

CSN-341

Project Report

Group-4

Group Details:

Name	Enrollment	Email Address	Contribution
1. Pranavdeep Singh	21119036	p_singh2@cs.iitr.ac.in	CDN Server Positioning and Prioritisation
2. Himani Panwar	21114041	h_panwar@cs.iitr.ac.in	CDN Server Positioning and Prioritisation
3. Piyush Arya	21114074	p_arya@cs.iitr.ac.in	IP Anycasting
4. Vardaan Dua	21115155	v_dua@cs.iitr.ac.in	IP Anycasting
5. Soham Singh	21114099	s_singh2@cs.iitr.ac.in	CDN Edge Server Caching
6. Rajat Raj Singh	21114079	r_rsingh@cs.iitr.ac.in	CDN Edge Server Caching
7. Priyansh Mawal	21114076	p_mawal@cs.iitr.ac.in	Prefetching
8. Atharv Chhabra	21118025	a_chhabra@cs.iitr.ac.in	Prefetching
9. Priyanshu Behera	21114077	p_behera@cs.iitr.ac.in	Load Balancing
10. Hrishit B P	21114042	h_bp@cs.iitr.ac.in	Load Balancing

Problem Statement: *Investigating the Optimization of Content Delivery Networks (CDNs) for Enhanced User Experience in Media Streaming and E-commerce*

Overview:

A content delivery network (CDN) is a geographically distributed group of servers that holds content close to end users. A CDN allows for the quick transfer of assets needed for loading Internet content, including HTML pages, JavaScript files, stylesheets, images, and videos. The popularity of CDN services continues to grow, and today the majority of web traffic is served through CDNs, including traffic from major sites like Facebook, Netflix, and Amazon. At its core, a CDN is a network of servers linked together with the goal of delivering content as quickly, cheaply, reliably, and securely as possible.

The aim of this project is to study different optimizations that can be applied at different fields of the Content Delivery Network, so that the end user can get the best service possible. In the subsequent sections, we have implemented and studied different optimizations that can be used to improve the performance of the CDNs with respect to user experience in media streaming and E-commerce.

Optimising Edge Server Positioning using Density Estimation

Introduction:

The placement of edge servers for a Content Delivery Network is a crucial aspect of optimizing performance and ensuring efficient content delivery. The goal is strategically positioning these servers to reduce latency, improve response times, and distribute content closer to end-users.

Due to an organization's financial and administrative constraints, the number of edge servers available for a CDN is limited. Hence, a thoughtful and data-driven approach is crucial for maximizing the effectiveness of a CDN.

Key optimisations enabled by strategic edge server placement:

- Reduced Latency
- Bandwidth Optimisations
- Improved Load Times
- Global Reach and Scalability

Approach:

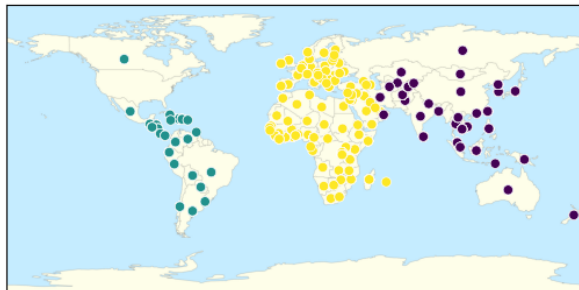
Used Gaussian Mixture Models to determine the optimal placement of edge servers given the dataset for access patterns around the world.

Gaussian Mixture Model is an unsupervised machine-learning algorithm whose roots are in Multivariate Normal Distributions. It enables tasks like Expectation-Maximisation, Clustering, etc. Using Gaussian Mixture Models with a tied and diagonal covariance matrix enables us to learn the best positions for a given number of edge servers for a given access pattern across the globe. We used world population data to estimate access patterns to demonstrate our project, assuming access rates are proportional to the population.

We have implemented the described technique in Python and generated plots for visualization purposes.

Plots:

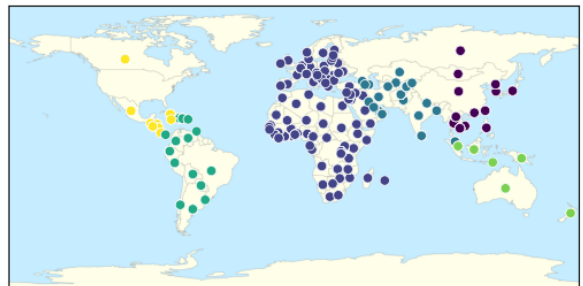
COUNTRY CLUSTERING CORRESPONDING TO 3 EDGE SERVERS



PROPOSED LOCATIONS FOR 3 EDGE SERVERS



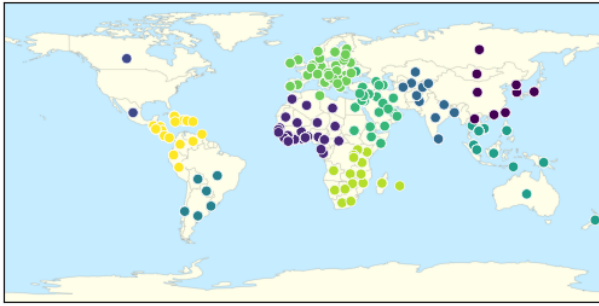
COUNTRY CLUSTERING CORRESPONDING TO 6 EDGE SERVERS



PROPOSED LOCATIONS FOR 6 EDGE SERVERS



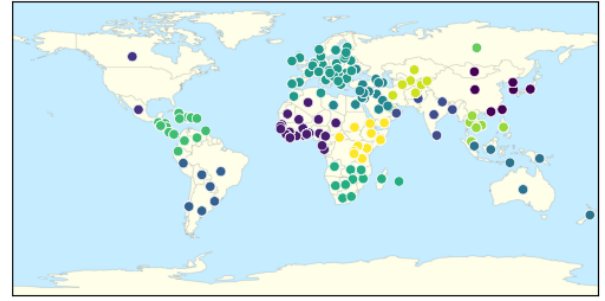
COUNTRY CLUSTERING CORRESPONDING TO 10 EDGE SERVERS



PROPOSED LOCATIONS FOR 10 EDGE SERVERS



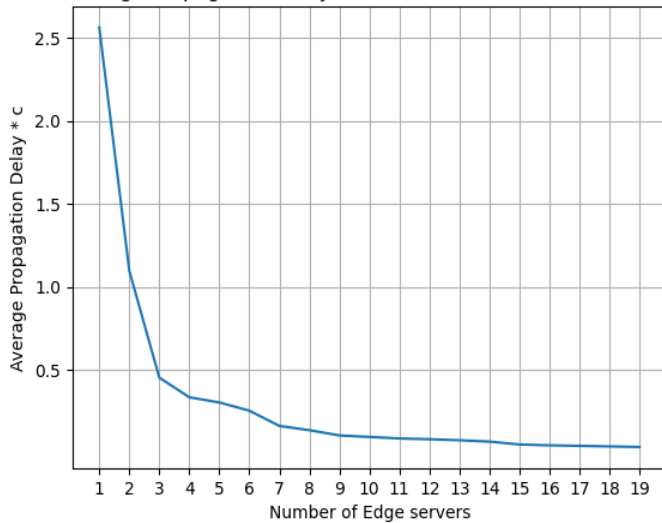
COUNTRY CLUSTERING CORRESPONDING TO 14 EDGE SERVERS



PROPOSED LOCATIONS FOR 14 EDGE SERVERS



Trend of Average Propagation delay from a receiver to the nearest edge server



The average propagation delays will naturally decrease as we increase the number of edge servers. But as we can notice in the graph given below, there exists an elbow point for the number of edge servers, i.e., after a certain number of edge servers, the marginal change in propagation delay with increasing the number of edge servers reduces drastically. This plot helps determine the optimal number of servers given the trade-off between the increase in costs due to multiple edge servers and the average response times for a user of the CDN.

The use of a modified GMM, where there exists a term in the loss function that penalizes large deviations in the number of requests serviced by each edge server, i.e., the variance of the number of requests serviced by each edge server, is ideal for the problem statement. By the use of this technique, we can ensure Load Balancing among edge servers, which is a crucial aspect of CDN Optimisations.

Priority-based request servicing:

We have designed and implemented a sophisticated priority scheduling mechanism for client requests within a Content Delivery Network (CDN). In addition to prioritization, we integrated caching at edge servers, facilitating faster access to frequently requested content. The architecture adopts a hierarchical edge server model, which enhances the scalability and efficiency of the CDN. To optimize network performance, a dual-protocol approach is implemented, utilizing UDP for request handling and TCP for reliable data transfer.

Entities:

The project encompasses three key entities running on different end systems - Clients, Edge Servers, and Main Servers. Clients initiate requests for content, Edge Servers prioritize and process these requests with the help of their cache, and Main Servers store content and pass it to edge servers when requested. Used socket programming in C++ to enable communication among these entities as described.

Priority Parameters:

- User Priority - The project introduces a user-centric priority system, ensuring that clients with higher priority are served with greater urgency. This prioritization is crucial in scenarios where certain users or applications require expedited access to content.
- File Priority - Similar to user priority, the project assigns priorities to files based on their significance. Critical files are given higher priorities, leading to quicker retrieval and delivery.
- Edge Server Priority (Hierarchical Architecture) - In cases where a hierarchical edge server architecture is employed, different edge servers are assigned priorities. This not only aids in load balancing but also ensures efficient distribution of client requests among the various edge servers.

Implementation:

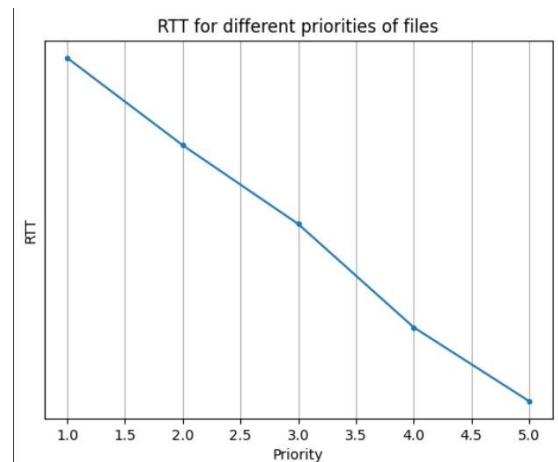
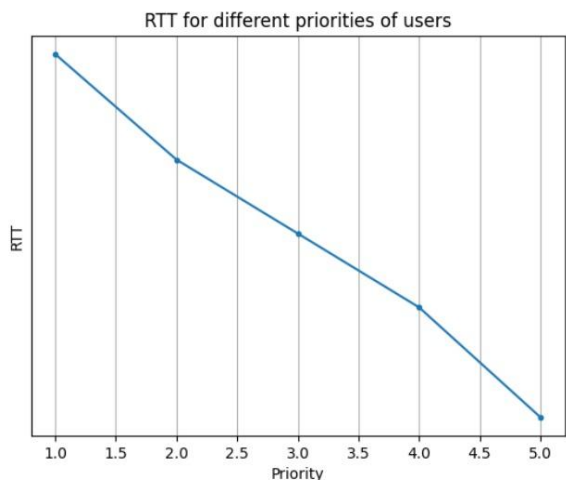
- Used C++ to implement the project.
- UDP for Request Handling - The project utilizes the lightweight UDP protocol for client requests. This decision is driven by the desire to minimize connection overhead and reduce latency, especially for time-sensitive applications.
- TCP for Reliable Data Transfer - For the actual data transfer process, TCP is chosen to ensure reliable, ordered, and error-checked delivery of content to clients. This is crucial for maintaining data integrity in a CDN environment.
- In-built Reliability Mechanism - The C++ code includes an in-built reliability mechanism to address UDP packet loss. In the event of packet loss, the application automatically re-sends requests after a timeout period, bolstering the system's resilience to network uncertainties.
- Used fork() for allowing concurrency in request servicing at the Main Server in case of multiple requests from various edge servers.

- The main server always listens at a port for TCP requests. Upon the establishment of a connection, the main server creates a child process and services the request. Once the request is serviced, the child process exits
- The edge server always has a UDP port open for receiving requests from the client, and once the service queue is full, uses a TCP port for servicing the requests in the order of priority.
- Used a priority queue to maintain the order between the requests, with a custom comparator which takes into account different aspects like file and user priority.

Process Flow:

- A client sends a request to the edge server using a UDP Packet
- This request is received and stored at the edge server.
- The edge server maintains a data structure storing the outstanding requests i.e., which are yet to be served in the order of their priority which is calculated on the basis of different metrics as listed above
- After a particular number of requests accumulate at the edge server or a certain amount of time elapses after servicing the previous set of requests, the edge server starts to service these requests
- In case the content requested is not present at the edge server, the edge server requests it from the main server via a TCP connection and stores it in the edge server's cache to avoid going back to the server in case another user requests the same content/file.
- The edge server now services the request via a TCP connection to ensure reliable data transfer

Plots:



These plots depict the effect of priority on the service time experienced by a user. For a user/file having a higher i.e., greater priority, the request is towards the start of the priority queue maintained by the server, leading to a shorter service time and hence a better user experience.

Window Mechanism:

To prevent the potential starvation of lower-priority entities, a window mechanism is implemented. This ensures that lower-priority user requests or less critical files are not indefinitely delayed. Instead, they are processed within a defined window of time or number of requests, striking a balance between high-priority and lower-priority requests.

Conclusion:

In conclusion, this project successfully implements a comprehensive priority scheduling system within a CDN network. By incorporating caching, adopting a dual-protocol approach with UDP and TCP, and introducing user, file, and edge server prioritization, the project significantly enhances the overall performance, responsiveness, and efficiency of the CDN. The in-built reliability mechanism and the window mechanism contribute to a well-balanced and adaptive CDN system capable of catering to diverse user and content priorities while avoiding indefinite delays for lower priority requests. The findings from this project hold potential for practical applications in optimizing CDN networks for various scenarios and user requirements.

Efficient routing to nearest Edge servers using IP-Anycasting:

We have worked on Investigating the optimisations of Content Delivery Networks for Enhanced User Experience in Media Streaming and E-Commerce through the well known technique of IP Any-Casting . We have simulated the IP Any-Casting Technique using C++ and have incorporated the following in our simulation :

1. Open Shortest Path First for INTRA-AS Routing
2. Border Gateway Protocol for INTER-AS Routing
3. Prefix Matching
4. IP Any Casting by keeping the IP Addresses for edge servers(content carrying servers) same

Introduction-

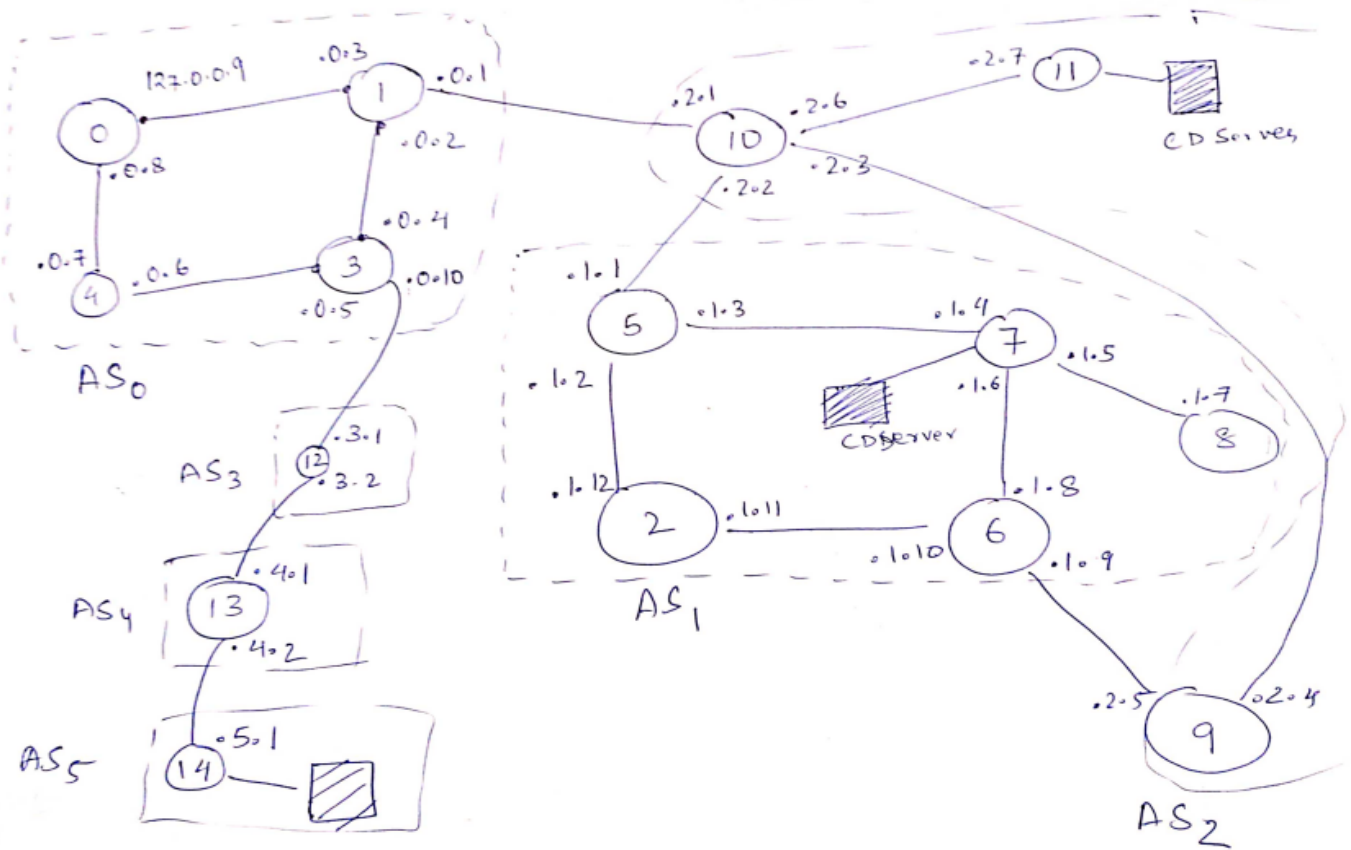
IP-Anycast is a networking technique that allows multiple servers or network devices to share the same IP address, effectively creating a distributed service. It enables organizations to enhance the availability, redundancy, and scalability of their services by directing traffic to the nearest(or best-performing) server within a group of geographically dispersed nodes. Below is our simulation of IP-Anycast, and the results of different queries made on it to demonstrate that IP-Anycast is indeed a better approach to routing packets among different autonomous systems.

Optimisations Achieved using IP Any-Casting Technique-

- **Proximity-Based Routing** : Anycasting enables routing to the nearest server in terms of network topology. When a user makes a request, the anycast system directs the traffic to the closest available server. This reduces the number of network hops and minimizes latency, resulting in faster response times.
- **Low Latency and response time** : Anycasting enhances fault tolerance and high availability by allowing traffic to be redirected to the next available server in the anycast group in the event of server failures or network issues. This improves the overall reliability of the CDN.
- **DDOS Mitigation using Distribution of Traffic** : Anycasting allows the same IP address to be assigned to multiple servers across different locations. In the case of a DDoS attack, the attack traffic is distributed across multiple servers, helping to distribute the load and prevent overwhelming a single server.
- **Scalability** : Anycasting supports the seamless addition of new servers to the anycast group. This makes it easier to scale the CDN infrastructure as traffic demands increase.

SIMULATION DETAILS :

Input network in the code-



Calculated Next Hop Info-

```

st_nexthop.txt
Asst next hop Info for 0
Dest asn- 2 NextHop- {edge router,direct interface} : {1 - 127.0.0.1, 10 - 127.0.2.1}
Dest asn- 3 NextHop- {edge router,direct interface} : {3 - 127.0.0.10, 12 - 127.0.3.1}

Asst next hop Info for 1
Dest asn- 2 NextHop- {edge router,direct interface} : {1 - 127.0.0.1, 10 - 127.0.2.1}
Dest asn- 3 NextHop- {edge router,direct interface} : {3 - 127.0.0.10, 12 - 127.0.3.1}

Asst next hop Info for 2
Dest asn- 2 NextHop- {edge router,direct interface} : {5 - 127.0.1.1, 10 - 127.0.2.2}

Asst next hop Info for 3
Dest asn- 2 NextHop- {edge router,direct interface} : {1 - 127.0.0.1, 10 - 127.0.2.1}
Dest asn- 3 NextHop- {edge router,direct interface} : {3 - 127.0.0.10, 12 - 127.0.3.1}

Asst next hop Info for 4
Dest asn- 2 NextHop- {edge router,direct interface} : {1 - 127.0.0.1, 10 - 127.0.2.1}
Dest asn- 3 NextHop- {edge router,direct interface} : {3 - 127.0.0.10, 12 - 127.0.3.1}

Asst next hop Info for 5
Dest asn- 2 NextHop- {edge router,direct interface} : {5 - 127.0.1.1, 10 - 127.0.2.2}

Asst next hop Info for 6
Dest asn- 2 NextHop- {edge router,direct interface} : {6 - 127.0.1.9, 9 - 127.0.2.5}

Asst next hop Info for 7
Dest asn- 2 NextHop- {edge router,direct interface} : {5 - 127.0.1.1, 10 - 127.0.2.2}

Asst next hop Info for 8
Dest asn- 2 NextHop- {edge router,direct interface} : {5 - 127.0.1.1, 10 - 127.0.2.2}

Asst next hop Info for 9
Dest asn- 0 NextHop- {edge router,direct interface} : {10 - 127.0.2.1, 1 - 127.0.0.1}
Dest asn- 1 NextHop- {edge router,direct interface} : {9 - 127.0.2.5, 6 - 127.0.1.9}
  
```


Link State Info-

```
anycast.cpp  ls.txt  X
ls.txt
1  Link State Info for 0
2  0 4
3  0 1
4  1 0
5  1 3
6  3 1
7  3 4
8  4 0
9  4 3
10
11 Link State Info for 1
12 0 4
13 0 1
14 1 0
15 1 3
16 3 1
17 3 4
18 4 0
19 4 3
20
21 Link State Info for 2
22 2 5
23 2 6
24 5 2
25 5 7
26 6 2
27 6 7
28 7 5
29 7 6
30 7 8
31 8 7
32
33 Link State Info for 3
34 0 4
```

Routing Table-

```
anycast.cpp  routing_info.txt  X
routing_info.txt
1  Routing Table Info for 0
2  Dest- 0 Nexthop- 0 ip interface of current router- ip interface of next hop router-
3  Dest- 1 Nexthop- 1 ip interface of current router- 127.0.0.9 ip interface of next hop router- 127.0.0.3
4  Dest- 3 Nexthop- 1 ip interface of current router- 127.0.0.9 ip interface of next hop router- 127.0.0.3
5  Dest- 4 Nexthop- 4 ip interface of current router- 127.0.0.8 ip interface of next hop router- 127.0.0.7
6
7  Routing Table Info for 1
8  Dest- 0 Nexthop- 0 ip interface of current router- 127.0.0.3 ip interface of next hop router- 127.0.0.9
9  Dest- 1 Nexthop- 1 ip interface of current router- ip interface of next hop router-
10 Dest- 3 Nexthop- 3 ip interface of current router- 127.0.0.2 ip interface of next hop router- 127.0.0.4
11 Dest- 4 Nexthop- 0 ip interface of current router- 127.0.0.3 ip interface of next hop router- 127.0.0.9
12
13 Routing Table Info for 2
14 Dest- 2 Nexthop- 2 ip interface of current router- ip interface of next hop router-
15 Dest- 5 Nexthop- 5 ip interface of current router- 127.0.1.12 ip interface of next hop router- 127.0.1.2
16 Dest- 6 Nexthop- 6 ip interface of current router- 127.0.1.11 ip interface of next hop router- 127.0.0.10
17 Dest- 7 Nexthop- 5 ip interface of current router- 127.0.1.12 ip interface of next hop router- 127.0.1.2
18 Dest- 8 Nexthop- 5 ip interface of current router- 127.0.1.12 ip interface of next hop router- 127.0.1.2
19
20 Routing Table Info for 3
21 Dest- 0 Nexthop- 1 ip interface of current router- 127.0.0.4 ip interface of next hop router- 127.0.0.2
22 Dest- 1 Nexthop- 0 ip interface of current router- 127.0.0.4 ip interface of next hop router- 127.0.0.2
23 Dest- 3 Nexthop- 3 ip interface of current router- ip interface of next hop router-
24 Dest- 4 Nexthop- 4 ip interface of current router- 127.0.0.5 ip interface of next hop router- 127.0.0.6
25
26 Routing Table Info for 4
27 Dest- 0 Nexthop- 0 ip interface of current router- 127.0.0.7 ip interface of next hop router- 127.0.0.9
28 Dest- 1 Nexthop- 0 ip interface of current router- 127.0.0.7 ip interface of next hop router- 127.0.0.9
29 Dest- 3 Nexthop- 3 ip interface of current router- 127.0.0.6 ip interface of next hop router- 127.0.0.5
30 Dest- 4 Nexthop- 4 ip interface of current router- ip interface of next hop router-
31
32 Routing Table Info for 5
33 Dest- 2 Nexthop- 2 ip interface of current router- 127.0.1.2 ip interface of next hop router- 127.0.1.12
34 Dest- 5 Nexthop- 5 ip interface of current router- ip interface of next hop router-
35 Dest- 6 Nexthop- 3 ip interface of current router- 127.0.1.2 ip interface of next hop router- 127.0.1.12
36 Dest- 7 Nexthop- 7 ip interface of current router- 127.0.1.3 ip interface of next hop router- 127.0.1.4
```

Testing on some queries -

1. From router 13 :

```
IP_ANYCAST\" ; if ($?) { g++ anycast.cpp -o anycast } ; if ($?) { .\anycast }
Enter Query
Enter id of current Node
13
Outputting the path to the nearest cdn server
AS PATH :: AS5 1.255.0
NEXT HOP
GOTO EDGE ROUTER 13
THEN GO FROM INTERFACE 127.0.4.2 TO INTERFACE 127.0.5.1
```

2. From router 12 :

```
PS C:\Users\hp\Desktop\NETWORKS_PROJ\IP_ANYCAST> cd "c:\Users\hp\Desktop\NETWORKS_PROJ\IP_ANYCAST\" ; if ($?) { g++ anycast.cpp -o anycast } ; if ($?) { .\anycast }
Enter Query
Enter id of current Node
12
Outputting the path to the nearest cdn server
AS PATH :: AS0 AS2 1.255.0
NEXT HOP
GOTO EDGE ROUTER 12
THEN GO FROM INTERFACE 127.0.3.1 TO INTERFACE 127.0.0.10
PS C:\Users\hp\Desktop\NETWORKS_PROJ\IP_ANYCAST> █
```

Caching

Caching in Content Delivery Networks (CDNs) is a crucial mechanism that involves storing copies of web content at various strategically located servers around the world. The primary goal of caching in CDNs is to reduce latency and improve the overall performance of websites or web applications for end users

How caching helps to optimise the Content Delivery Network?

Caching in CDNs means replicating content from origin servers to multiple edge servers located in different geographical locations. These edge servers are strategically placed to be closer to end users. By having copies of content closer to the end user, CDNs significantly reduce the time it takes for a request to reach the server and for the server to respond. This minimizes latency and improves the overall loading speed of web pages. Faster loading times and reduced latency contribute to an improved user experience. Users are more likely to stay on a website that loads quickly and responds promptly to their requests.

Types of caching:

- **Static Content Caching:** This involves caching static assets like images, stylesheets, and scripts that don't change frequently.
- **Dynamic Content Caching:** Some CDNs also cache dynamic content, such as personalized data or frequently accessed database queries. However, this requires more sophisticated cache management strategies.

In our investigation, we have only focused on static caching.

Implementation:

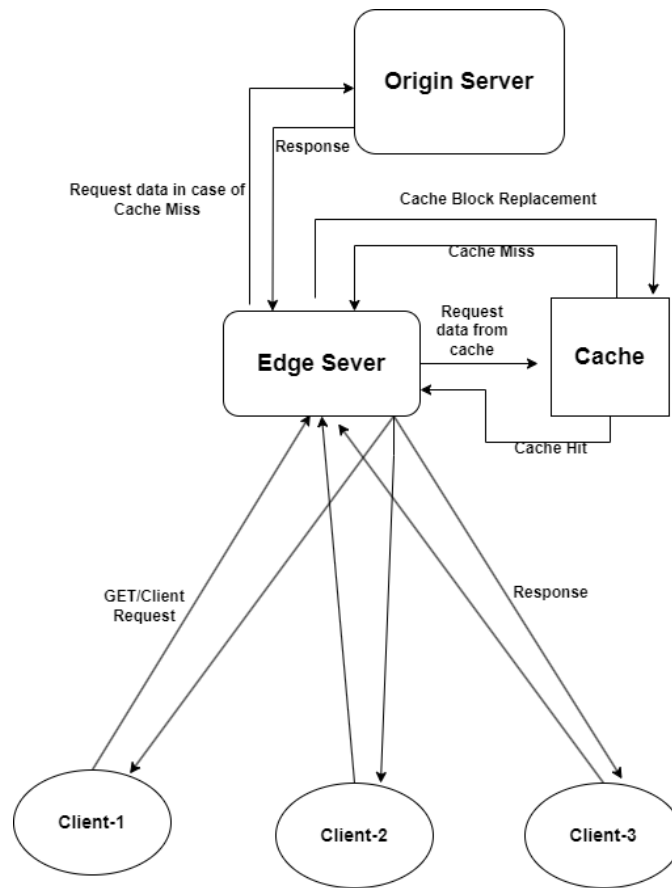
We simulated the working of CDN by making three different c++ programs- first one for the main CDN server, second for the edge server and the third for the client process. We used socket programming so that the three different programs are able to connect and send data to each other using TCP like in an actual Content Delivery Network.

The main server has all the files that the clients can request. Caching has been implemented on the edge server; it stores the recently requested files and serves them whenever the client asks for them.

Thus, the flow of the request is as follows:

1. Whenever the client wants a file, it establishes a TCP connection to the edge server and makes the request.
2. The edge server receives the request and checks if the requested file is in the cache or not.
3. If it is available in the cache, the edge server sends the file to the client without the intervention of the main server.
4. If the requested file isn't available in the cache the edge server connects to the main server using TCP and asks for the required file.

5. The main server receives the request and sends all the objects of the file(web page) to the edge server.
6. After receiving the file from the main server, the edge server replies back to the client with the requested file, and also caches it so that any further request for the same file objects by same or other client can be serviced without the intervention of the main server.



Caching Techniques used:

We have implemented a 4-way set associative cache and compared three different types of cache replacement policies to determine which is the best type of cache for different type of file request patterns. Cache replacement policies are used to determine the best cache block to replace when the cache is full and we have to insert a recently requested block in to the cache.

We implemented the three most used cache replacement techniques:

1. First In First Out (FIFO): The block which was inserted first is replaced.
2. Least Recently Used (LRU): the block which is the least recently used is replaced.
3. Least Frequently Used (LFU): the block which is the least frequently used is replaced.

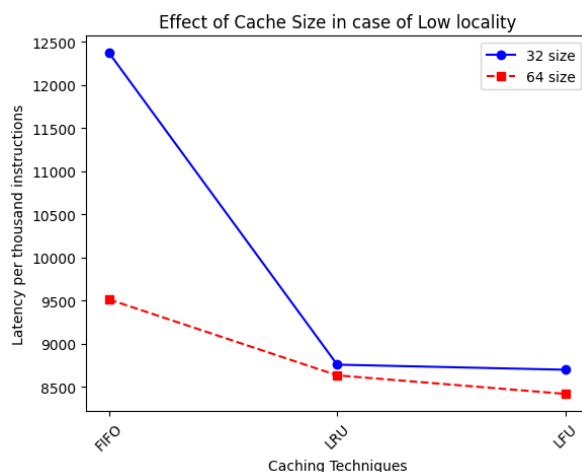
Another important thing that is widely used in CDN caching and which we have also implemented is using Time To Live field with each cache block.

When websites respond to CDN servers with the requested content, they attach the content's TTL as well, letting the servers know how long to store it. The TTL is stored in a part of the response called the HTTP header, and it specifies for how many seconds, minutes, or hours content will be cached. When the TTL expires, the cache removes the content. Some CDNs will also purge files from the cache early if the content is not requested for a while, or if a CDN customer manually purges certain content.

We studied the performance of our CDN cache system on two different types of file request patterns- first one is the low locality request pattern, in which the file objects requested by different client/clients are very less related to each other i.e. one client is requesting an object of one web page and another client is requesting some other object related to some other web page. The second one is the high locality request pattern where clients are mostly requesting objects of the same web page.

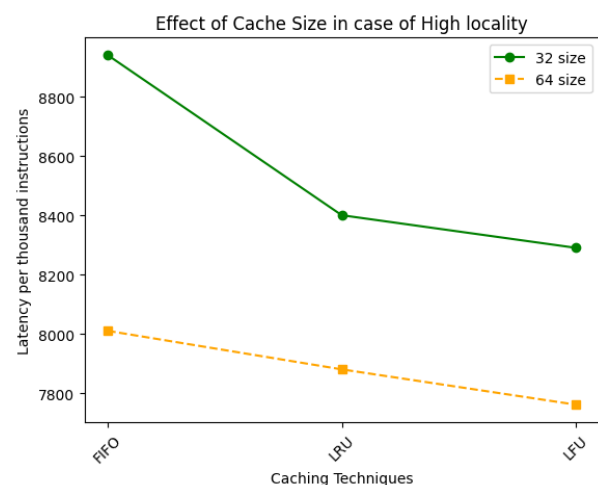
The results of our study are summarized below.

Results:

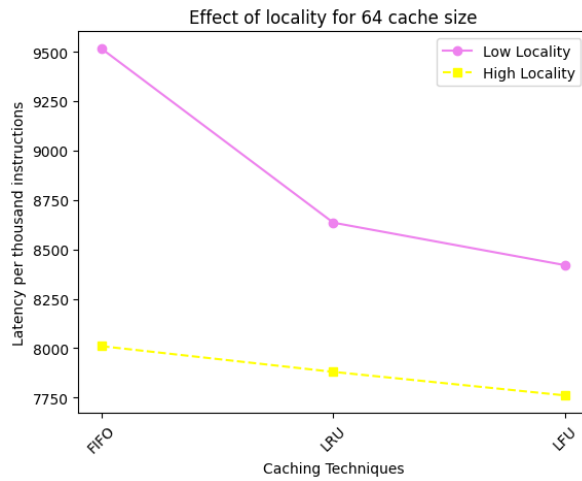


We have compared the effect of cache size in a low locality scenario. We can observe that in either of the cache sizes, LRU performs better than FIFO which in turn has better throughput than FIFO caching technique as expected. Cache size refers to the amount of memory allocated for caching data to improve performance. Increasing cache size can reduce the number of times data has to be retrieved from disk, which in turn can reduce latency. However, there is an optimal cache size at which the benefits of caching start to decline.

Next we have analyzed the effect of Cache size in high locality scenarios on different caching techniques. The result we got complies with the general trend. If the cache size is too small, the number of cache misses increases, negatively impacting performance. If the cache size is too large, it consumes more memory, reducing the available memory for other processes. The overall impact of cache size on latency depends on various factors such as the size of the data set, the workload, and the specific



implementation.



We have also researched the effect of locality on different caching techniques in optimal cache size scenarios. We came to the conclusion that high locality can result in faster content delivery to the end users as they are majorly requesting objects of the same web pages, as compared to low locality request pattern where clients majorly request objects of different web pages. To minimize cache delay, it is important to consider cache locality when designing algorithms and data structures.

Prefetching

What is Prefetching?

Prefetching is a technique used in computer systems, particularly in the context of memory and storage, to improve performance by anticipating and loading data into the cache or main memory before it is requested. The goal is to reduce the latency associated with accessing data, making it available more quickly when requested by the Clients from the CDN servers.

There are different types of prefetching, and they are commonly applied in various computing scenarios:

Memory Prefetching:

- Hardware Prefetching
- Software Prefetching

Storage Prefetching:

- Disk Prefetching
- Web Browsing
- Database Prefetching

Prefetching aims to hide the latency of fetching data from relatively slower storage or memory by overlapping the data retrieval with other computations. If predictions are accurate, prefetching can significantly improve overall system performance. However, if predictions are inaccurate, it may lead to wasted resources and negatively impact performance.

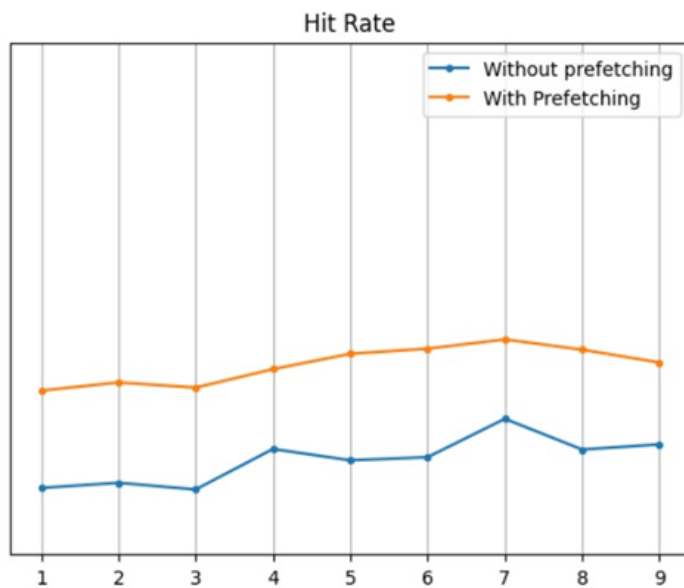
Approach:

We focused on analyzing how prefetching between the Main CDN Server and the Edge CDN Servers can help in improving the latency, load times and hit rates of the Edge Server Caches. We simulated the overall CDN architecture using Mainly 3 C++ programs, the Main Server, the Edge Servers and the Clients, each of which run on different computers. A connection is established between the Clients and the Edge Servers using TCP, and the same is true for a connection between an Edge Server and the Main Server. A client requests some specified data from the Edge Server, which then replies to the client with the appropriate response. If the data is present in the cache of the Edge server, it responds with the same, otherwise it makes a request to the Main server and fills its cache with the same, for the purpose of replying to future requests.

The idea of prefetching comes in the response of the Main server to the Edge server. When an Edge server requests a given set of data from the Main server, the Main server not only provides that data to

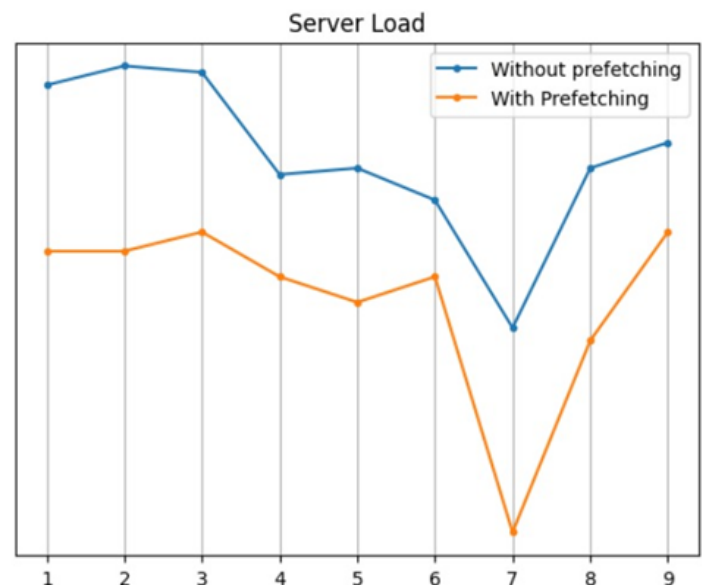
the Edge server, but also piggybacks some additional related data that it anticipates the client might request in the near future. This helps amortize the latency costs over a longer period and thus, reduces the response times for the client by a big margin. For the prefetching algorithm, we avoided an expensive ML model as our purpose was not to analyze a specific prefetching algorithm, but the optimizations that the idea of prefetching can potentially result in. We implemented a simple prefetching algorithm, whereby the client is assumed to make requests of integers within a difference of 5 from each other. When the Main server replies to a request from the Edge server, it supplies 2 additional integers to the Edge server in addition to the one currently requested. These 2 integers are in the range of about 5 integers from the requested integer. Once the Edge server gets a response from the Main server, it would update its cache with the new data and send a response back to the client.

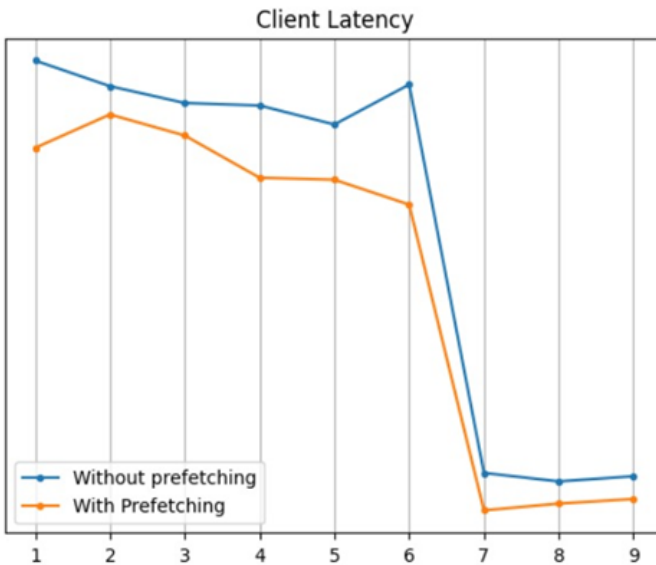
Results:



We can observe an increase in the hit rates for the caches of the Edge servers when we make use of prefetching. This is an obvious implication of the fact that the prefetching algorithm made correct anticipations of the data that the client would request in the near future. Consequently, the data that the Edge server prefetched from the Main server was updated in the cache and resulted in a higher hit rate in comparison to the no prefetching case.

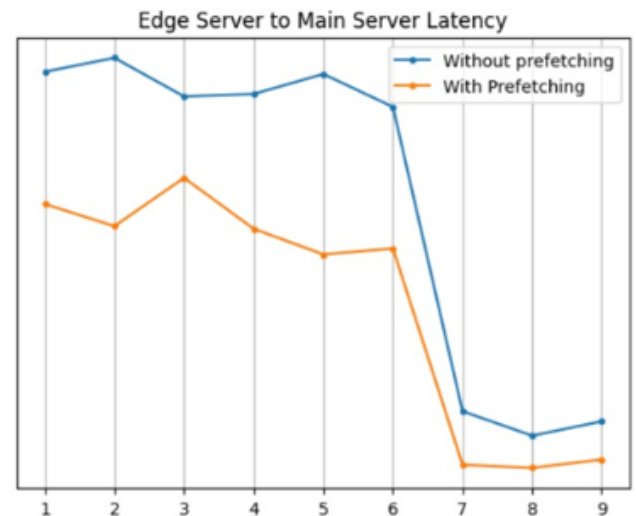
To measure the load on the Main server, we used the metric as the number of requests served by the Main server in a given interval of time. The higher the number of requests in an interval, the higher the load on the Main server. We can clearly observe that when we make use of prefetching, the load on the Main server reduces considerably. This is because the potential related data that the Edge server would request in the near future is actually already sent due to prefetching, reducing the number of requests that the Edge server makes from the Main server, consequently reducing the load.





Another major advantage of prefetching is lower client latencies. Consider as an example the case where a client makes three requests. Instead of the Edge server making 3 requests from the Main server and then replying to the client, the Edge server would cache the prefetched items in its cache and then reply to the client with the same. Thus, the client doesn't need to wait for the Edge server to retrieve the items from the Main server over and over again, and just gets its responses from the Edge server cache, resulting in improved response times.

Another aspect that we observed improvements in was that of the Edge server to Main server latency. This observed an improvement due to the fact that instead of making requests over and over again for each data item, the Edge server receives more data in a given amount of time due to prefetching, which saves future Round Trip Times to the Main server. Note that even though a reduction is observed in latencies, we'd need greater bandwidth in order to support sending more data in the response of a request from the Edge server. So it's really a give and take argument.



Load Balancing

Description:

In the rapidly evolving landscape of online media streaming and e-commerce, delivering a seamless user experience is crucial for the success of digital platforms. This project focuses on investigating and optimizing Content Delivery Networks (CDNs) with a specific emphasis on load balancing to enhance user experience. The objective is to ensure efficient content distribution, reduce latency, and improve overall performance.

Introduction:

CDNs provide the content to the users through a range of geographically distributed edge servers which are nearer to the users. We have taken into account the following constraints defined for a edge server which are

- 1) Max number of connections
- 2) Distance of edge server from the client (Euclidean distance b/w edge server and client)
- 3) Propagation Speed of the signal

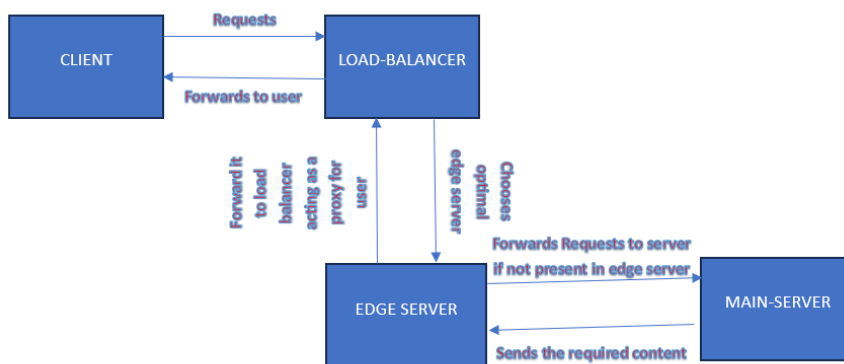
Note: We have assumed all the servers and the clients to have the same Transmitting Rate.

Implementation Details:

We have used pthreads for establishing separate connections one between load balancer and the client and the other between load balancer and the main server. The meta data is fetched from all the three edge servers at the very beginning.

We have 3 files named as ed_test(num) for different edge servers of the CDN network, and a client file. The required connections use TCP sockets.

Working Flow & Approach:



The client requests a particular content from the load balancer which is then diverted by the load-balancer to the most optimal edge server on the basis of costs assigned to each of the parameters and then the requests goes back to the client in a similar fashion.

Algorithm:

It is implemented in the load balancer file in which we have associated cost with each of the 3 parameters discussed above. It is calculated for each thread or each connection.

```
int choose_server(pair<int,int> client_coords){
    double w1,w2;
    int server = 0;
    double value = 1e10;
    int x = client_coords.first;
    int y = client_coords.second;
    for(int i = 0;i<NUM_EDGE_SERVERS;i++){
        double dist = pow(abs(x-server_metadata[i].coordinates.first),2)+pow(abs(y-server_metadata[i].coordinates.second),2);
        dist = sqrt(dist);
        double prop_delay = dist/server_metadata[i].signal_speed;
        double server_load = ((double)num_connections[i])/server_metadata[i].max_no_of_connections;
        double penalty = w1*prop_delay+w2*server_load;
        if(penalty<value){
            server = i;
            value = penalty;
        }
    }
    return server;
}
```

Outputs (Explained):

A) Started load balancer, it gets metadata from the edge servers, here it receives 10, 12, and 14 which are transmission speed of the medium(chosen randomly here for simulation)

```
priyanshu@DESKTOP-G7GII8N:/mnt/e/Priyanshu's CSN-341_project$ ./l1.out
Sent
Received 10
Sent
Received 12
Sent
Received 14
Connection Port: 8080
█
```

B) Created first client, we pass x and y coordinates, then we accept a number as input to establish the connection. This entering of numbers can be considered analogous to opening of the CDN application, let's assume when we open an application it randomly sends some number to the server.

```
priyanshu@DESKTOP-G7GII8N:/mnt/e/Priyanshu's CSN-341_project$ ./c1.out 2 3
2
3
Enter number
4
Data sent.
1
█
```

- C) In the same step we can see that the load-balancer has received coordinates from the client and has sent an acknowledgement.

```
priyanshu@DESKTOP-G7GII8N:/mnt/e/Priyanshu's CSN-341_project$ ./l1.out
Sent
Received 10
Sent
Received 12
Sent
Received 14
Connection Port: 8080
2 3
sent ack
█
```

- D) In the same step, we can see the output of edge server3, it shows that a new thread has been created, because the client has connected here.

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL	PORTS
hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation\$./e1.out	hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation\$./e2.out	hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation\$./e3.out	hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation\$./e1.out	hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation\$./e2.out
Connection Port: 9090	Connection Port: 9091	Connection Port: 9092	Connection Port: 9090	Connection Port: 9091
New thread created	New thread created	New thread created	New thread created	New thread created
Sent metadata	Sent metadata	Sent metadata	Sent metadata	Sent metadata
█	█	█	█	█

- E) Data requested from client

```

priyanshu@DESKTOP-G7GII8N:/mnt/e/Priyanshu's CSN-341_project$ ./c1.out 2 3
2
3
Enter number
4
Data sent.
1
Web_series1
author1
1

```

F) Load Balancer taking the input from client

```

priyanshu@DESKTOP-G7GII8N:/mnt/e/Priyanshu's CSN-341_project$ ./l1.out
Sent
Received 10
Sent
Received 12
Sent
Received 14
Connection Port: 8080
2 3
sent ack
Received data: Web_series1
11
author1
Acknowledgement received from client
sent ack

```

G) Main server receiving the request

```

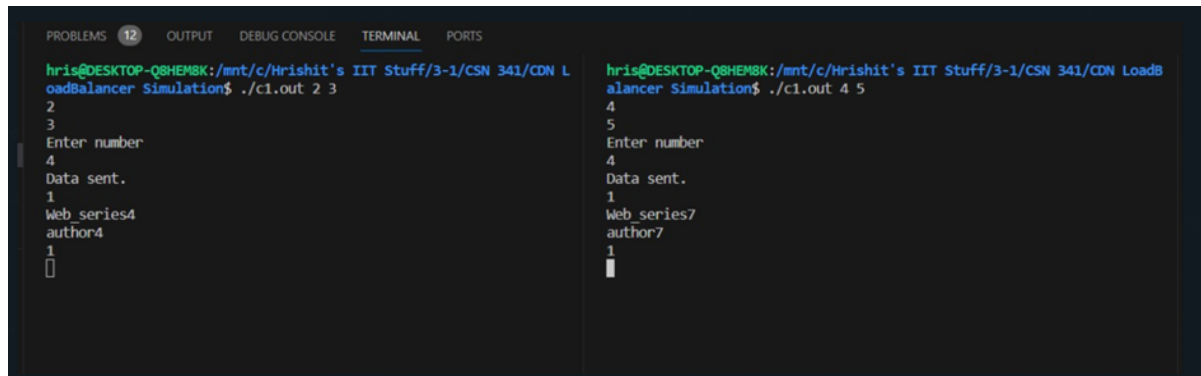
priyanshu@DESKTOP-G7GII8N:/mnt/e/Priyanshu's CSN-341_project$ ./m1.out
Connection Port: 8081
New thread created
11
Received data: Web_series1

```

Optimization Achieved:

Here we will see another request from some other client will be diverted to some other edge server.

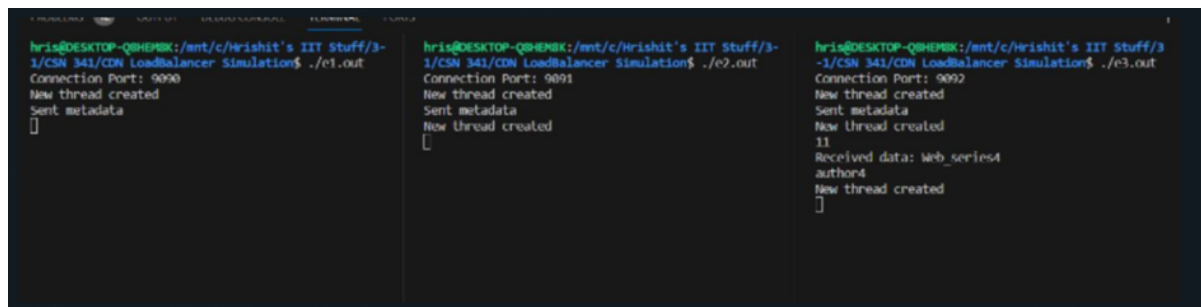
A) A new request from a client with different coordinates is being made.



```
hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation$ ./c1.out 2 3
2
3
Enter number
4
Data sent.
1
Web_series4
author4
1
[]

hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation$ ./c1.out 4 5
4
5
Enter number
4
Data sent.
1
Web_series7
author7
1
[]
```

B) This time the thread was created in a different edge server for connections keeping in mind the number of connections made and the propagation speed(here edge server 2 was having the 2nd highest speed).



```
hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation$ ./c1.out
Connection Port: 9000
New thread created
Sent metadata
[]

hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation$ ./c2.out
Connection Port: 9001
New thread created
Sent metadata
New thread created
[]

hris@DESKTOP-Q8HEM8K:/mnt/c/Hrishit's IIT Stuff/3-1/CSN 341/CDN LoadBalancer Simulation$ ./c3.out
Connection Port: 9002
New thread created
Sent metadata
New thread created
11
Received data: Web_series4
author4
New thread created
[]
```