DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment: 1

Student Name: Ashmit Agrawal **UID:** 23BCS11854
Branch: B.E.- C.S.E. **Section/Group:** 23BCS_KRG-1A
Semester: 6 **Date of Performance:** 10-01-26
Subject Name: System Design **Subject Code:** 23CSH-314

1. **Aim:** To design, develop, and analyze a basic URL shortening system that transforms long URLs into compact short URLs, and to evaluate its performance in terms of latency and throughput under different traffic loads.

2. Objective:

1. To gain an understanding of how a URL shortening service operates internally.
2. To build a working URL shortening and redirection mechanism.
3. To evaluate system latency and throughput under simulated concurrent user requests.
4. To analyze how architectural and design decisions affect non-functional performance metrics.

Requirements (Tools):

- Postman – for testing API endpoints
- Diagramming tools: draw.io, Excalidraw, Lucidchart
- Python libraries and frameworks (e.g., Flask)

3. Procedure and Output:

1. **Functional Requirements:**

A. Given a long url => short

url Premium user:

A. Custom url (optional)

B. Expiry date

B. Given a short url => redirect to the => long url

2. **Non-functional requirements:**

100 million daily active user => 1 million users are actually doing shorting of url

1. Latency: 20ms (on url shortening, on url redirection)

2. Availability: 24 x 7

3. Consistency

4. Scalability: Vertical or horizontal

5. The generated short url -> always unique

3. **Api design (url shorter):**

Pre-defined functions:

1. Get: Fetch some data from db.
2. Post: When you want to insert some data into db.
3. Put / Patch: Update
4. Delete: Remove the data

Local host server: Https://127.0.0.1/shorten

App.route(/)

1. Post Api call - Req

{

Logic:

Url: "Long url "

Custom url: "Custom"

Expiry date:

}

Res:

{

Short_url: "Short url "

Short_code: "123abc"

}

Https://127.0.0.1/123abc

2. Get api call (</short_code?)

Https://127.0.0.1/get_employee(empid)

4. **Database schema design:**

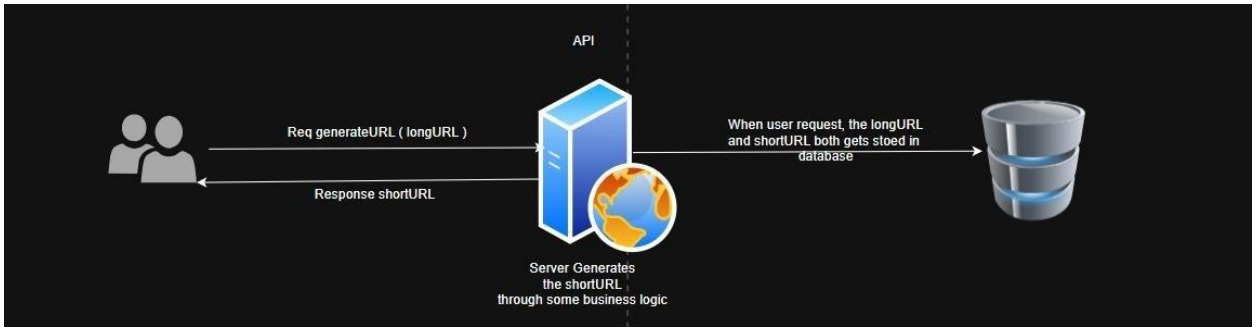
T1 user (id, name, phone etc.)

T2 url_mapping (id, longurl, shorturl, expiry date, customurl)

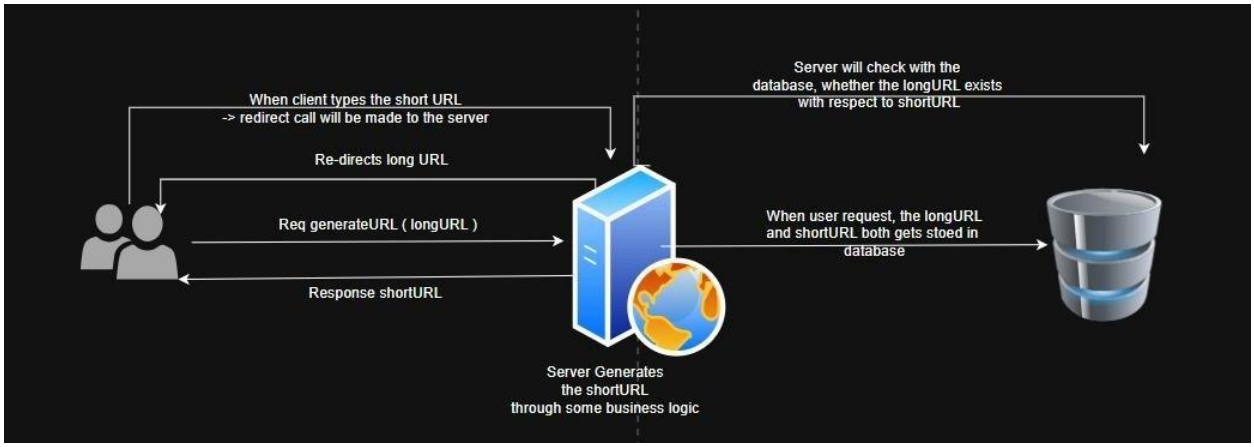
5. **High Level Design:**

Now according to the functional requirement of the system, we can identify that: There will be a client who is requesting, then there will a server upon which computation will be going on, and lastly there will be a database in which storage will be done.

1. shortURL Generation:



2. Re-direction: When user enters shortURL in browser:



3. We need to have REST APIs for client registration and login as well.

6. **Low Level Design:**

Now if you see in HLD we have only specified the specific things like the database, the server, but we haven't talked about:

1. Which type of database is needed??

2. Which type of server is needed and what sort of computation has been done inside the server.

Let's begin with the SERVER: how longURL is converted into shortURL

longURL: https://www.amazon.com/rehan.products.catalog/orders/cart/payment

shortURL: https://bit.ly/5PLcymn

{5PLcymn} - shortened URL

LONGurl - BASE10 - LARGE STRING

BASE 64 - HDGB25

Approach 01: Encryption

encrypt(longURL) = shortURL

Most Popular algorithm for encryption lib is: MD5, SHA1, BASE64 Problem

in this approach:

1. Encrypted length is very large & the req was of short length url

2. To resolve this, what we can do is we can take first 4 letters from encrypted text. But in this also a problem can occur.

Link 1:
https://www.youtube.com/watch?v=HHUi8F_qAXM&t=1s06e2116c3e064129e81226fb7b57502c

Link 2:
https://www.youtube.com/watch?v=jhjsdbi8793hch?2fdgv06e265f903040b075b90ed40ad15a4f6e5a184a2

[both starting 4 char are same, pointing to same link in DB Duplication of links]

Resolution:

At first, we can store the 4byte code & corresponding longURL in DB. Second time, if same 4byte code is generate, it will be compared in the DB, if code already exist -> again generate a MD5, 4byte unique code.

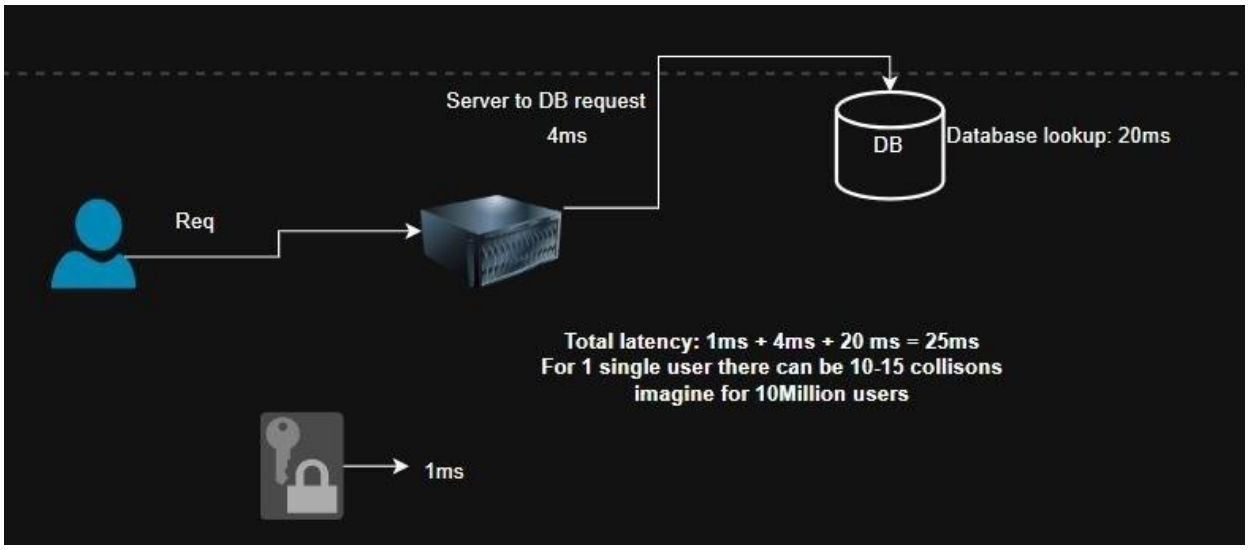
[This results into high latency issue = 16ms + 16ms]

Possible Problems:

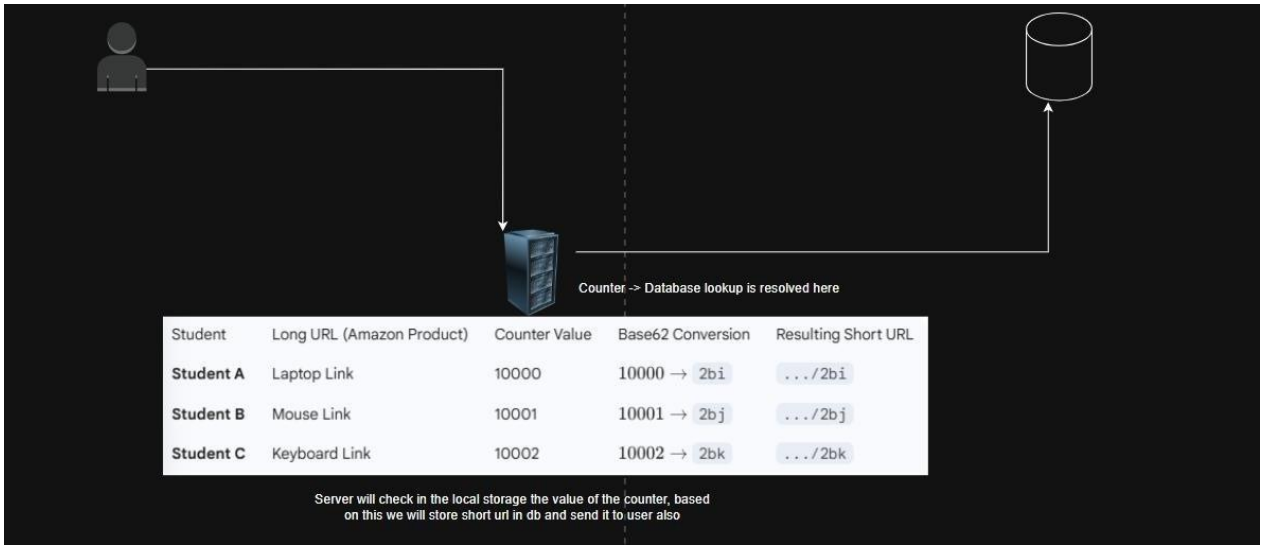
1. Full table scan in database.

2. More number of collisions

3. These collisions will result into latency issue



Approach 02: Counter Approach



Student	Long URL (Amazon Product)	Counter Value	Base62 Conversion	Resulting Short URL
Student A	Laptop Link	10000	$10000 \rightarrow$ `2bi`	`.../2bi`
Student B	Mouse Link	10001	$10001 \rightarrow$ `2bj`	`.../2bj`
Student C	Keyboard Link	10002	$10002 \rightarrow$ `2bk`	`.../2bk`

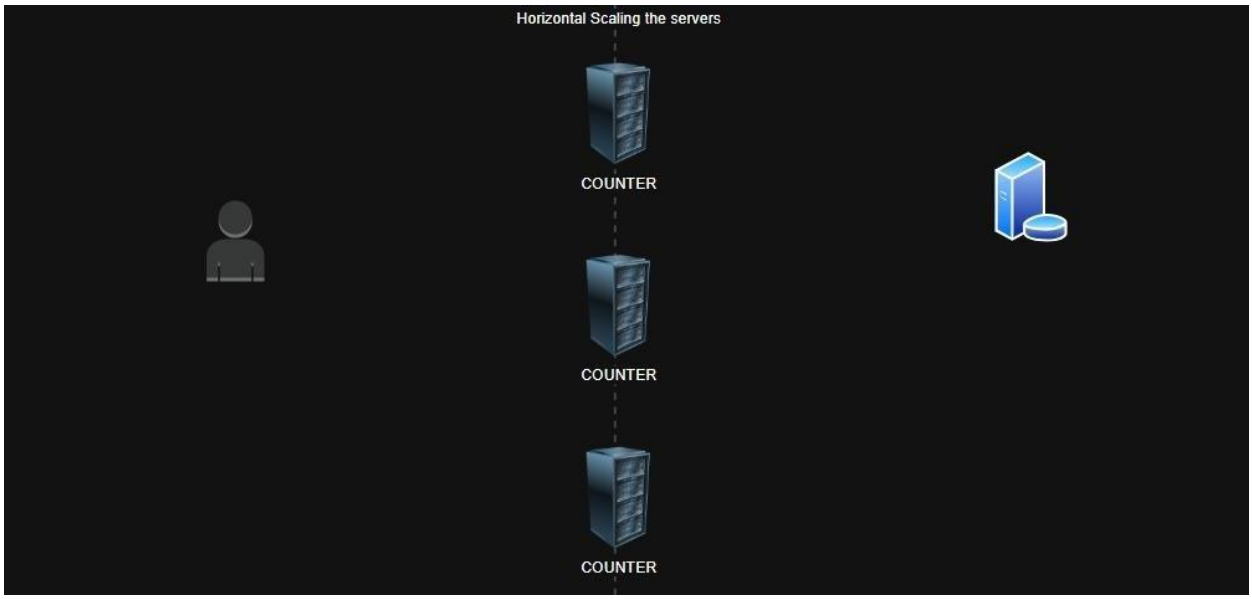
Problem with this approach:

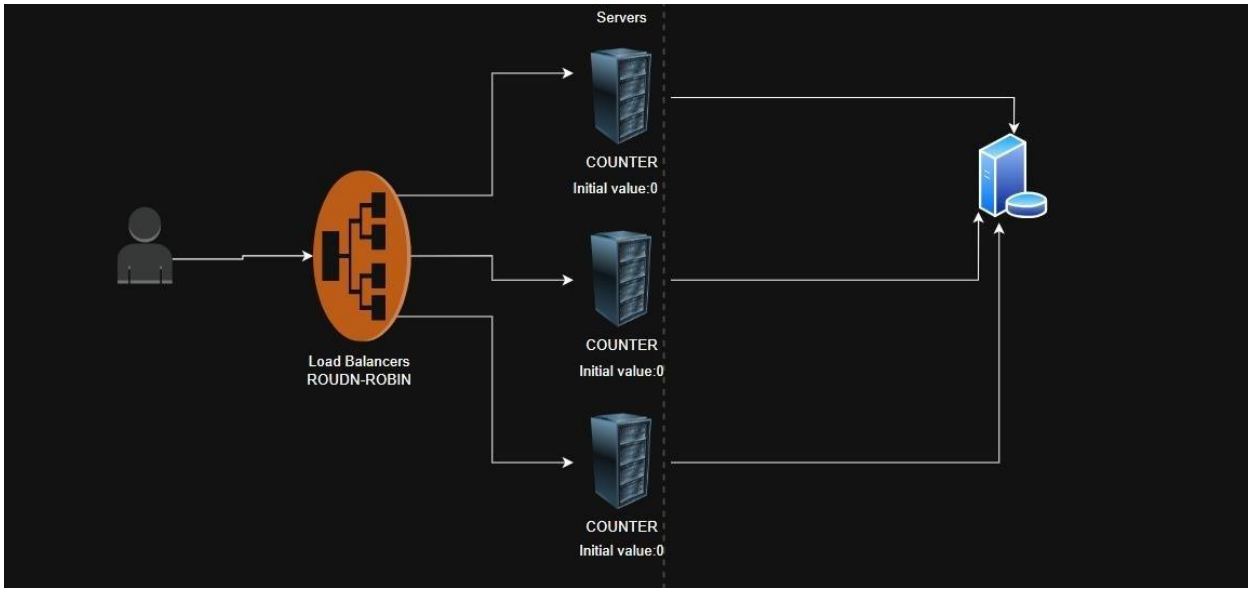
Server follows monolithic architecture, it can process the request for 1 user easily by managing counter value, but what happens when 100 million user comes.

Resolution: Scaling

- Vertical scaling here is not possible as number of users are 100 M

- We will do the horizontal scaling over here





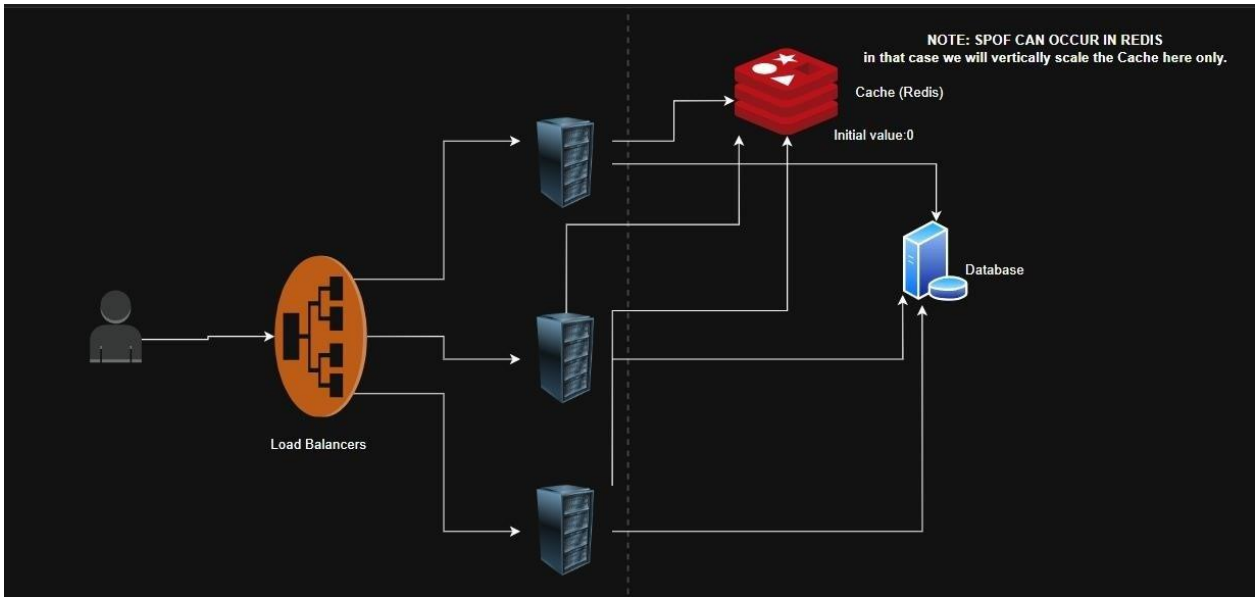
Problem:

Now in this method there can be a problem of dirty read that is server 1's counter may have some other value and server 2's counter may have some other value

Resolution:

Rather than storing the counter in the server itself, we can make the counter variable globally available using a CACHE

Final Low-level Design:



4. Learning Outcomes:

a. Understood and differentiated functional and non-functional requirements for designing a URL shortener system.
b. Designed and tested RESTful APIs for URL shortening and redirection using Flask and Postman.
c. Created high-level and low-level system designs to ensure scalability, availability, and performance.
d. Designed database schemas and implemented backend logic for URL mapping and user management.