EXPERIMENT – 01

URL Shortener Designing

1. Objective

The objective of this experiment is to design a **URL Shortener system** that converts long URLs into short, unique URLs and redirects users efficiently while meeting high availability, scalability, and low latency requirements.

2. Requirement Gathering

2.1 Functional Requirements

1. The system should generate a **short URL** from a given **long URL**.
2. The system should support **custom short URLs** (optional).
3. The system should support **URL expiration**:
    - Default expiration
    - Custom expiration date
4. Users should be redirected to the **original long URL** when accessing the short URL.
5. REST APIs should be available for:
    - URL creation
    - URL redirection
    - User registration and login

2.2 Non-Functional Requirements

Requirement	Description
Low Latency	URL creation and redirection should respond within **200 ms**
Scalability	Support **100 million daily active users** and **1 billion shortened URLs**
Uniqueness	Every shortened URL must be unique
Availability	System should be available **24×7**
CAP Tradeoff	**Availability > Consistency**

3. Core Entities of the System

1. **User**
2. **Short URL**
3. **Long URL**

4. API Endpoints Design

4.1 Create Short URL

- **Method:** POST
- **Endpoint:** `/v1/url`
- **Request Body:**

```
{
  "longURL": "string",
  "customURL": "string (optional)",
  "expirationDate": "date"
}
```

- **Response:**

```
{
  "shortURL": "string"
}
```

4.2 Redirect to Long URL

- **Method:** GET
- **Endpoint:** `/v1/url/{shortURL}`
- **Functionality:**
  Redirects the user to the corresponding long URL if it exists and is not expired.

5. High Level Design (HLD)

5.1 Architecture Overview

The system consists of the following components:

1. **Client** – Sends requests to generate or access short URLs
2. **Server (Application Layer)** – Handles business logic
3. **Database** – Stores URL mappings
4. **Cache** – Stores counters and frequently accessed URLs

5.2 URL Creation Flow

1. Client sends a request with a long URL.
2. Server generates a unique short URL.
3. The server stores the mapping of short URL and long URL in the database.
4. Server responds with the generated short URL.

5.3 URL Redirection Flow

1. The user accesses the short URL.
2. Server checks the database/cache for the mapping.
3. If found and valid, the server redirects the user to the original long URL.

6. Database Schema Design

6.1 User Table

Column Name	Description
user_id	Unique user identifier
username	User name
email	User email
password	Encrypted password
created_at	Account creation time

6.2 URL Table

Column Name	Description
id	Primary key
shortURL	Generated short URL
longURL	Original long URL
customURL	Custom alias (optional)
expirationDate	Expiry date
created_at	Creation timestamp

7. Low Level Design (LLD)

7.1 URL Shortening Approaches

Approach 01: Encryption-Based Approach

Method:

- Apply hashing algorithms such as **MD5**, **SHA1**, or **Base64** to the long URL.

Example:

Long URL → MD5 Hash → Short Code

Problems:

1. Generated hash length is large.
2. Truncating hash increases collision probability.
3. Different URLs may produce the same prefix.
4. Requires database checks for collisions.
5. High latency due to collisions and full table scans.

Approach 02: Counter-Based Approach (Preferred)

Method:

1. Maintain a global counter.
2. Convert the counter value to **Base62**.
3. Use this encoded value as the short URL.

Example:

Counter Value → Base62 → Short URL

7.2 Challenges in Counter Approach

1. Single server counters cause scalability issues.
2. Horizontal scaling causes **dirty reads**.
3. Multiple servers may generate duplicate counter values.

7.3 Solution: Distributed Counter Using Cache

- Use a **distributed cache (Redis)** to store the counter.
- All servers fetch and increment the counter from Redis.
- Load Balancer distributes traffic using the **Round Robin** strategy.

7.4 Architecture Components

Component	Purpose
Load Balancer	Distributes incoming traffic
Application Servers	Handle API requests
Redis Cache	Global counter storage
Database	Persistent storage

7.5 Single Point of Failure (SPOF)

- Redis can become a SPOF.
- **Solution:** Vertical scaling and replication of Redis.

8. Performance Analysis

Operation	Estimated Latency
Cache Lookup	1 ms
Server Processing	4 ms
Database Access	20 ms
Total Latency	**~25 ms**

This meets the requirement of **< 200 ms** response time.

9. Conclusion

The URL Shortener system is designed to be:

- Highly available
- Horizontally scalable
- Low latency
- Collision-free using a distributed counter approach

By using **Redis for counter management**, **Load Balancers**, and **eventual consistency**, the system efficiently supports millions of users and billions of URLs.