CS 433 Computer Networks

Final Project Report

Smart Proxy Server

November 14, 2023

Team 6:

Adit Kaushik, 21110010

Ayush Modi, 21110039

Anish Karnik, 21110098

Rutwik More, 21110133

1. Abstract

This report details the development and implementation of a smart proxy server with a primary emphasis on intelligent HTTP header manipulation as its defining characteristic. The server architecture integrates sophisticated algorithms enabling comprehensive parsing and dynamic manipulation of HTTP headers, thus empowering the system to intelligently manage network traffic. Key functionalities include header parsing and manipulation, intelligent caching mechanisms, load balancing, X-Forwarded-For capabilities, and size compression. However, the cornerstone of this system lies in its advanced header manipulation capabilities, enabling dynamic alteration of headers to optimize network performance. Through the manipulation of HTTP headers, the proxy server optimizes data flow and resource utilization. This report elucidates the technical intricacies behind header manipulation, outlining its pivotal role in defining the server as 'smart' and elucidating its impact on network efficiency. Evaluation results showcase the server's prowess in improving response times, distributing network load, taking considerations of client data and giving server specific responses. The conclusions drawn from this project shed light on the significance of header manipulation as a fundamental attribute of a smart proxy server, highlighting its potential implications in modern network infrastructures and its role in shaping the future of network optimization.

2. Introduction

In today's digitally connected world, the efficient management of network resources stands very important for optimal performance and security. This project endeavors to address these challenges through the development and implementation of a cutting-edge smart proxy server.

Proxy servers have long served as intermediaries between clients and the internet, facilitating requests, enhancing security, and improving performance. However, the evolution of network demands has mandated a new breed of proxies that transcend conventional functionalities. This project aims to introduce a paradigm shift by engineering a smart proxy server—a system equipped with advanced mechanisms, parsing techniques and particularly focusing on dynamic HTTP header manipulation.

At the core of this project lies the recognition that HTTP headers wield substantial influence over network communication. Leveraging this insight, the proxy server is designed to dynamically parse, analyze, and manipulate these headers in real-time. By doing so, it gets the hold over network traffic, enabling efficient caching strategies,and enhanced load balancing.

The key objective of this report is to know the details of this smart proxy server's architecture, focusing primarily on its defining feature: intelligent HTTP header manipulation. Additionally, the report presents comprehensive evaluations showcasing the server's efficacy in improving network performance, enhanced caching and various use cases of header manipulations.

Through this exploration, this report details the technical nuances of the smart proxy server. The ensuing sections provide the architectural design, functionalities, implementation specifics, evaluation methodologies, and results, culminating in insights into the implications and future prospects of this approach to network optimization.

3. Concepts Learned

The development of the smart proxy server led to a profound exploration of several foundational concepts integral to modern network management and optimization. This section encapsulates the pivotal insights garnered during the project's lifecycle.

1. **HTTP Header Dynamics**: Understanding the intricate dynamics of HTTP headers emerged as a fundamental prerequisite. Delving into header structures, their roles in client-server communication, and the potential for manipulation provided the groundwork for advanced functionalities within the proxy server.
2. **Smart Header Manipulation Techniques**: Building intelligent algorithms capable of real-time header analysis and manipulation was a significant learning curve. Crafting algorithms that could interpret headers, make informed decisions, and dynamically alter them to optimize network performance required a nuanced understanding of header attributes and their implications.
3. **Load Balancing Strategies**: Implementing load balancing mechanisms within the proxy server necessitated in-depth knowledge of traffic distribution techniques. This involved exploring various algorithms to distribute incoming requests across multiple servers efficiently, ensuring optimal resource utilization.
4. **Caching Strategies**: Designing an effective caching mechanism involved learning about caching policies, eviction strategies, and data storage optimizations. Crafting strategies to store and retrieve frequently accessed data locally within the proxy server proved crucial in reducing latency and bandwidth consumption.

The assimilation of these concepts not only facilitated the development of the smart proxy server but also provided a comprehensive understanding of the intricate interplay between network protocols and optimization strategies. These insights form a solid foundation for further advancements in network management and optimization paradigms.

4. Technologies and Tools Used

- Python: We have used the Python programming language for the creation of the smart proxy server. The primary reason for this choice is the wide-range of libraries and frameworks that Python supports and that we can leverage these libraries with conviction for the various features we have added to our proxy server.

- Libraries: socket, threading, logging, select, time, re

- Socket: This library is required for the creation of network sockets for communication. Make a server socket to receive incoming connections. Create client sockets to connect to remote servers. Sockets are used to send and receive data.
- Threading: This library is used to create and manage threads, allowing for concurrent job execution. Handle many client connections at the same time by starting a new thread for each one.
- This library generates log messages that provide information about the program's progress. Important events should be recorded, such as when the proxy server boots up, accepts a connection, receives/sends data, or blocks incoming requests.

- Postman: For testing and monitoring the API calls made to and from the proxy server.
- Curl: For sending requests to the proxy server through terminal

5. Basic Architectural Overview of a Proxy Server

5.1 Definitions:

1. Proxy Server - A proxy server acts as an intermediary between client devices, such as a computer or smartphone, and the internet. It receives requests from clients seeking resources (web pages, files, or other services) and forwards these requests on their behalf to the destination server. Upon receiving the response from the destination server, the proxy server then relays this information back to the client.
2. Reverse Proxy - A reverse proxy is similar to a normal proxy, the difference being in the relative positioning of the proxy in between the clients and servers. In a reverse proxy, the proxy sits in front of an origin server, unlike in forward proxy where it sits in between the client and internet, and intercepts requests made by the client at the **network edge** [1]. It then sends requests to and receives responses from the origin server.

The smart proxy server we have developed is built upon the concept of a reverse proxy.
Below is a diagrammatic representation of general reverse proxy.

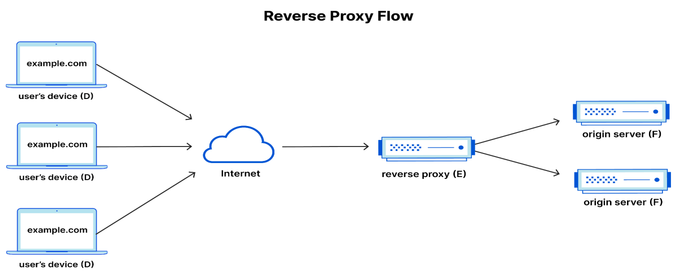


Figure 1 [1]

Thus, our smart proxy server can be seen as an organization-specific proxy that handles all the network traffic coming into and going out of the organization. The core functionalities of the proxy that we have implemented are mostly based on http header manipulations, and reverse proxies are often employed in use cases involving extensive header manipulations. Reverse proxies can parse and modify headers from client requests before sending them to the backend servers. And thus, they are also extensively used in tasks which involve features such as load balancing.

Henceforth, we have made the decision for the choice of the type of proxy and the specific features to be implemented in our proxy based on the above mentioned reasonings. In the next section, we delve deeper into the specific features that we have implemented for our smart proxy server.

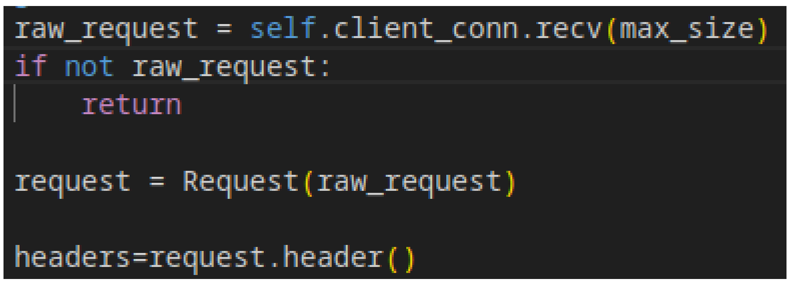
6. Functionalities

6.1 Smart HTTP Request Parsing and Header Manipulation

The main feature which makes our proxy smart is its ability to parse http requests and manipulate the headers from the requests. This can be broken down into two steps, the first of which is request parsing and second is header manipulation.

6.1.1 HTTP Request Parsing:

The raw request made by the client is received by the proxy through the connection socket. This request is then sent as an argument to the Request class, which handles and parses the raw request to get information about the request for further processing and manipulation.

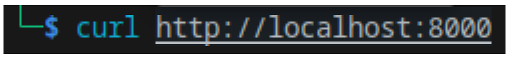


```python
raw_request = self.client_conn.recv(max_size)
if not raw_request:
    return

request = Request(raw_request)

headers=request.header()
```

The code inside the Request class is explained below:

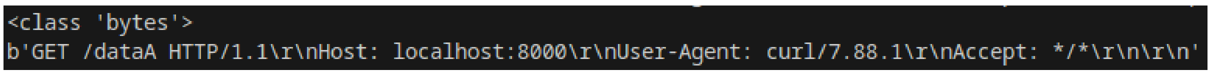
The raw request that we receive is in the form of bytes. Below is an example of a raw request made by the client.

The request sent through the terminal: (Making requests will further be explained in the Implementation Section of this report)



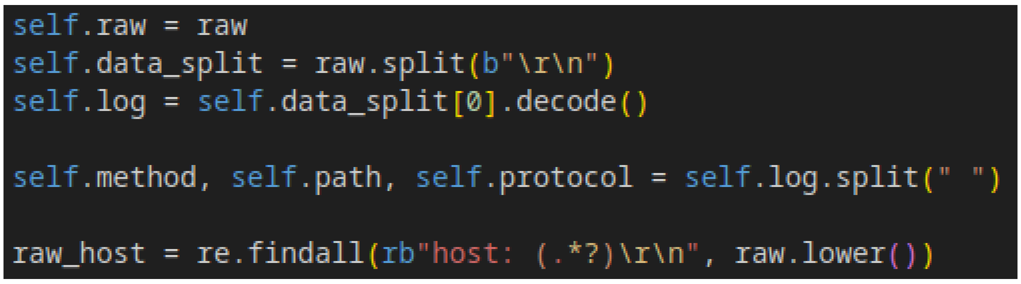
```
└─$ curl http://localhost:8000
```

The request received is as shown below. Note the class 'bytes' which denotes that the request received is in the bytes form.



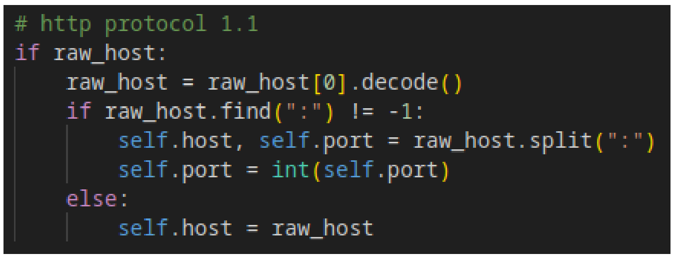
```
<class 'bytes'>
b'GET /dataA HTTP/1.1\r\nHost: localhost:8000\r\nUser-Agent: curl/7.88.1\r\nAccept: */*\r\n\r\n'
```

So firstly, this raw request is decoded into string form so that we can use this string for parsing the request. This is then stored in the log variable through which we extract the information about the method of the request, the path, and the protocol used.

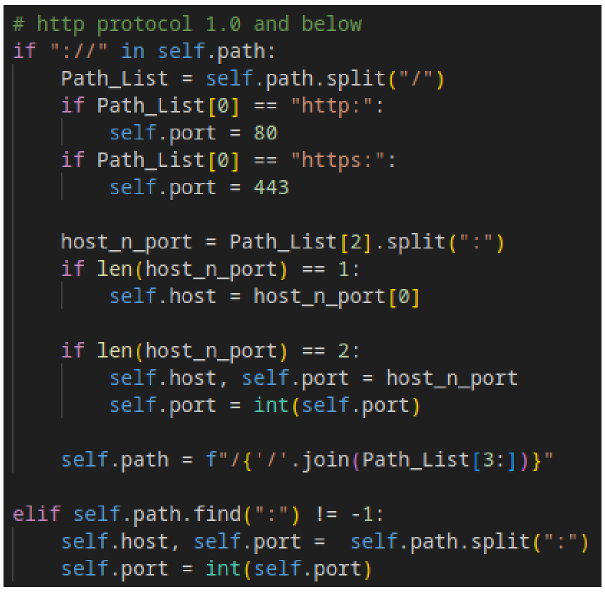


```python
self.raw = raw
self.data_split = raw.split(b"\r\n")
self.log = self.data_split[0].decode()

self.method, self.path, self.protocol = self.log.split(" ")

raw_host = re.findall(rb"host: (.*?)\r\n", raw.lower())
```

The proxy then based upon the version of http protocol, processes the request accordingly. The proxy will then extract the host and port numbers.

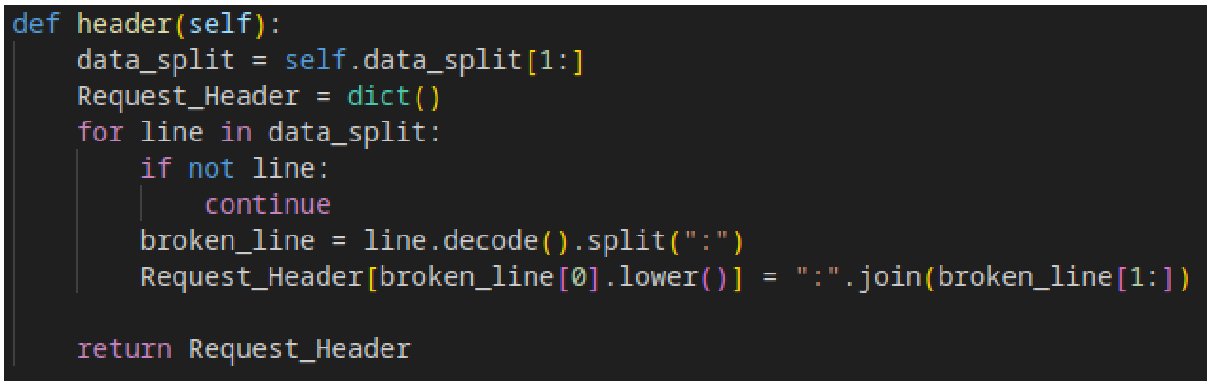


```python
# http protocol 1.1
if raw_host:
    raw_host = raw_host[0].decode()
    if raw_host.find(":") != -1:
        self.host, self.port = raw_host.split(":")
        self.port = int(self.port)
    else:
        self.host = raw_host
```



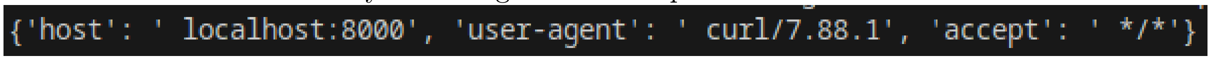
```python
# http protocol 1.0 and below
if "://" in self.path:
    Path_List = self.path.split("/")
    if Path_List[0] == "http:":
        self.port = 80
    if Path_List[0] == "https:":
        self.port = 443

    host_n_port = Path_List[2].split(":")
    if len(host_n_port) == 1:
        self.host = host_n_port[0]

    if len(host_n_port) == 2:
        self.host, self.port = host_n_port
        self.port = int(self.port)

    self.path = f"/{'/'.join(Path_List[3:])}"

elif self.path.find(":") != -1:
    self.host, self.port =  self.path.split(":")
    self.port = int(self.port)
```

Lastly, this is the header function inside the Request class, which returns the headers associated with the http request made by the client. It returns a python dictionary in which the key is the header name, and the value is the value of the header respectively.



```python
def header(self):
    data_split = self.data_split[1:]
    Request_Header = dict()
    for line in data_split:
        if not line:
            continue
        broken_line = line.decode().split(":")
        Request_Header[broken_line[0].lower()] = ":".join(broken_line[1:])

    return Request_Header
```

This is the headers dictionary that we get for the request we showed above:



```
{'host': ' localhost:8000', 'user-agent': ' curl/7.88.1', 'accept': ' */*'}
```

Hence, up till now, we have successfully parsed the raw request made by the client and have extracted all the useful information from the request which will be used by the proxy for its implementing its various functionalities.

6.1.2 Header Manipulations:

After parsing the http request, we now have all the required information about the request and thus we can proceed with handling this request and performing the required manipulations on the headers.

We have performed both header addition and header removal to demonstrate header manipulation by our proxy server.

These are the header manipulations that our proxy server performs:

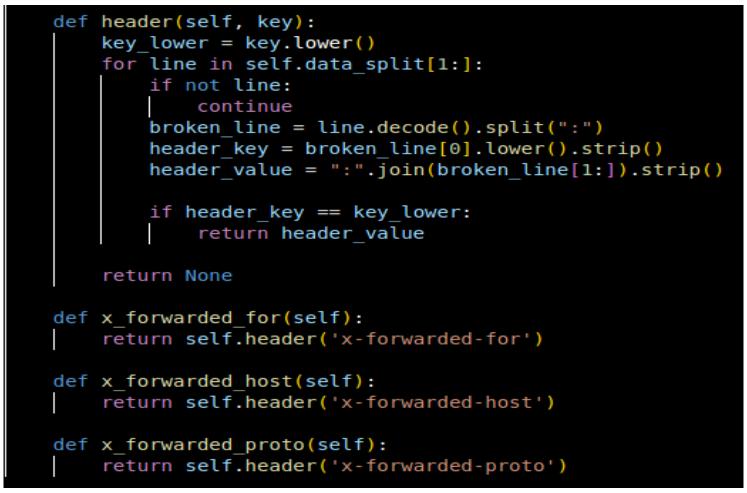
6.1.2.1 X-Forwarded-For:

The X-Forwarded-For (XFF) request header is a standard header for identifying a client's originating IP address when connecting to a web server via a proxy server. When a client connects to a server directly, the client's IP address is transmitted to the server (and is frequently recorded in server access logs). However, if a client connection goes via any forward or reverse proxies, the server only sees the IP address of the last proxy, which is generally ineffective. As a result, the X-Forwarded-For request header is utilized to send a more usable client IP address to the server.

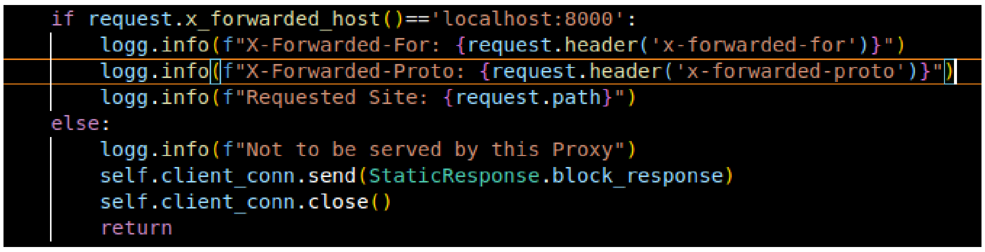
The X-Forwarded For is a list which contains the IP address of the client and all the proxies through which the request went through. As the request moves forward to proxies, the IP addresses of the previous proxy are inserted into the list. Hence, whenever the server wants to access the client IP address it selects the first element from the list.

We have also implemented the X-Forwarded Host and X-Forwarded Proto in the proxy server to keep track of the protocols and the requested web addresses which can be used by the server for specific purposes.

The code for implementing this includes parsing headers, decoding and storing key value pairs for all the three X-Forwarded inputs. Then we have created functions to get the values of the respective inputs, i.e. for IPs, host and protocol.



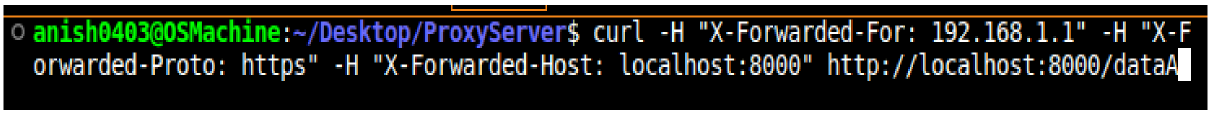
```
def header(self, key):
    key_lower = key.lower()
    for line in self.data_split[1:]:
        if not line:
            continue
        broken_line = line.decode().split(":")
        header_key = broken_line[0].lower().strip()
        header_value = ":".join(broken_line[1:]).strip()

        if header_key == key_lower:
            return header_value

    return None

def x_forwarded_for(self):
    return self.header('x-forwarded-for')

def x_forwarded_host(self):
    return self.header('x-forwarded-host')

def x_forwarded_proto(self):
    return self.header('x-forwarded-proto')
```



```
if request.x_forwarded_host()=='localhost:8000':
    logg.info(f"X-Forwarded-For: {request.header('x-forwarded-for')}")
    logg.info(f"X-Forwarded-Proto: {request.header('x-forwarded-proto')}")
    logg.info(f"Requested Site: {request.path}")
else:
    logg.info(f"Not to be served by this Proxy")
    self.client_conn.send(StaticResponse.block_response)
    self.client_conn.close()
    return
```

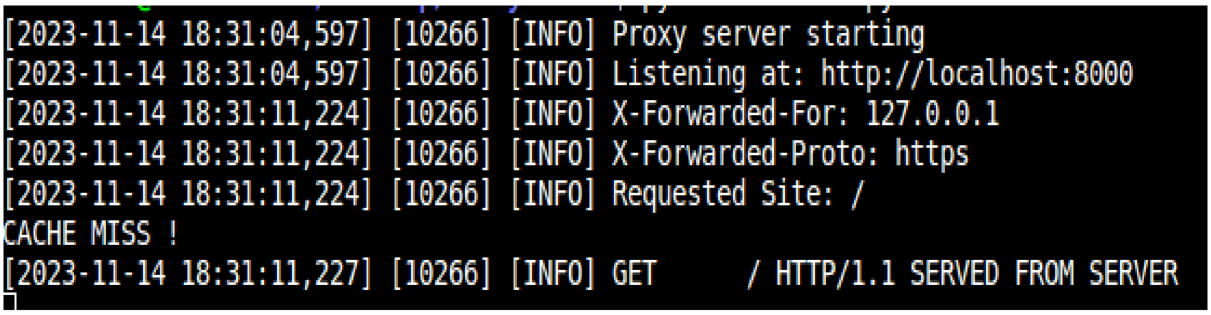
Syntax:
X-Forwarded-For: <client>, <proxy1>, <proxy2>

Now, we will show how the headers need to be modified for X-Forwarded implementation. This is how we would be giving command line inputs for its execution.



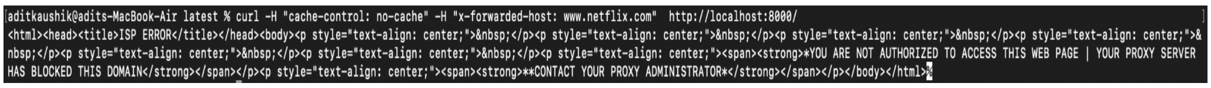
```
anish0403@OSMachine:~/Desktop/ProxyServer$ curl -H "X-Forwarded-For: 192.168.1.1" -H "X-F
orwarded-Proto: https" -H "X-Forwarded-Host: localhost:8000" http://localhost:8000/dataA
```

On giving the command, the proxy will parse headers and extract the necessary information from it. We have displayed all the three details to ensure proper working of the proxy server and if it executes the desirable implementation or not. The output seems as shown below.

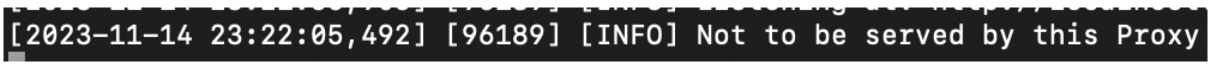


```
[2023-11-14 18:31:04,597] [10266] [INFO] Proxy server starting
[2023-11-14 18:31:04,597] [10266] [INFO] Listening at: http://localhost:8000
[2023-11-14 18:31:11,224] [10266] [INFO] X-Forwarded-For: 127.0.0.1
[2023-11-14 18:31:11,224] [10266] [INFO] X-Forwarded-Proto: https
[2023-11-14 18:31:11,224] [10266] [INFO] Requested Site: /
CACHE MISS !
[2023-11-14 18:31:11,227] [10266] [INFO] GET      / HTTP/1.1 SERVED FROM SERVER
```

As we can see we have accessed the site other than the desired(localhost in this case), we won't get the response from the proxy server.



```
aditkaushik@adits-MacBook-Air latest % curl -H "cache-control: no-cache" -H "x-forwarded-host: www.netflix.com"  http://localhost:8000/
<html><head><title>ISP ERROR</title></head><body><p style="text-align: center;">&nbsp;</p><p style="text-align: center;">&nbsp;</p><p style="text-align: center;">&nbsp;</p><p style="text-align: center;">&
nbsp;</p><p style="text-align: center;">&nbsp;</p><p style="text-align: center;">&nbsp;</p><p style="text-align: center;"><span><strong>*YOU ARE NOT AUTHORIZED TO ACCESS THIS WEB PAGE | YOUR PROXY SERVER
HAS BLOCKED THIS DOMAIN</strong></span></p><p style="text-align: center;"><span><strong>**CONTACT YOUR PROXY ADMINISTRATOR*</strong></span></p></body></html>%
```

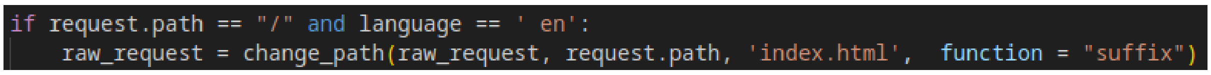


```
[2023-11-14 23:22:05,492] [96189] [INFO] Not to be served by this Proxy
```

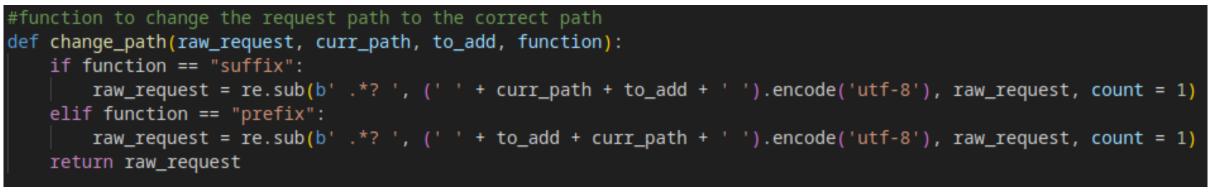
Although there are also some security concerns associated with X-Forwarded For headers. The user's privacy may be hindered if the proxy is untrustworthy. If the proxy is malicious it may spoof the header and store sensitive information of the clients. Any usage of X-Forwarded-For for security purposes (such as rate restriction or IP-based access control) must only use IP addresses added by a trusted proxy. Using untrustworthy values can lead to rate-limiter bypass, access-control bypass, and other negative security or availability implications.

6.1.2.2 Path Transforms:

The proxy also performs path transform on the path that is requested by the client.



```python
if request.path == "/" and language == ' en':
    raw_request = change_path(raw_request, request.path, 'index.html',  function = "suffix")
```

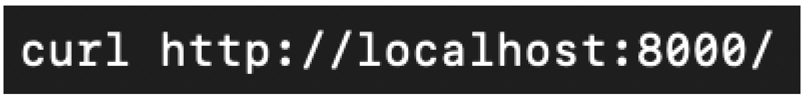


```python
#function to change the request path to the correct path
def change_path(raw_request, curr_path, to_add, function):
    if function == "suffix":
        raw_request = re.sub(b' .*? ', (' ' + curr_path + to_add + ' ').encode('utf-8'), raw_request, count = 1)
    elif function == "prefix":
        raw_request = re.sub(b' .*? ', (' ' + to_add + curr_path + ' ').encode('utf-8'), raw_request, count = 1)
    return raw_request
```

As we can see, we have used regular expressions to append to the path of the original raw_request.

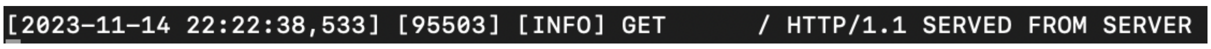
It is demonstrated using this example below:

When the client requests for the path "/" using the following,



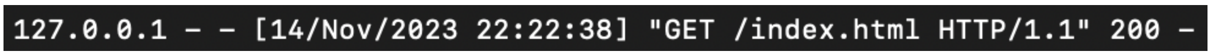
```
curl http://localhost:8000/
```

This is what the proxy server receives—



```
[2023-11-14 22:22:38,533] [95503] [INFO] GET       / HTTP/1.1 SERVED FROM SERVER
```

The proxy server fetches the response from the main server. But before forwarding the request to the main server, it changes the path to /index.html. This is because proxy understands that the client actually wants the default page and the default page with the main servers is actually named index.html. Therefore, this is what the main server receives—

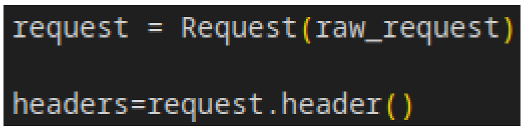


```
127.0.0.1 - - [14/Nov/2023 22:22:38] "GET /index.html HTTP/1.1" 200 -
```

6.1.2.3 “accept-language" header:

The accept-language header is a header which is widely used by websites which serve content in more than one language. Our proxy server intercepts in between and smartly handles the client requests which consist of this header.

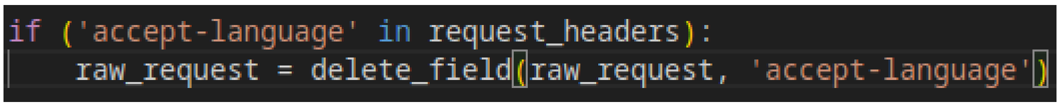
In this case, the type of header transform we are performing is header removal. Whenever a client sends a request which consists of this particular header, the proxy server intercepts and decides how to handle this request by itself, and then sends the request to one of the servers after removing this “accept-language" header field from the client request. That is, the server would receive the request as if the client had never specified the “accept-language" header field in its request. Thus as a result, now the server has one header less on which it needs to worry about.



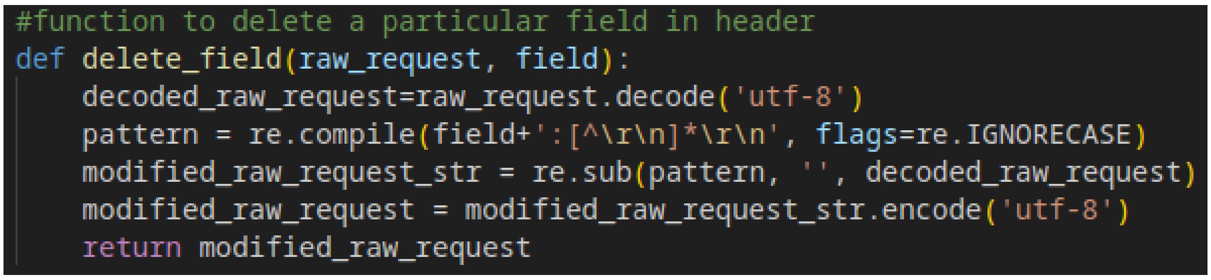
```python
request = Request(raw_request)

headers=request.header()
```

Through this we have gotten all the headers associated with the request stored in the headers dictionary.

Below is how we handle the header manipulation in this case:



```python
if ('accept-language' in request_headers):
    raw_request = delete_field(raw_request, 'accept-language')
```



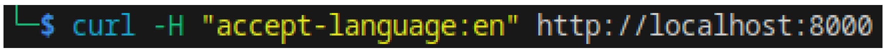
```python
#function to delete a particular field in header
def delete_field(raw_request, field):
    decoded_raw_request=raw_request.decode('utf-8')
    pattern = re.compile(field+':[^\r\n]*\r\n', flags=re.IGNORECASE)
    modified_raw_request_str = re.sub(pattern, '', decoded_raw_request)
    modified_raw_request = modified_raw_request_str.encode('utf-8')
    return modified_raw_request
```

Firstly, the raw request is decoded to bring it to string form, and then we have used the regular expressions library in python for matching the required pattern and removing it from the string.

This modified string is then encoded back into the bytes form so that it can be used normally for sending this request to the server.

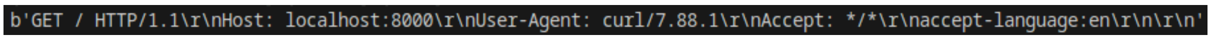
Below is an example demonstrating this:

Request made by client: (Requests explained in the implementation section)



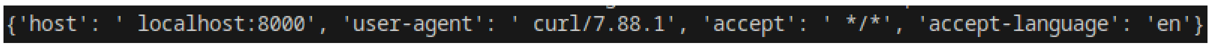
```
└─$ curl -H "accept-language:en" http://localhost:8000
```

Raw Request received by proxy in bytes form:



```
b'GET / HTTP/1.1\r\nHost: localhost:8000\r\nUser-Agent: curl/7.88.1\r\nAccept: */*\r\naccept-language:en\r\n\r\n'
```

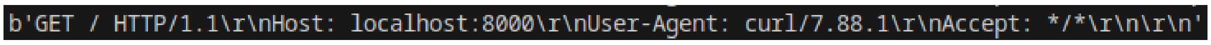
Headers:



```
{'host': ' localhost:8000', 'user-agent': ' curl/7.88.1', 'accept': ' */*', 'accept-language': 'en'}
```

Note the “accept-language" header in the raw request.

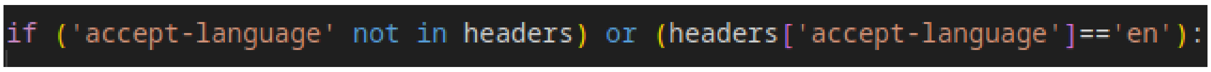
Now, after performing this header manipulation and removing the “accept-language" header, the raw request has become:



```
b'GET / HTTP/1.1\r\nHost: localhost:8000\r\nUser-Agent: curl/7.88.1\r\nAccept: */*\r\n\r\n'
```

Thus, as we can see, the header “accept-language" has been removed from the request.

Now, as we have removed this header from the request, the proxy server will make the decision on how to handle this request based on the “accept-language" header. Below is how this is done:



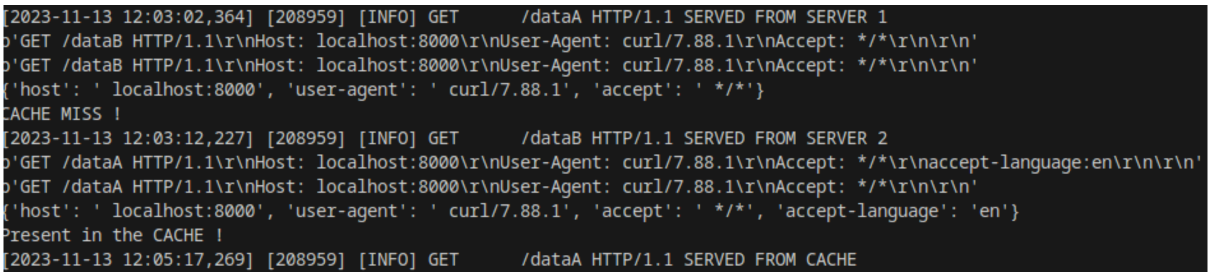
```
if ('accept-language' not in headers) or (headers['accept-language']=='en'):
```

That is, if the “accept-language" header is not specified by the user, or the value of this header is “en", that is English, then we send this request to either server 1 or server 2.

In the else condition above, that is, if the value of the header “accept-language" is anything other than “en", for example, “fr" for french, then we send this request to server 3.
The implementation of sending the request to the server is explained in the load balancing part of this same section below. In this part, we are focusing on explaining the proxy's role in firstly removal of the header, and then managing how this removal of header will still send the request to appropriate servers.

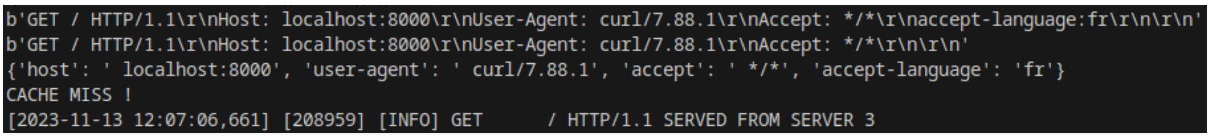
Demonstration:

When the accept-language header is set to “accept-language:en", it is served by server1



```
[2023-11-13 12:03:02,364] [208959] [INFO] GET      /dataA HTTP/1.1 SERVED FROM SERVER 1
b'GET /dataB HTTP/1.1\r\nHost: localhost:8000\r\nUser-Agent: curl/7.88.1\r\nAccept: */*\r\n\r\n'
b'GET /dataB HTTP/1.1\r\nHost: localhost:8000\r\nUser-Agent: curl/7.88.1\r\nAccept: */*\r\n\r\n'
{'host': ' localhost:8000', 'user-agent': ' curl/7.88.1', 'accept': ' */*'}
CACHE MISS !
[2023-11-13 12:03:12,227] [208959] [INFO] GET      /dataB HTTP/1.1 SERVED FROM SERVER 2
b'GET /dataA HTTP/1.1\r\nHost: localhost:8000\r\nUser-Agent: curl/7.88.1\r\nAccept: */*\r\naccept-language:en\r\n\r\n'
b'GET /dataA HTTP/1.1\r\nHost: localhost:8000\r\nUser-Agent: curl/7.88.1\r\nAccept: */*\r\n\r\n'
{'host': ' localhost:8000', 'user-agent': ' curl/7.88.1', 'accept': ' */*', 'accept-language': 'en'}
Present in the CACHE !
[2023-11-13 12:05:17,269] [208959] [INFO] GET      /dataA HTTP/1.1 SERVED FROM CACHE
```

When “accept-language:fr", it is served by server3



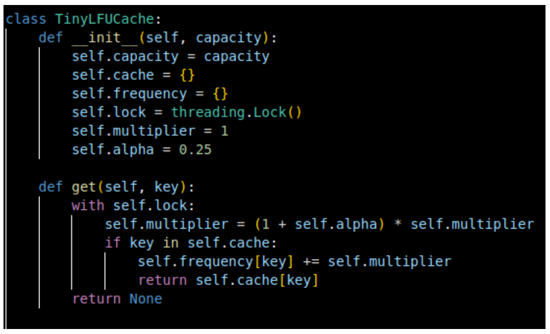
```
b'GET / HTTP/1.1\r\nHost: localhost:8000\r\nUser-Agent: curl/7.88.1\r\nAccept: */*\r\naccept-language:fr\r\n\r\n'
b'GET / HTTP/1.1\r\nHost: localhost:8000\r\nUser-Agent: curl/7.88.1\r\nAccept: */*\r\n\r\n'
{'host': ' localhost:8000', 'user-agent': ' curl/7.88.1', 'accept': ' */*', 'accept-language': 'fr'}
CACHE MISS !
[2023-11-13 12:07:06,661] [208959] [INFO] GET      / HTTP/1.1 SERVED FROM SERVER 3
```

Thus, in this way, our proxy server has made the decision on which server to send this request to and has relieved the servers from having to handle this header by changing the raw request and removing this header field from the request.

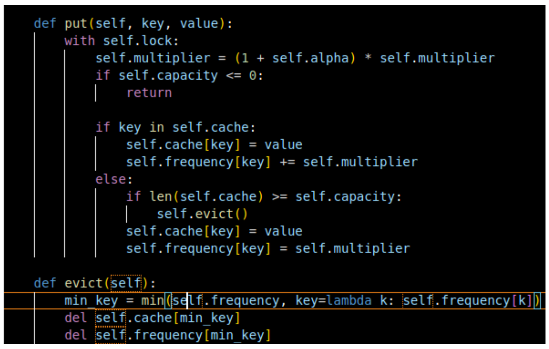
6.2 Intelligent Caching Mechanism

We use the TinyLFU caching mechanism to implement the caching mechanism. The cache consists of two main components: cache and frequency, both of these are dictionaries and store key value pairs. Frequency keeps track of the access frequency of each key. The cache also contains multiplier and alpha values, where the multiplier updates its value as we request entities from the cache. Basically, alpha can also be viewed as the aging factor.

The get method is called when the proxy server receives a request. It is responsible for checking whether the requested content is present in the cache. If the key is found in the cache, the access frequency of the key is updated using the TinyLFU algorithm, and the corresponding value is returned. If the key is not in the cache, the method returns null result. The put method is called when the proxy server retrieves content from a backend server and wants to store it in the cache. Similar to the get method, it updates the access frequency using the TinyLFU algorithm. If the key is already in the cache, the value is updated, and the frequency is incremented. If the key is not in the cache, and the cache is at capacity, the evict method is called to identify and remove the least frequently used item.

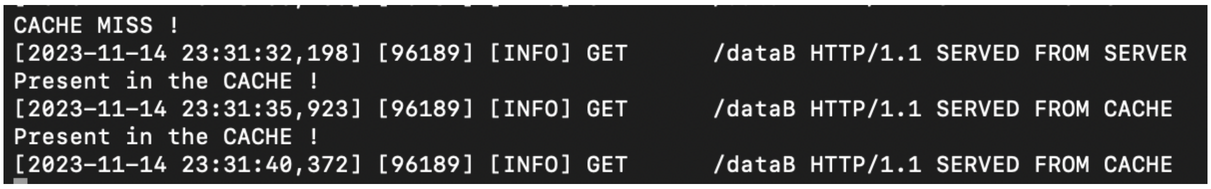


```python
class TinyLFUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}
        self.frequency = {}
        self.lock = threading.Lock()
        self.multiplier = 1
        self.alpha = 0.25

    def get(self, key):
        with self.lock:
            self.multiplier = (1 + self.alpha) * self.multiplier
            if key in self.cache:
                self.frequency[key] += self.multiplier
                return self.cache[key]
        return None
```

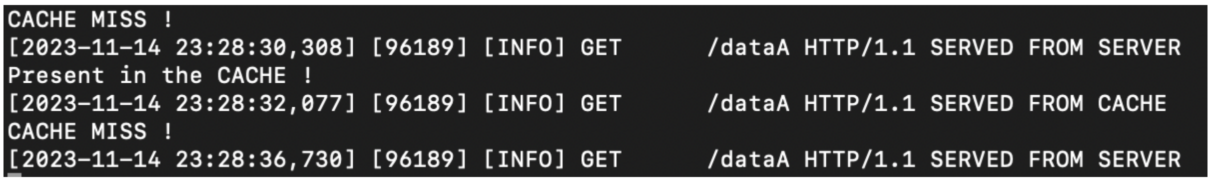


```python
    def put(self, key, value):
        with self.lock:
            self.multiplier = (1 + self.alpha) * self.multiplier
            if self.capacity <= 0:
                return

            if key in self.cache:
                self.cache[key] = value
                self.frequency[key] += self.multiplier
            else:
                if len(self.cache) >= self.capacity:
                    self.evict()
                self.cache[key] = value
                self.frequency[key] = self.multiplier

    def evict(self):
        min_key = min(self.frequency, key=lambda k: self.frequency[k])
        del self.cache[min_key]
        del self.frequency[min_key]
```

The connection handle class makes use of the TinyLFU caching mechanism. The proxy uses this to implement a cache for itself and accordingly gives responses if it is a hit or miss. The connection handle and how it uses TinyLFU is described in the Implementation section of the report.



```
CACHE MISS !
[2023-11-14 23:31:32,198] [96189] [INFO] GET      /dataB HTTP/1.1 SERVED FROM SERVER
Present in the CACHE !
[2023-11-14 23:31:35,923] [96189] [INFO] GET      /dataB HTTP/1.1 SERVED FROM CACHE
Present in the CACHE !
[2023-11-14 23:31:40,372] [96189] [INFO] GET      /dataB HTTP/1.1 SERVED FROM CACHE
```



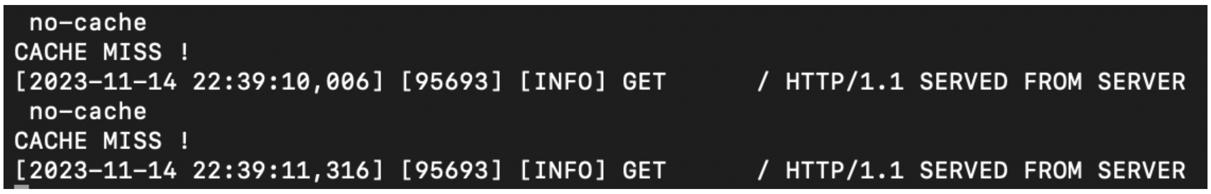
```
CACHE MISS !
[2023-11-14 23:28:30,308] [96189] [INFO] GET      /dataA HTTP/1.1 SERVED FROM SERVER
Present in the CACHE !
[2023-11-14 23:28:32,077] [96189] [INFO] GET      /dataA HTTP/1.1 SERVED FROM CACHE
CACHE MISS !
[2023-11-14 23:28:36,730] [96189] [INFO] GET      /dataA HTTP/1.1 SERVED FROM SERVER
```

In summary, this proxy server's caching mechanism employs the TinyLFU algorithm to efficiently manage a cache of defined capacity, keeping frequently visited content and making eviction choices based on access frequencies. This reduces the load on backend systems while also improving response times for repeat requests.

6.3 Load Balancing

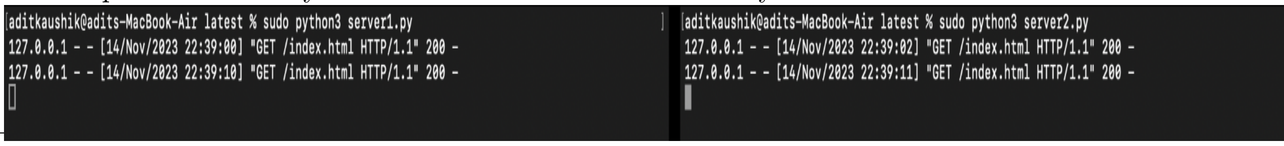
Our proxy server also performs static load balancing and simple dynamic load balancing. There are three instances of load balancing present—

1. When a client requests for the default page "/" in english, the servers 1 and 2 are given turns one by one to serve the client. For example, the first client is served by server 1, then the next client is served by 2 and so on. This is simple dynamic load balancing. When the proxy server receives 2 "GET /" requests,—



```
 no-cache
CACHE MISS !
[2023-11-14 22:39:10,006] [95693] [INFO] GET      / HTTP/1.1 SERVED FROM SERVER
 no-cache
CACHE MISS !
[2023-11-14 22:39:11,316] [95693] [INFO] GET      / HTTP/1.1 SERVED FROM SERVER
```

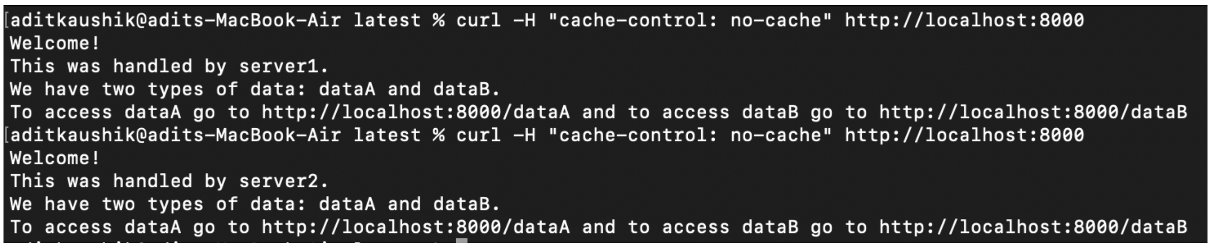
First request is served by server1 and the second is served by server 2—



```
aditkaushik@adits-MacBook-Air latest % sudo python3 server1.py
127.0.0.1 - - [14/Nov/2023 22:39:00] "GET /index.html HTTP/1.1" 200 -
127.0.0.1 - - [14/Nov/2023 22:39:10] "GET /index.html HTTP/1.1" 200 -
```

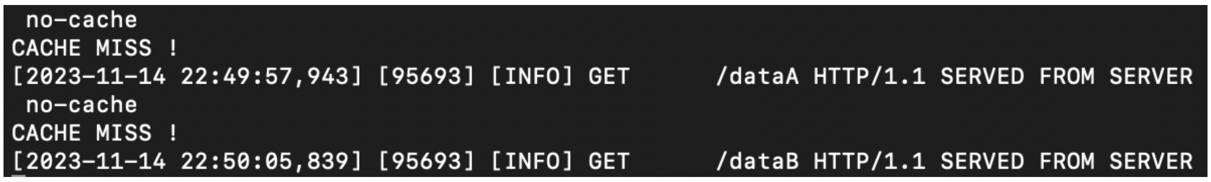
```
aditkaushik@adits-MacBook-Air latest % sudo python3 server2.py
127.0.0.1 - - [14/Nov/2023 22:39:02] "GET /index.html HTTP/1.1" 200 -
127.0.0.1 - - [14/Nov/2023 22:39:11] "GET /index.html HTTP/1.1" 200 -
```

And this is what the clients receives—



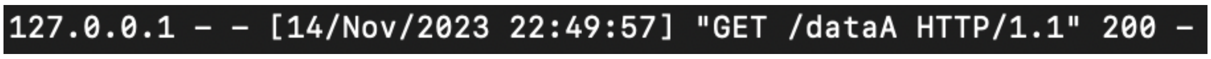
```
aditkaushik@adits-MacBook-Air latest % curl -H "cache-control: no-cache" http://localhost:8000
Welcome!
This was handled by server1.
We have two types of data: dataA and dataB.
To access dataA go to http://localhost:8000/dataA and to access dataB go to http://localhost:8000/dataB
aditkaushik@adits-MacBook-Air latest % curl -H "cache-control: no-cache" http://localhost:8000
Welcome!
This was handled by server2.
We have two types of data: dataA and dataB.
To access dataA go to http://localhost:8000/dataA and to access dataB go to http://localhost:8000/dataB
```

1. We have two types of data — dataA and dataB. When the client asks for dataA, the proxy always forwards the request to server 1 and when the client asks for dataB, the proxy always forwards the request to server 2. This is a type of static load balancing. Also, dataA is much more dynamic than dataB. This means that dataA changes frequently. Therefore, the proxy server can only store dataA in cache for only 5 seconds and then, whenever the client requests for dataA, it is fetched from server1 again. When 2 clients request for dataA and dataB, this is what the proxy server gets—



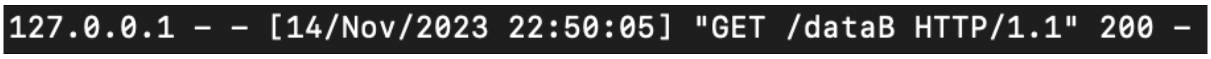
```
 no-cache
CACHE MISS !
[2023-11-14 22:49:57,943] [95693] [INFO] GET      /dataA HTTP/1.1 SERVED FROM SERVER
 no-cache
CACHE MISS !
[2023-11-14 22:50:05,839] [95693] [INFO] GET      /dataB HTTP/1.1 SERVED FROM SERVER
```

This is what server1 gets—



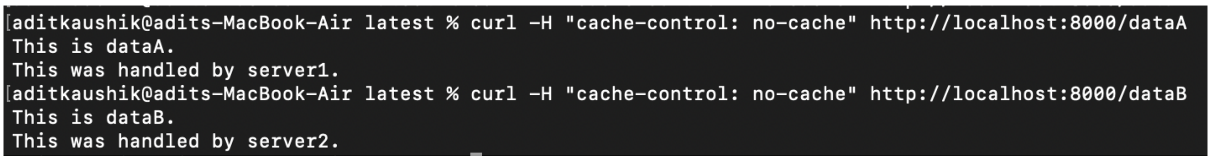
```
127.0.0.1 - - [14/Nov/2023 22:49:57] "GET /dataA HTTP/1.1" 200 -
```

This is what server2 gets—



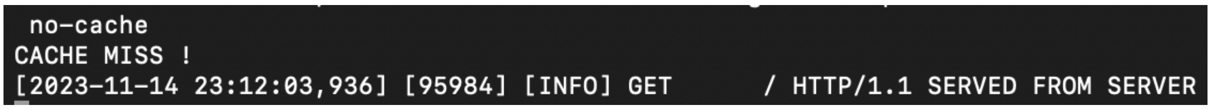
```
127.0.0.1 - - [14/Nov/2023 22:50:05] "GET /dataB HTTP/1.1" 200 -
```

And, this is what the clients receive—



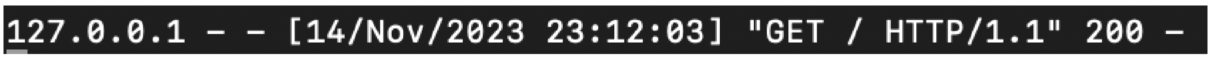
```
[aditkaushik@adits-MacBook-Air latest % curl -H "cache-control: no-cache" http://localhost:8000/dataA
This is dataA.
This was handled by server1.
[aditkaushik@adits-MacBook-Air latest % curl -H "cache-control: no-cache" http://localhost:8000/dataB
This is dataB.
This was handled by server2.
```

1. When a client asks for a response in English, the request is forwarded to one of the servers — server1 or server2 but when the client asks for response in any other language, it is directed to server3, no matter what path the client asks for. This is because in production, the requests for responses in english would be much larger than for other languages. The requests for response in other languages would be too few and rare. Therefore, the proxy does not even cache the responses given in other languages as they would be wasting the proxy resources. When the client requests for a response in hindi, this is what the proxy receives—



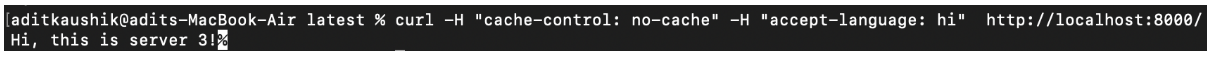
```
 no-cache
CACHE MISS !
[2023-11-14 23:12:03,936] [95984] [INFO] GET      / HTTP/1.1 SERVED FROM SERVER
```

This is what server 3 receives—



```
127.0.0.1 - - [14/Nov/2023 23:12:03] "GET / HTTP/1.1" 200 -
```

And this is what the client receives—



```
[aditkaushik@adits-MacBook-Air latest % curl -H "cache-control: no-cache" -H "accept-language: hi"  http://localhost:8000/
 Hi, this is server 3!%
```

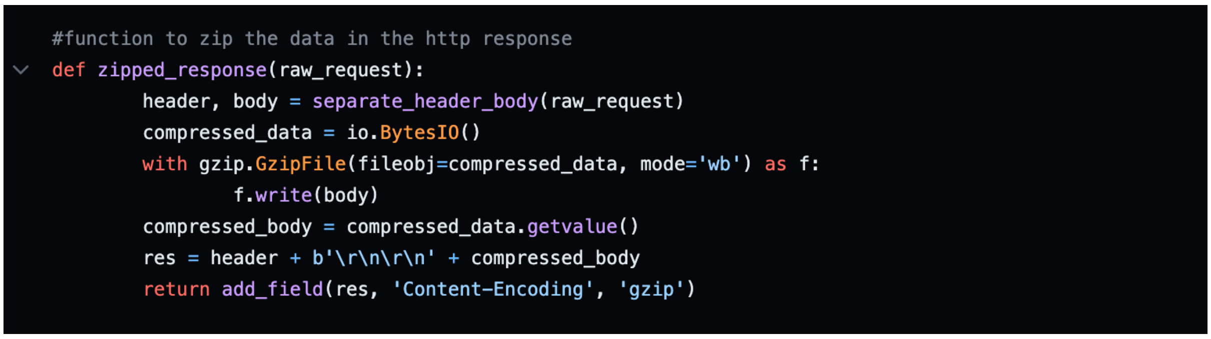
6.4 Size Compression / Content Encoding

In our smart proxy server project, the implementation of content encoding plays a pivotal role in optimizing data transmission over the network. Content encoding involves compressing the response data before transmitting it to the client, thereby reducing the amount of data transferred and improving overall network efficiency. Below, we elaborate on the implementation and impact of content encoding in our project.

6.4.1 Gzip Compression Overview:

Gzip is a widely-used compression algorithm that reduces the size of transmitted data by compressing it before sending and decompressing it upon reception. This compression technique is particularly effective for text-based data, such as HTML, CSS, and JavaScript files, which often constitute a significant portion of web content.

6.4.2 Implementation Code:



```python
#function to zip the data in the http response
def zipped_response(raw_request):
        header, body = separate_header_body(raw_request)
        compressed_data = io.BytesIO()
        with gzip.GzipFile(fileobj=compressed_data, mode='wb') as f:
                f.write(body)
        compressed_body = compressed_data.getvalue()
        res = header + b'\r\n\r\n' + compressed_body
        return add_field(res, 'Content-Encoding', 'gzip')
```

The zipped_response function takes the raw HTTP request as input and separates its header and body using the separate_header_body function (not provided in the code snippet). It then creates an in-memory byte stream (io.BytesIO()) to store the compressed data. The gzip. GzipFile is used to compress the body of the HTTP response. The mode='wb' specifies that the file is opened for writing in binary mode. The compressed data is obtained using compressed_data.getvalue(). Finally, the compressed header and body are combined with the appropriate HTTP protocol delimiter (b'\r\n\r\n') and returned as the compressed response.

6.4.3 Benefits and Impact:

Reduced Data Size: Gzip compression significantly reduces the size of the transmitted data, leading to faster data transfer and reduced bandwidth usage.

Improved Load Times: Smaller data size results in quicker load times for web pages and resources, enhancing the overall user experience.

Scalability: The implementation of content encoding contributes to the scalability of the proxy server, allowing it to handle a larger volume of requests with optimized resource utilization.

7. Implementation

7.1 Code Structure:

GitHub Repository Link: `https://github.com/anish-karnik/CN_Project`

The code of the proxy server is in the proxy.py file in the repository, which is the main file of our proxy server architecture.

We have created 3 servers, named server1, server2, and server3 respectively.
The files of the three servers are as per their names accordingly in the project repository.
These three servers could be stimulated to be a part of an organization to which our proxy server serves for.

The servers will be requested turn by turn when we ask to connect with the server returning index.html page. When a client needs dataA, then specifically server 1 will be called and server 2 will be called for dataB. For data other than these, the proxy server will not serve the client.

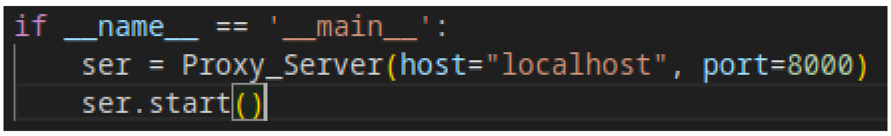
We have written codes for servers and the proxy server to implement the whole setup. Below is the description of code structures of server and proxy server.

7.1.1 Proxy Server:

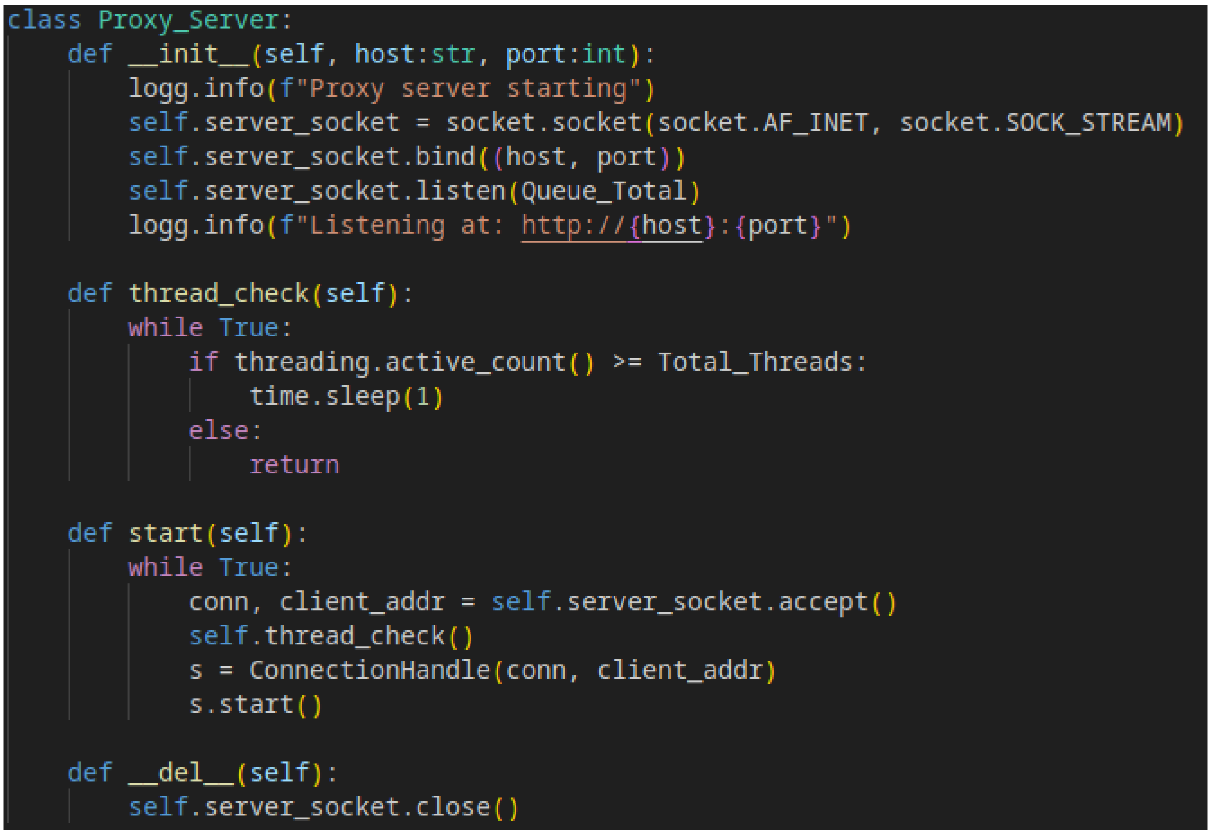
The proxy server is implemented combining various classes. These classes include TinyLFU Cache, StaticResponse, Error, Method, Request, Response, Protocol, ConnectionHandle, Proxy_Server, and other required functions.

- The main class Proxy_Server:

This class initializes the proxy server with a given host and port
Here, we are providing host to be localhost and port number to be 8000



```
if __name__ == '__main__':
    ser = Proxy_Server(host="localhost", port=8000)
    ser.start()
```

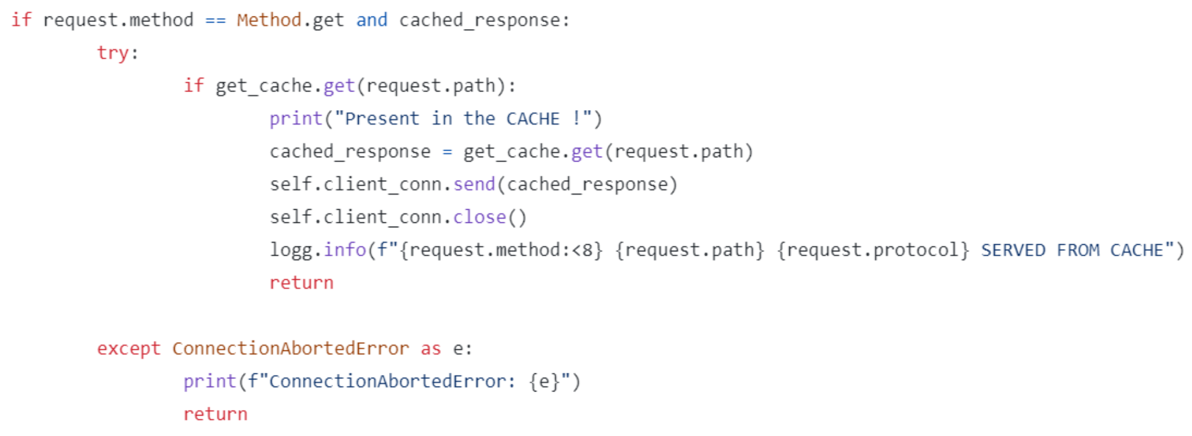


```python
class Proxy_Server:
    def __init__(self, host:str, port:int):
        logg.info(f"Proxy server starting")
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.bind((host, port))
        self.server_socket.listen(Queue_Total)
        logg.info(f"Listening at: http://{host}:{port}")

    def thread_check(self):
        while True:
            if threading.active_count() >= Total_Threads:
                time.sleep(1)
            else:
                return

    def start(self):
        while True:
            conn, client_addr = self.server_socket.accept()
            self.thread_check()
            s = ConnectionHandle(conn, client_addr)
            s.start()

    def __del__(self):
        self.server_socket.close()
```

Here, the init method creates a socket for the server and binds it to the specified host and port. The proxy server enters into an infinite loop in the start method where it accepts incoming connections using self.server_socket.accept(). It then sends this request to the ConnectionHandle Class which handles this request.

- ConnectionHandle Class:

The ConnectionHandle class can be run as a separate thread. It contains an initializing function and the run function which is the main executable. It performs X-Forwarded and language accepting.



```python
if request.method == Method.get and cached_response:
        try:
                if get_cache.get(request.path):
                        print("Present in the CACHE !")
                        cached_response = get_cache.get(request.path)
                        self.client_conn.send(cached_response)
                        self.client_conn.close()
                        logg.info(f"{request.method:<8} {request.path} {request.protocol} SERVED FROM CACHE")
                        return

        except ConnectionAbortedError as e:
                print(f"ConnectionAbortedError: {e}")
                return
```

Then, it checks if the HTTP request method is Get and if it is cached or not. This is a conditional block that specifically deals with caching responses for Get requests. It retrieves the cached response and then, it sends this cached response back to the client connection. It logs the event, indicating that the response was served from the cache.

```
if request.method == Method.head and cached_response:
        try:
                if head_cache.get(request.path):
                        print("Present in the CACHE !")
                        cached_response = head_cache.get(request.path)
                        self.client_conn.send(cached_response)
                        self.client_conn.close()
                        logg.info(f"{request.method:<8} {request.path} {request.protocol} SERVED FROM CACHE")
                        return

        except ConnectionAbortedError as e:
                print(f"ConnectionAbortedError: {e}")
                return

print("CACHE MISS !")
```

Similarly, this is applicable for the Head method as shown above.

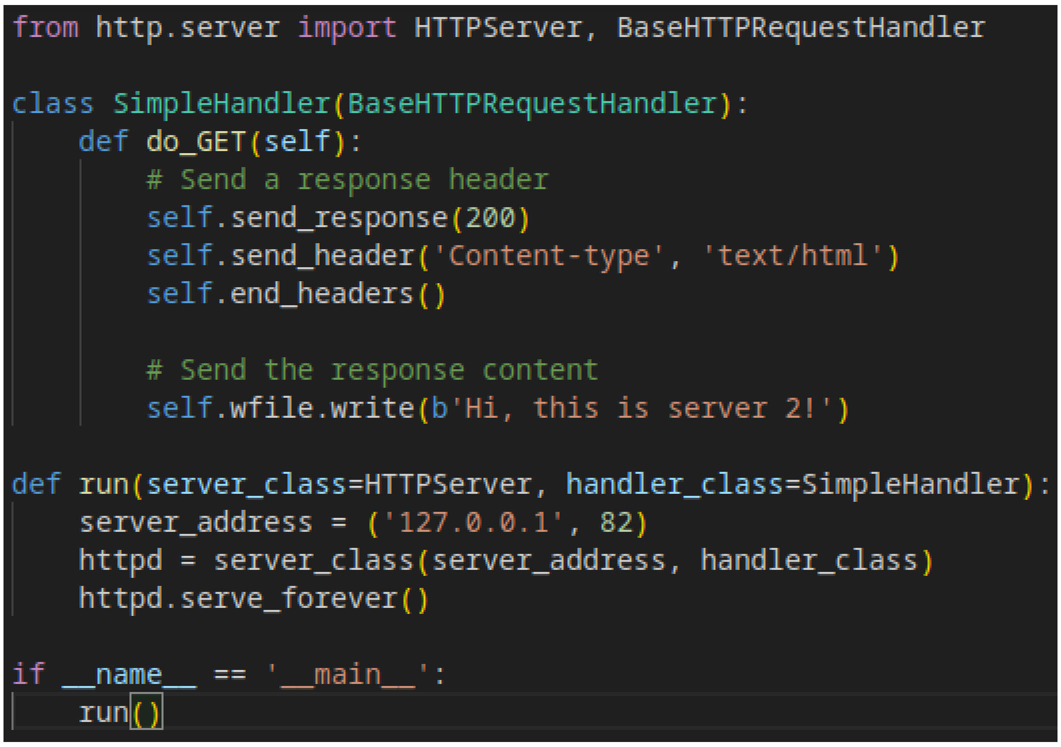
If the cache doesn't hit, we display that there is a cache miss on the terminal. Now, we need to send a request to the server and store it in a cache. Below is the code for requesting the server where the turn is 0 (Server 1) and we have to accept en language.

```
raw_request = change_path(raw_request, request.path, 'index.html',  function = "suffix")
if turn == 0:
        self.server_conn1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        try:
                self.server_conn1.connect(('127.0.0.1', 81))
        except:
                self.client_conn.send(Error.status_503.encode('utf-8'))
                self.client_conn.close()
                return
        turn = 1
        self.server_conn1.send(raw_request)
        self.server_conn1.settimeout(5)
        data = self.server_conn1.recv(max_size)
        self.server_conn1.close()
```

It modifies the raw_request by calling the change_path function, appending 'index.html' to the current path. The modification is done using the suffix function. It creates a new socket connection. This socket is intended for connecting to the backend server (server 1 in this case). If the connection to the backend server is successful, it sets turn to 1, indicating that the next request should be sent to a different backend server. It then sends the modified request to the backend server. There is also a timeout of 5 secs to receive data from the backend server. Atlast, it closes the connection to the server

7.1.2 Servers:

The code structure for the three servers server1, server2, server3 is as follows:

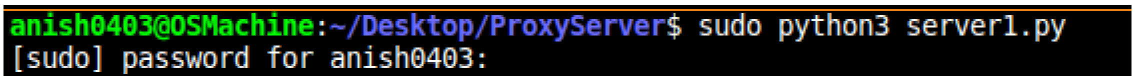

```python
from http.server import HTTPServer, BaseHTTPRequestHandler

class SimpleHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        # Send a response header
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

        # Send the response content
        self.wfile.write(b'Hi, this is server 2!')

def run(server_class=HTTPServer, handler_class=SimpleHandler):
    server_address = ('127.0.0.1', 82)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()

if __name__ == '__main__':
    run()
```

This sets up a basic http server using python's built-in "http.server" module. It defines the server address and port number, which in this case of server2, is (127.0.0.1 (localhost), 82). It also consists of a method for GET requests which is used to send the response header along with status code back to the client.

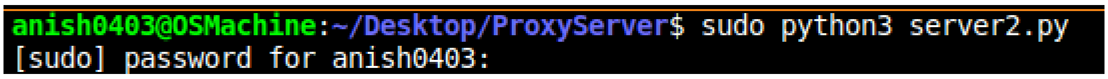
7.2 Steps to Run the Smart Proxy Server:

The steps to implement the whole system are as follows:

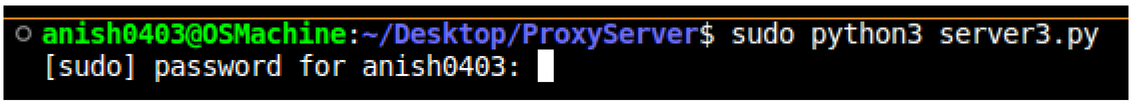
1. Start the server1 in root mode.


```
anish0403@OSMachine:~/Desktop/ProxyServer$ sudo python3 server1.py
[sudo] password for anish0403:
```

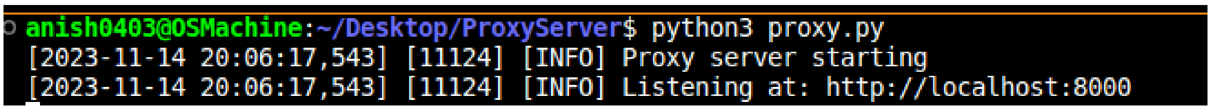
1. Start the server2 in root mode.


```
anish0403@OSMachine:~/Desktop/ProxyServer$ sudo python3 server2.py
[sudo] password for anish0403:
```

1. Start the server3 in root mode.


```
○ anish0403@OSMachine:~/Desktop/ProxyServer$ sudo python3 server3.py
  [sudo] password for anish0403:
```

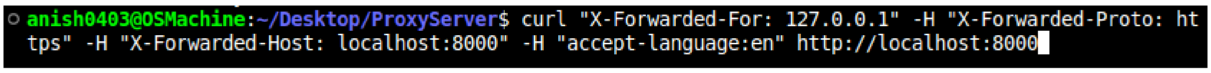
1. Run the Proxy Server code:



```
anish0403@OSMachine:~/Desktop/ProxyServer$ python3 proxy.py
[2023-11-14 20:06:17,543] [11124] [INFO] Proxy server starting
[2023-11-14 20:06:17,543] [11124] [INFO] Listening at: http://localhost:8000
```

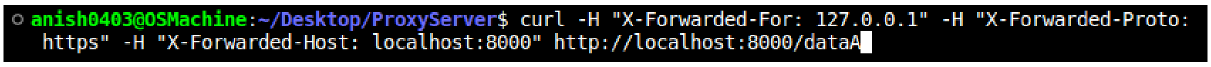
1. Open a new terminal. Send a request to the proxy server using curl. The **curl** command in Linux is used to download or upload data to a server using one of the supported protocols such as HTTP, FTP and so on. To send a header for a HTTP request we use the syntax: curl -H “Header Name : Input ".

After giving all the headers we would be giving the address of the site to be requested from the server. For example,



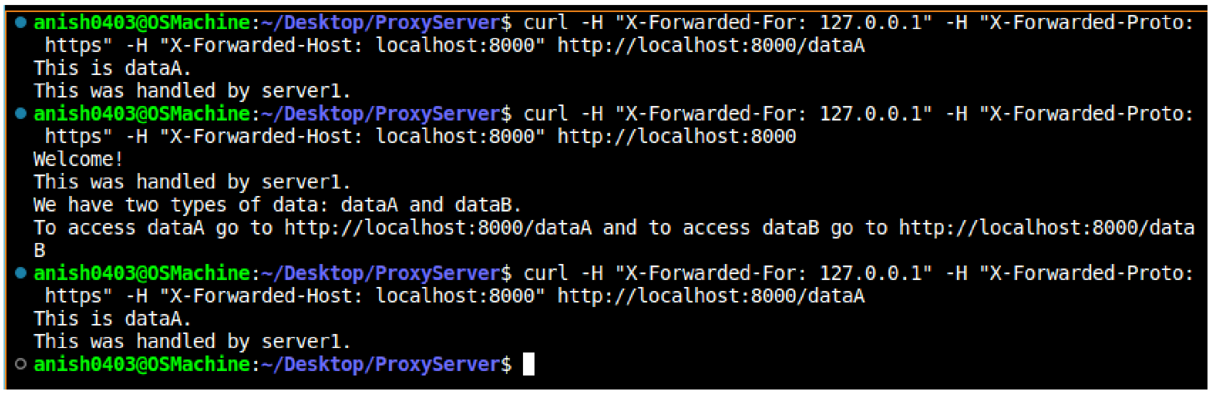
```
anish0403@OSMachine:~/Desktop/ProxyServer$ curl "X-Forwarded-For: 127.0.0.1" -H "X-Forwarded-Proto: ht
tps" -H "X-Forwarded-Host: localhost:8000" -H "accept-language:en" http://localhost:8000
```

The above url was to request for a subdomain, The user can also access the subdirectory (here named as dataA and dataB) by mentioning the path required in the way as specified below.



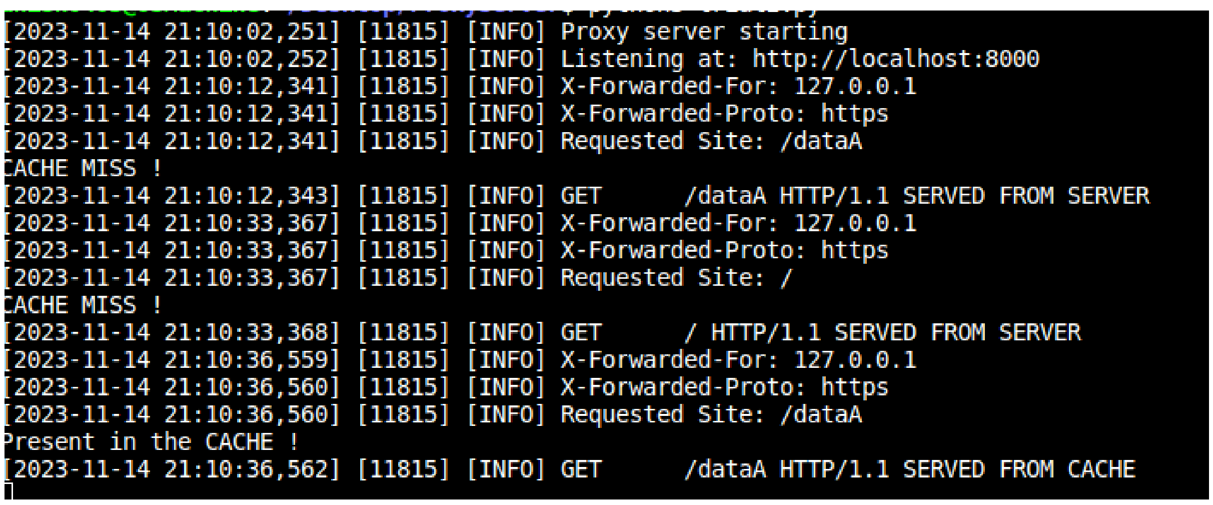
```
anish0403@OSMachine:~/Desktop/ProxyServer$ curl -H "X-Forwarded-For: 127.0.0.1" -H "X-Forwarded-Proto:
 https" -H "X-Forwarded-Host: localhost:8000" http://localhost:8000/dataA
```

1. After giving the curl command, the proxy server which listens to the request on port 8000, parses and transforms the headers accordingly, sends the request to the appropriate server and displays the needed output on the terminal where proxy is listening and the terminal from which we gave the curl commands. An example output is shown below.



```
anish0403@OSMachine:~/Desktop/ProxyServer$ curl -H "X-Forwarded-For: 127.0.0.1" -H "X-Forwarded-Proto:
 https" -H "X-Forwarded-Host: localhost:8000" http://localhost:8000/dataA
This is dataA.
This was handled by server1.
anish0403@OSMachine:~/Desktop/ProxyServer$ curl -H "X-Forwarded-For: 127.0.0.1" -H "X-Forwarded-Proto:
 https" -H "X-Forwarded-Host: localhost:8000" http://localhost:8000
Welcome!
This was handled by server1.
We have two types of data: dataA and dataB.
To access dataA go to http://localhost:8000/dataA and to access dataB go to http://localhost:8000/data
B
anish0403@OSMachine:~/Desktop/ProxyServer$ curl -H "X-Forwarded-For: 127.0.0.1" -H "X-Forwarded-Proto:
 https" -H "X-Forwarded-Host: localhost:8000" http://localhost:8000/dataA
This is dataA.
This was handled by server1.
anish0403@OSMachine:~/Desktop/ProxyServer$
```

Terminal from which we gave the curl commands



```
[2023-11-14 21:10:02,251] [11815] [INFO] Proxy server starting
[2023-11-14 21:10:02,252] [11815] [INFO] Listening at: http://localhost:8000
[2023-11-14 21:10:12,341] [11815] [INFO] X-Forwarded-For: 127.0.0.1
[2023-11-14 21:10:12,341] [11815] [INFO] X-Forwarded-Proto: https
[2023-11-14 21:10:12,341] [11815] [INFO] Requested Site: /dataA
CACHE MISS !
[2023-11-14 21:10:12,343] [11815] [INFO] GET      /dataA HTTP/1.1 SERVED FROM SERVER
[2023-11-14 21:10:33,367] [11815] [INFO] X-Forwarded-For: 127.0.0.1
[2023-11-14 21:10:33,367] [11815] [INFO] X-Forwarded-Proto: https
[2023-11-14 21:10:33,367] [11815] [INFO] Requested Site: /
CACHE MISS !
[2023-11-14 21:10:33,368] [11815] [INFO] GET      / HTTP/1.1 SERVED FROM SERVER
[2023-11-14 21:10:36,559] [11815] [INFO] X-Forwarded-For: 127.0.0.1
[2023-11-14 21:10:36,560] [11815] [INFO] X-Forwarded-Proto: https
[2023-11-14 21:10:36,560] [11815] [INFO] Requested Site: /dataA
Present in the CACHE !
[2023-11-14 21:10:36,562] [11815] [INFO] GET      /dataA HTTP/1.1 SERVED FROM CACHE
```

Terminal where proxy is listening

9. Summary

In this project, our team developed a Smart Proxy Server under Professor Sameer Kulkarni's guidance. Focusing on intelligent HTTP header manipulation, the server aims to optimize network traffic, incorporating advanced algorithms for parsing and manipulating headers, efficient caching mechanisms, load balancing strategies, and size compression. The project's key insights included HTTP header dynamics, dynamic manipulation algorithms, load balancing, and caching mechanisms.. We used Python as the language for implementing this smart proxy server. Features included smart HTTP request parsing, header manipulation (X-Forwarded-For, path transforms, and accept-language header handling), intelligent caching, dynamic load balancing, and size compression.

10. Takeaways

There were several key takeaways for our team throughout the course of this project:

- Effective Exploration: Tackling the challenge of building a proxy server without prior expertise demonstrated our team's ability to delve into complex topics and acquire both theoretical and practical knowledge on the fly.

- Structured Timeline Success: Adopting a weekly timeline strategy proved to be a motivating factor, ensuring consistent progress. Regular meetings with Professor Sameer Kulkarni and TA Dhyey Thummar played a pivotal role in steering our project in the right direction.

- Strategic Approach: Opting for a broad topic initially allowed us to explore various dimensions before honing in on specific aspects. This approach facilitated a more comprehensive understanding of the proxy server landscape.

11. Acknowledgements

We extend our sincere gratitude to Prof. Sameer Kulkarni for his insightful comments and valuable recommendations, which significantly contributed to improving the quality of the project. Special thanks to our Teaching Assistant Dhyey Thummar for his guidance and prompt resolution of our queries. Our heartfelt appreciation goes to teammates and friends who shared their valuable insights, supporting us throughout the project. Lastly, we appreciate the Department of Computer Science at IIT Gandhinagar for providing the opportunity and necessary resources for this endeavor.

12. References:

1. `https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/`

1. Luotonen, A. (1994). World Wide Web Proxies. W3C. `https://www.w3.org/History/1994/WWW/Proxies/`

   \

2. Wessels, D. (2004). Proxies. W3C.

   `https://www.w3.org/Daemon/User/Proxies/Proxies.html`

1. W3C. (2015). Using Proxy Servers. W3C. `https://www.w3.org/Library/User/Using/Proxy.html`

1. AVG. (2022). Smart DNS proxy server vs VPN. AVG Signal. `https://www.avg.com/en/signal/smart-dns-proxy-server-vs-vpn`

1. Nginx Inc. (2022). Introduction to reverse proxying. Nginx. `https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/#introduction`

1. Nginx Inc. (2022). Reverse proxy server. Nginx Resources. `https://www.nginx.com/resources/glossary/reverse-proxy-server/`

1. Nginx Inc. (2022). Basic HTTP features. Nginx.

   `https://nginx.org/en/#basic_http_features`

1. MDN Web Docs. Content Encoding

   `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Encoding`

1. https://blog.devgenius.io/optimizing-network-performance-a-guide-to-compression-headers-in-http-api-calls-f1e31e0e3057

1. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-For#security_and_privacy_concerns