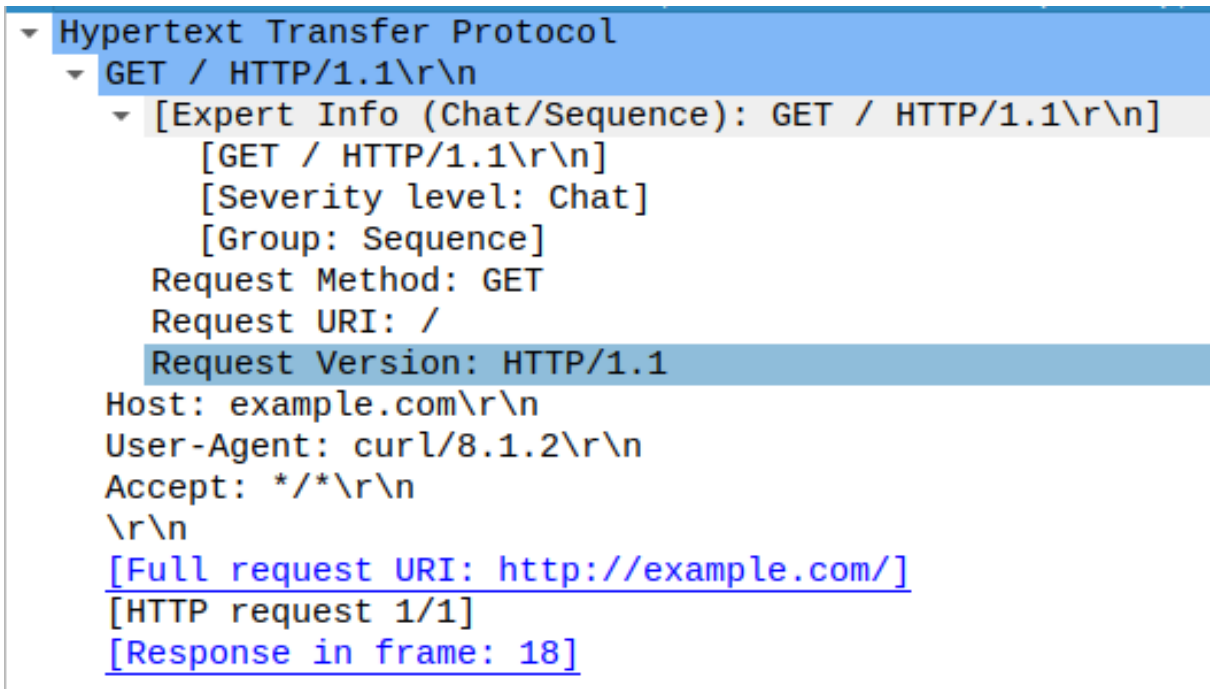
Server on seeing the request responds to the request accordingly(*line highlighted in dark blue*)



5	5.977872	localhost	localhost	DNS	88	Standard query 0xcfd0 A example.com OPT
6	5.977879	localhost	localhost	DNS	88	Standard query 0x81d5 AAAA example.com OPT
7	5.978031	192.168.0.101	172.17.1.1	DNS	88	Standard query 0x7e98 A example.com OPT
8	5.978085	192.168.0.101	172.17.1.1	DNS	88	Standard query 0x8431 AAAA example.com OPT
9	6.357958	172.17.1.1	192.168.0.101	DNS	252	Standard query response 0x8431 AAAA example.
10	6.357958	172.17.1.1	192.168.0.101	DNS	240	Standard query response 0x7e98 A example.com
11	6.358387	localhost	localhost	DNS	252	Standard query response 0x81d5 AAAA example.
12	6.358689	localhost	localhost	DNS	240	Standard query response 0xcfd0 A example.com
13	6.359295	192.168.0.101	example.com	TCP	80	44660 → http(80) [SYN] Seq=0 Win=64240 Len=0
14	6.377549	example.com	192.168.0.101	TCP	80	http(80) → 44660 [SYN, ACK] Seq=0 Ack=1 Win=
15	6.377599	192.168.0.101	example.com	TCP	72	44660 → http(80) [ACK] Seq=1 Ack=1 Win=64256
16	6.377820	192.168.0.101	example.com	HTTP	146	GET / HTTP/1.1
17	6.394793	example.com	192.168.0.101	TCP	72	http(80) → 44660 [ACK] Seq=1 Ack=75 Win=1843
18	7.279627	example.com	192.168.0.101	HTTP	1681	HTTP/1.1 200 OK (text/html)

Status code of the response is **200.** The Status phrase is **OK.** This means everything went fine, and the server was able to fulfil the client request.

Request Header:



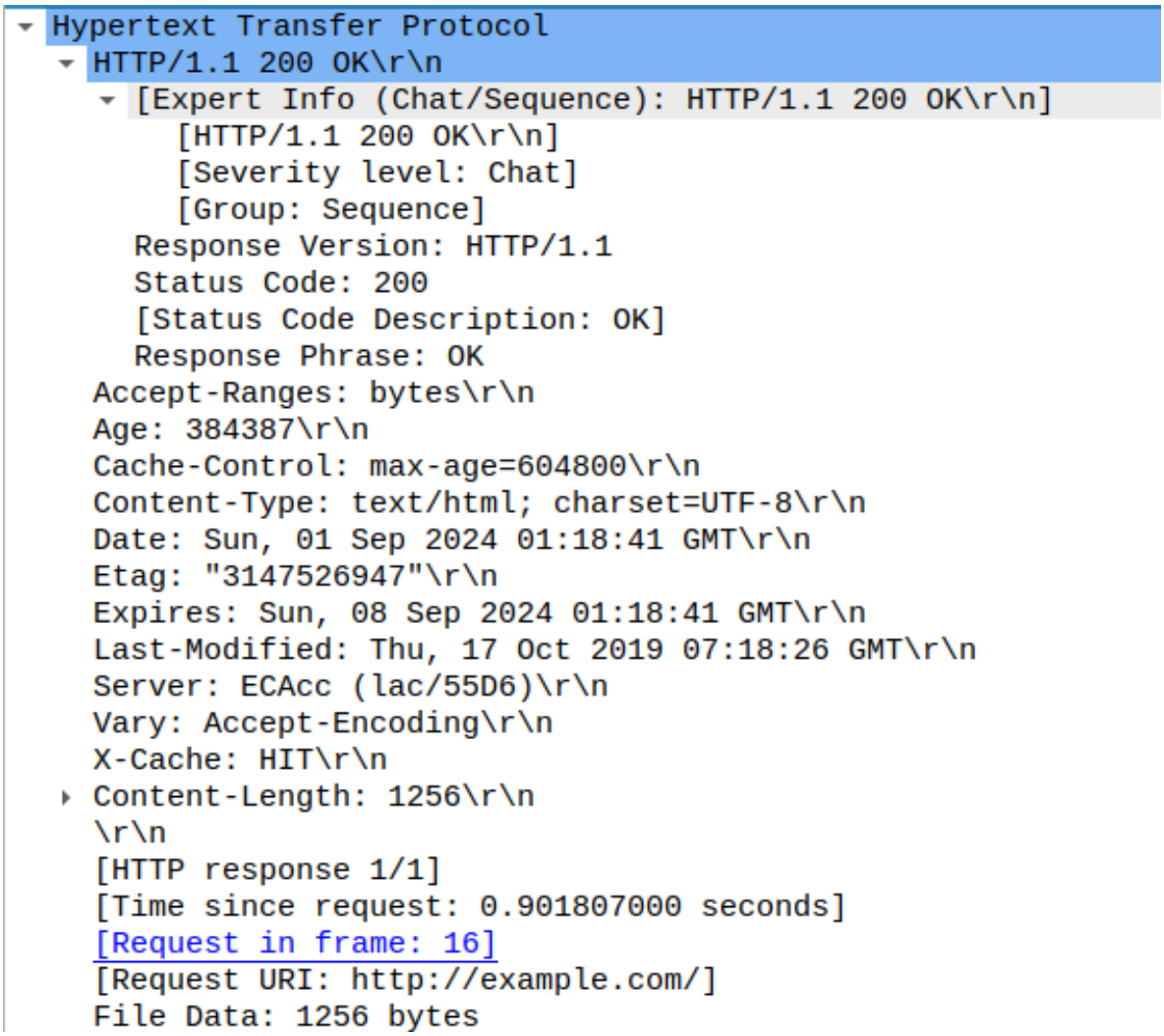
```
Hypertext Transfer Protocol
  GET / HTTP/1.1\r\n
    [Expert Info (Chat/Sequence): GET / HTTP/1.1\r\n]
      [GET / HTTP/1.1\r\n]
      [Severity level: Chat]
      [Group: Sequence]
    Request Method: GET
    Request URI: /
    Request Version: HTTP/1.1
  Host: example.com\r\n
  User-Agent: curl/8.1.2\r\n
  Accept: */*\r\n
  \r\n
  [Full request URI: http://example.com/]
  [HTTP request 1/1]
  [Response in frame: 18]
```

HTTP Request headers include

1. The method used (*here, GET*),
2. url requested (*here, /*)
3. HTTP version used (*here, HTTP/1.1*)
4. Host (*here, www.example.com*)
5. User-Agent (*here, curl/8.1.2*)
6. Accept (*here, */**).

Accept specifies the media types that the client is able to handle.
/ means that the client can accept any media type.

Response Header:



HTTP Response Headers include

1. HTTP version used (*here, HTTP/1.1*)
2. Response code and phrase (*here, 200 OK*)
3. Accept-Ranges (*here, bytes*)

4. Age (*here, 384387*)
5. Cache-Control (*here, max-age=604800*)
6. Content-Type (*here, text/html; charset=UTF-8*)
7. Date (*here, Sun, 01 Sep 2024 01:18:41 GMT*)
8. Etag (*here, Etag: "3147526947"*)
9. Expires (*here, Expires: Sun, 08 Sep 2024 01:18:41 GMT*)
10. Last-Modified (*here, Thu, 17 Oct 2019 07:18:26 GMT*)
11. Server (*here, ECAcc (lac/55D6)*)
12. Vary (*here, Accept-Encoding*)
13. X-Cache (*here, HIT*)
14. Content-Length (*here, 1256*)

Accept-Ranges indicates whether the server supports partial requests and, if so, which type of range requests it accepts.
bytes indicates that the server supports byte-range requests, which means clients can request specific portions (byte ranges) of the resource. This is useful for resuming interrupted downloads or for streaming scenarios.

Age shows the age of the cached response in seconds.
In this case, the response has been in a cache for 384387 seconds (about 4.5 days).

Cache-Control can contain one or more directives, separated by commas, each defining a specific caching behaviour.
max-age=604800 directive informs caches that the response is considered fresh for 604800 seconds (7 days). After this period, the cache should revalidate the content with the origin server.

Content-Type specifies the media type of the response body.
text/html; charset=UTF-8 indicates that the response is of type text/html (HTML document) and is encoded using UTF-8 character encoding.

Date specifies the date and time at which the response was generated by the server. It is in GMT format.

ETag(Entity Tag) is a unique identifier assigned to a specific version of a resource. If the resource changes, the ETag value changes. This helps with caching and conditional requests, allowing clients to check if the resource has been modified.

Expires specifies the date and time after which the response is considered stale. After this date, the client must validate the response with the server.

Last-Modified indicates the last time the resource was modified. It helps in caching and conditional requests, where a client can use the If-Modified-Since request header to check if the resource has been updated.

Server provides information about the server software handling the request.

Vary indicates that the server's response can change depending on the value of one or more specified request headers.
Accept-Encoding tells caches that responses with different Accept-Encoding values should be treated separately.

X-Cache: HIT indicates that the response was served from a cache rather than being fetched from the origin server.

Content-Length specifies the size of the response body in bytes.

For **Response Time**,
Firstly, we can see that Wireshark automatically gives the field Time since request.
Secondly, we can also see the time at which the response was received and subtract that from the time when the request was sent.
Using the first approach, the Response Time is: 0.901807

Using the second approach, the Response Time is:

7.279627 - 6.377820 = 0.901807

To examine the HTTP content we expand the **Line-based text data: text/html (46 lines)** field given in wireshark.

```
<!doctype html>\n
<html>\n
<head>\n
    <title>Example Domain</title>\n
\n
    <meta charset="utf-8" />\n
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />\n
    <meta name="viewport" content="width=device-width, initial-scale=1" />\n
    <style type="text/css">\n
    body {\n
        background-color: #f0f0f2;\n
        margin: 0;\n
        padding: 0;\n
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;\n
        \n
    }\n
    div {\n
        width: 600px;\n
        margin: 5em auto;\n
        padding: 2em;\n
        background-color: #fdfdff;\n
        border-radius: 0.5em;\n
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);\n
    }\n
    a:link, a:visited {\n
        color: #38488f;\n
        text-decoration: none;\n
    }\n
    @media (max-width: 700px) {\n
        div {\n
            margin: 0 auto;\n
            width: auto;\n
        }\n
    }\n
    </style>    \n
</head>\n
\n

 \n
 <body>\n
 <div>\n
     <h1>Example Domain</h1>\n
     <p>This domain is for use in illustrative examples in documents. You may use this\n
     domain in literature without prior coordination or asking for permission.</p>\n
     <p><a href="https://www.iana.org/domains/example">More information...</a></p>\n
 </div>\n
 </body>\n
 </html>\n
```

Question 2.

To simulate a DNS search we used map data structure one for root dns .Then two for TLDs i.e com , org .Few authoritative for google , example , new_domain , example_org(two fake websites just for simulation). In these authoritative DNS servers the ip address for host servers are stored. Apart from these I also made a local DNS which is basically a cache (following LRU algorithm). Now when I search for an ip address of a hostname it first checks if it is in the localDNS or not if it is there then it returns the ip in cache otherwise it will first go the root DNS map from there to corresponding TLDs and eventually return the IP address if present. I also added a random time delay and error probability which simulates delay and not acknowledgement of packet. Now if such things happens i.e sender resends the page.but if the error repeats for more than 3 times then it returns error.

Now for the webcache part also i have implemented LRU cache (size 5 as told in question)and when i send a get request the page returned as well as stored in the cache and then if a cache hit happens the response comes from the cache itself.

To do this we have to use unordered map, list, and sockets (winsock and winsock2 for Windows and sys/socket.h for Linux).

The header files are shown below.

```cpp
#include <bits/stdc++.h>
#include <string>
#include <unordered_map>
#include <list>
#ifndef SOCKET_INCLUDES_H
#define SOCKET_INCLUDES_H

#ifdef _WIN32
    // Windows-specific headers
    #include <winsock2.h>
    #include <ws2tcpip.h>
    #include <windows.h>

    // Link with Ws2_32.lib
    #pragma comment(lib, "Ws2_32.lib")

#elif defined(__linux__)
    // Linux-specific headers
    #include <sys/types.h>
    #include <sys/socket.h>
    #include <netinet/in.h>
    #include <arpa/inet.h>
    #include <unistd.h>
    #include <netdb.h>
#else
    #error "Platform not supported"
#endif

#include <iostream>
#include <cstring>   // For memset and memcpy

#endif // SOCKET_INCLUDES_H

#pragma comment(lib, "Ws2_32.lib")
```

Also if you are running from ide like codeBlocks or VScode go to project and build options and there add Ws2_32 in the list of files to be added to the linker if not already done.

To implement the probability and failing case i have made a waitandReturn function

```cpp
int waitAndReturn() {
    int tot_wait;
    for (int attempt = 0; attempt < 3; ++attempt) {
        int prob = rand() % 3;
        int waitTime = (prob == 0) ? 1 : (prob == 1) ? 3 : 4;
        int result= (rand() % 100 < 85) ? 1 : -1;
        if (waitTime > 2 || result == -1) {
           this_thread::sleep_for(chrono::seconds(2)); // if wait is greater than 2 secs system waits for 2 secs and then breaks so does it again
           tot_wait+=2;
           continue;
        }
        this_thread::sleep_for(chrono::seconds(waitTime));
        tot_wait+=waitTime;
        if (result == 1) {
            cout << "Waited for " << waitTime << " seconds, result: " << result << endl;
            return result;
        } else {
            cout << "Received -1, retrying..." << endl;
        }
    }
    cout << "failed" << endl;
    return -1;
}
```

Code also contains implementation of both local DNS and web cache with LRU cache using the following code

```cpp
// ____________________________________________________
class LRUCachedns {
public:
    LRUCachedns(size_t capacity) : capacity_(capacity) {}

    string get(const string& domain) {
        auto it = cache_.find(domain);
        if (it == cache_.end()) {
            return "";
        } else {
            access_list_.splice(access_list_.begin(), access_list_, it->second.second);
            return it->second.first;
        }
    }

    void put(const string& domain,  string& ip_address) {
        auto it = cache_.find(domain);
        if (it != cache_.end()) {
            it->second.first = ip_address;
            access_list_.splice(access_list_.begin(), access_list_, it->second.second);
        } else {
            if (cache_.size() == capacity_) {
                cache_.erase(access_list_.back());
                access_list_.pop_back();
            }
            access_list_.push_front(domain);
            cache_[domain] = {ip_address, access_list_.begin()};
        }
    }

private:
    size_t capacity_;
    list<string> access_list_;
    unordered_map<string, pair<string, list<string>::iterator>> cache_;
};
```

```
//_____________________________________________
map<string , string>root;
map<string , string>com_tld;
map<string , string>org_tld;
map<string , string>google;
map<string , string>example;
map<string , string>example_org;
map<string , string>new_domain;
map<string , map<string , string>> servers;

//_____________________________________________
void init_servers(){
 root["com"]="1.1.1.1";
 root["org"]="1.1.1.2";
 com_tld["google"]="1.1.1.3";
 com_tld["new_domain"]="1.1.1.4";
 com_tld["example"]="1.1.1.5";
 org_tld["example_org"]="1.1.1.6";
 google["www.google.com"]="192.12.24.78";
 example["www.example.com"]="162.145.19.80";
 new_domain["www.new_domain.com"]="93.48.70.98";
 example_org["www.example_org.org"]="69.69.6.9";
 servers["1.1.1.0"]=root;
 servers["1.1.1.1"]=com_tld;
 servers["1.1.1.2"]=org_tld;
 servers["1.1.1.3"]=google;
 servers["1.1.1.4"]=new_domain;
 servers["1.1.1.5"]=example;
 servers["1.1.1.6"]=example_org;
}
//_____________________________________________
```

The Data Structures and initializations are given here

Questions:

1:Discuss how the series of DNS lookups affects the overall time to retrieve a web page. What are the implications of querying multiple DNS servers sequentially?

Ans: If DNS lookups takes time it will increase the overall time to retrieve a webpage .If we can get the ip in the local DNS server itself then it wilL take less time otherwise.When sequentially querying each level adds latency also there is additional time to cache the record and maintain it as an LRU .Also failures and delays cause retransmission as additional overheads.

2. Reflect on how the size of the cache impacts the performance of the Proxy server. What are the trade-offs of different cache sizes?

Ans: Bigger the size of cache hit rate might be higher in general .Higher the hit-rate lower lower the response time also lower is the

traffic to the institutional server causing better response time. But after a level the cache size increase does not increase the time rather increase cost also LRU cache needs a handful of operations to reorder its elements which increase management time.

3. Analyze the challenges involved in integrating DNS lookups with HTTP communication. How did you handle the coordination between these processes?

Ans: *Integrating DNS lookups with HTTP communication involves challenges like managing asynchronous DNS queries, handling timeouts, and ensuring efficient caching. So I had to bypass these process by tricks like precomputing delays and failures and using them to take decisions. Also i first resolved ip and then did web caching because it would have taken code using multithreading and asynchronous processes.*

4. Consider the performance implications of cache misses (when a requested page is not in the cache). How does this influence the user's experience?

ANs: *If there is a cache miss there are latency for each level of queries and if there is failure or delay then there is significant overhead for retransmission.In our case in worst case it would take 6 seconds of waiting time.*

Question 3 :

This C++ program implements a multithreaded HTTP proxy server using the Winsock library, which allows it to relay requests from clients to target servers and send back the responses. The proxy also manages cookies across sessions and logs all communication.:

1. **Cookie Management (Ck and CkMgr Classes):**
    - **a.** The Ck class represents individual cookies with attributes for the name, value, and expiration time.
    - **b.** The CkMgr class handles cookie storage and retrieval for different client sessions, ensuring thread-safe access. It allows the proxy to maintain cookies between multiple requests from the same client, mimicking a browser's behavior.
2. **Proxy Operations (Proxy Class):**
    a. Request Handling: The proxy listens on a specified port (8080 in this case) for incoming HTTP requests from clients. Upon

receiving a request, it determines the target server by parsing the "Host" header.

b. Cookie Handling: Before forwarding the request to the target server, the proxy appends any cookies associated with the client's session. After receiving the server's response, it extracts and saves any cookies that the server sets, ensuring they are available for future requests from the same client.
c. Response Forwarding: The proxy sends the server's response back to the client, effectively acting as an intermediary that can monitor and modify the traffic.
d. Logging: Every request and response is logged into a file, allowing for easy review of all communications handled by the proxy. This includes details about the session, the request sent, and the response received.

3. **Concurrency:**
    a. The proxy server is designed to handle multiple client connections simultaneously using threads. But we could not implement traffic from multiple clients to local host port 8080 so we should just have a cookie for one user but our c++ code can handle it. Each incoming connection is managed by a separate thread, allowing the server to efficiently handle many requests in parallel without blocking.

The program acts as an intermediary between clients (like web browsers) and web servers. It intercepts HTTP requests, forwards them to the appropriate server, processes the responses (including managing cookies), and returns the data to the client. This setup is commonly used in web debugging, content filtering, or implementing additional security measures in network communications.