Design URL Shortening Service like TinyURL

Asked in: Google, Facebook, Amazon, Adobe

Let’s understand the problem

Design a URL Shortening service like Tiny URL. The goal is to design a highly scalable service that could allow users to create shorter URLs, given long URLs, and have read and write functionality.

What is Tiny-URL?

Tiny-URL is a URL-shortening web service that creates shorter aliases for long URLs. Whenever the user visits the short URL, the user will be redirected to the original URL.

Before jumping to the solution, as an interviewee, we should always discuss with the interviewer what he/she wants. This brings us to a discussion about the requirements and features of the system.

Requirements of the System

The URL shortening service like Tiny-URL should have the following features:

Functional Requirements

- Users should be able to generate a shortened URL from the original URL.
- The short link should redirect users to the original link.
- Users should have the option to give a custom short link for their URL.

Design Goals

- If the system fails, it will imply that all the short links will not function. So our system should be highly available.
- URL redirection should happen in real-time with minimal latency.
- Shortened links should not be predictable in any manner.

Extended/Optional Goals

- The service should be **REST API** accessible.
- **Analytics:** How many times the URL is visited?
- Users should be able to specify the **expiration time** of the URL.

System Analysis

This section will examine the estimate of the number of monthly requests, storage, and so on. The most important thing to keep in mind here is that our system will be read-heavy, meaning the number of read requests per second can be up to 1000 times greater than the number of writing requests. We will assume that our read/write ratio is 100:1.

key components

The following are the key components of the URL-shortening application:

1. **Clients:** Web Browsers/Mobile apps. They will communicate with the backend servers via the HTTP protocol.
2. **Load Balancer:** To distribute the load evenly among the backend servers.
3. **Web Servers:** Multiple instances of web servers will be deployed for horizontal scaling.
4. **Database:** It will be used to store the mapping of long URLs to short URLs.

Estimation of Scale and Constraints

Traffic Estimation: We can assume that we will have 500 million new URL shortening requests per month. With a 100:1 read/write ratio, we can expect 50 billion redirections during the same period (100 * 500 million =

50 billion).

- If we calculate this value for each second, i.e., queries per second (QPS) = 500 million / (30 days * 24 hours * 3600 seconds) = ~200 URLs/s.
- Considering the 100:1 read/write ratio, URL redirections per second will be = 100 * 200 URLs/s = 20K/s.

Storage Estimation: Let's assume that we are willing to store our data (short URL + long URL) for ten years. Then, the number of URLs we will be storing would be = 500 million * 10 * 12 = 60 billion URLs.

Now, let's assume the average URL length to be 120 characters (120 bytes), and we add 80 more bytes to store the information about the URL. The total storage requirement would be = 60 billion * 200 bytes = 12TB.

Encoding URL

What is the minimum length of the shortened URL to represent 60 billion URLs? We will use (a-z, A-Z, 0-9) to encode our URLs, which leads us to use base62 encoding. Now, the question arises: what should be the length of the short URL generated to cater to 60 billion requests? Let us find out.

The longer our key, the more URLs we have, and the less we have to worry about our system ever running out. However, the point of this system is to keep our URL as short as possible. Let's stick with **7** because even if we consumed 1000 keys per second, it would take 111 years to run out!

Encoding

To generate a unique short URL, we can compute it using the unique **hash (MD5, SHA256, etc.**) of the original URL and then encode it using **base62**. If we use the MD5 algorithm as our hash function, it will produce a **128-bit hash value**. After base62 encoding, we will get a string with more than seven characters. We can take the first 7 characters for the key.

MD5(Long URL) -> base62encode(128-bit hash value) -> **FE66AB7**1IT...

There is a problem with this approach, as it could (however unlikely) result in duplications or collisions in the database. One solution to this problem is to use a counter, which keeps track of a count (0-3.5 trillion) so that in each case if we encode the counter value, there is a guarantee of no duplicates or collisions.

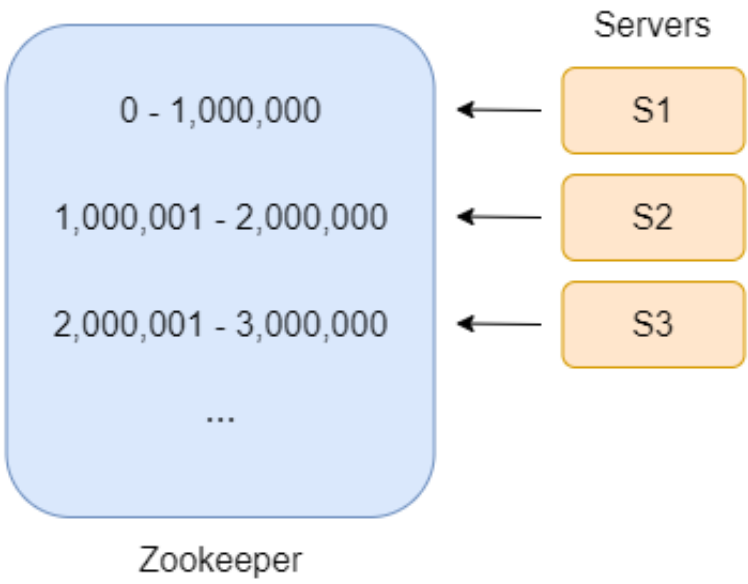
Let's see how we will use the counter. We will use a single server that will be responsible for maintaining a counter. Whenever the application servers receive a request, they will talk to the counter, which returns a unique number and increments the counter.

Two potential problems

1. Single Point of Failure.
2. If requests spike, our counter host may not be able to handle it.

In this case, we will use a **distributed systems manager**, such as **Zookeeper.** Let us see how a distributed systems manager like Zookeeper solves our problem:

1. It gives our application servers unused ranges.
2. To avoid having Zookeeper be a single point of failure, we will run **multiple instances**.
3. If a new server is added, it will give them an unused range. If the range runs out, the existing server can go to Zookeeper and ask for a new, unused range.
4. If one of the **servers dies**, **1 million keys** are possibly **wasted**, which is **acceptable**, given the 3.5 Trillion keys we have.
5. However, sequentially generating URLs can be a security threat. We can add another random 10–15 bits after the counter number to increase the randomness.



Base62 encoding algorithm

```
def base62encode(long_URL_counter):
    base62_string = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    hash_string = ''
    while long_URL_counter > 0:
        has_string += base62_string[long_URL_counter % 62]
        long_URL_counter /= 62
    return hash_string
```

System APIs

We can use **REST APIs** to expose the functionality of our service. Below is an example of the definitions of the APIs for creating and deleting URLs without service:

Parameters

- `api_dev_key` (string): The API developer key of a registered account. This will be used to throttle users based on their allocated quota.
- `original_url` (string): Original URL to be shortened.
- `custom_alias` (string): Optional custom key for the URL.
- `user_name` (string): Optional username to be used in the encoding.
- `expire_date` (string): Optional expiration date for the shortened URL.

Returns (string): A successful insertion returns the shortened URL.

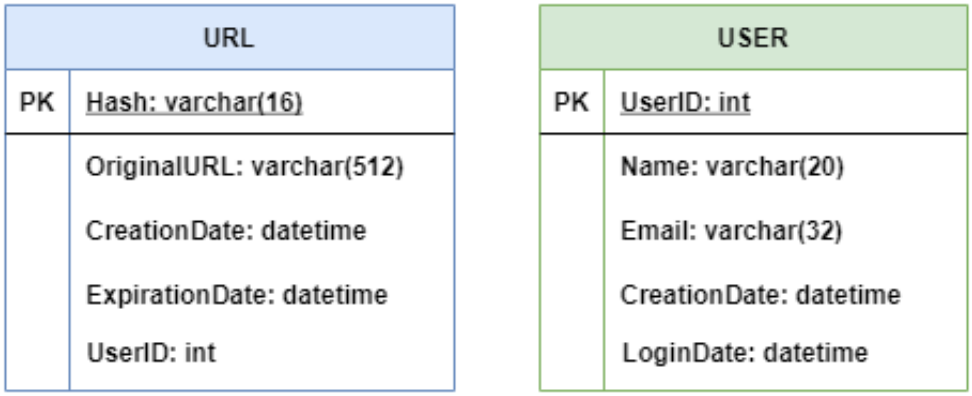
Otherwise, it returns an error code.

The `url_key` is a string that delineates the shortened URL that needs to be deleted. A successful deletion will return. `URL Removed`.

Database design

While designing systems, we have two types of databases (of course, we don't want to back to the old days where file systems were used): **Relational Databases** or **NoSQL.** For our system, relational queries will not be a large part of it/occur rarely. So, here we will go with NoSQL. **A NoSQL choice would be easier to scale.**

Database schema: We will need two main tables: 1 to store user information and 1 to store URL information.



Scaling the service

Caching

We know that our database is going to be read heavily. We have found a way to speed up the writing process, but the reads are still slow. So we have to find some way to speed up the reading process. Let's see.

We can speed up our database reads by putting as much data in memory as possible, AKA a **cache**. This becomes important if we get massive load requests to a single link. If we have the redirect URL in memory, we can process those requests quickly. Our cache could store, let's say, **20% of**

the most used URLs. When the cache is full, we would want to replace a URL with a more popular one.

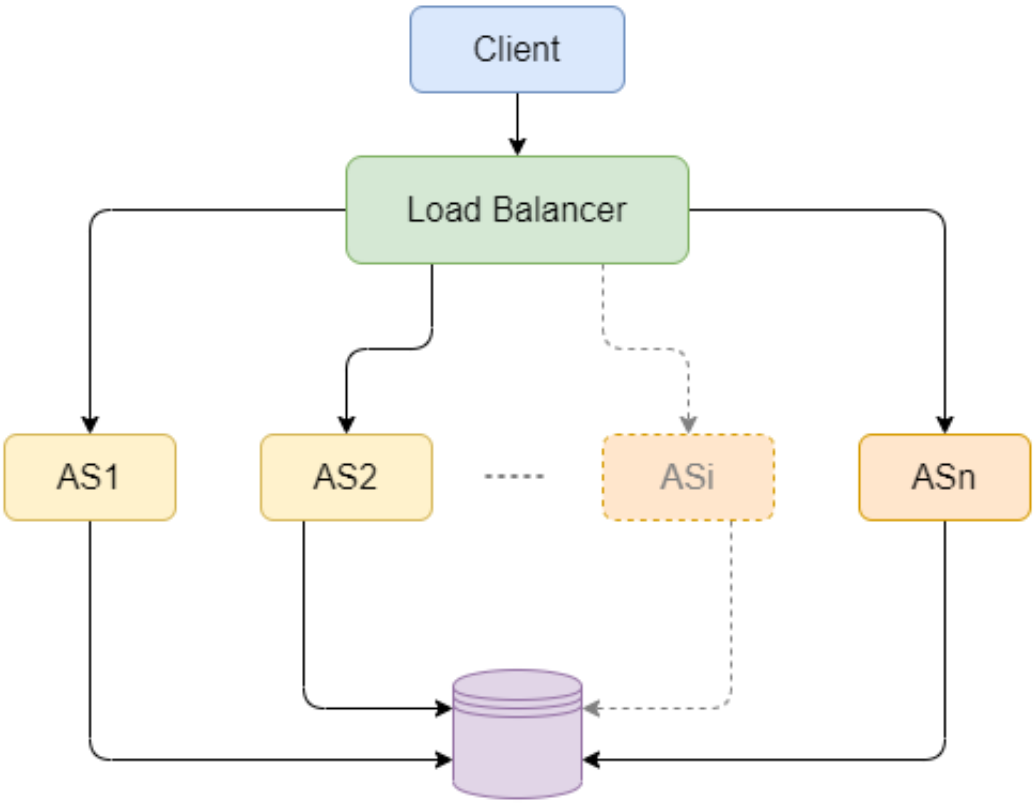
A **Least Recently Used (LRU)** cache eviction system would be a good choice.
We can **shard** the cache too. This will help us store more data in memory because of the more machines. Deciding which thing goes to which shard can be done using "**Hashing"** or "**Consistent Hashing."**

Load Balancing

With multiple access systems like this one, a web server may not handle it yet. To solve this problem, I will use many web servers. Each web server will take a partial request from the user.

Since we have a large-scale system with multiple access happening simultaneously, a server may not handle the requests. To solve this problem, we can use multiple servers with a load balancer between **"the client and the server"** and "**the server and the database"** and avoid a single point of failure, and we will use multiple load balancers.

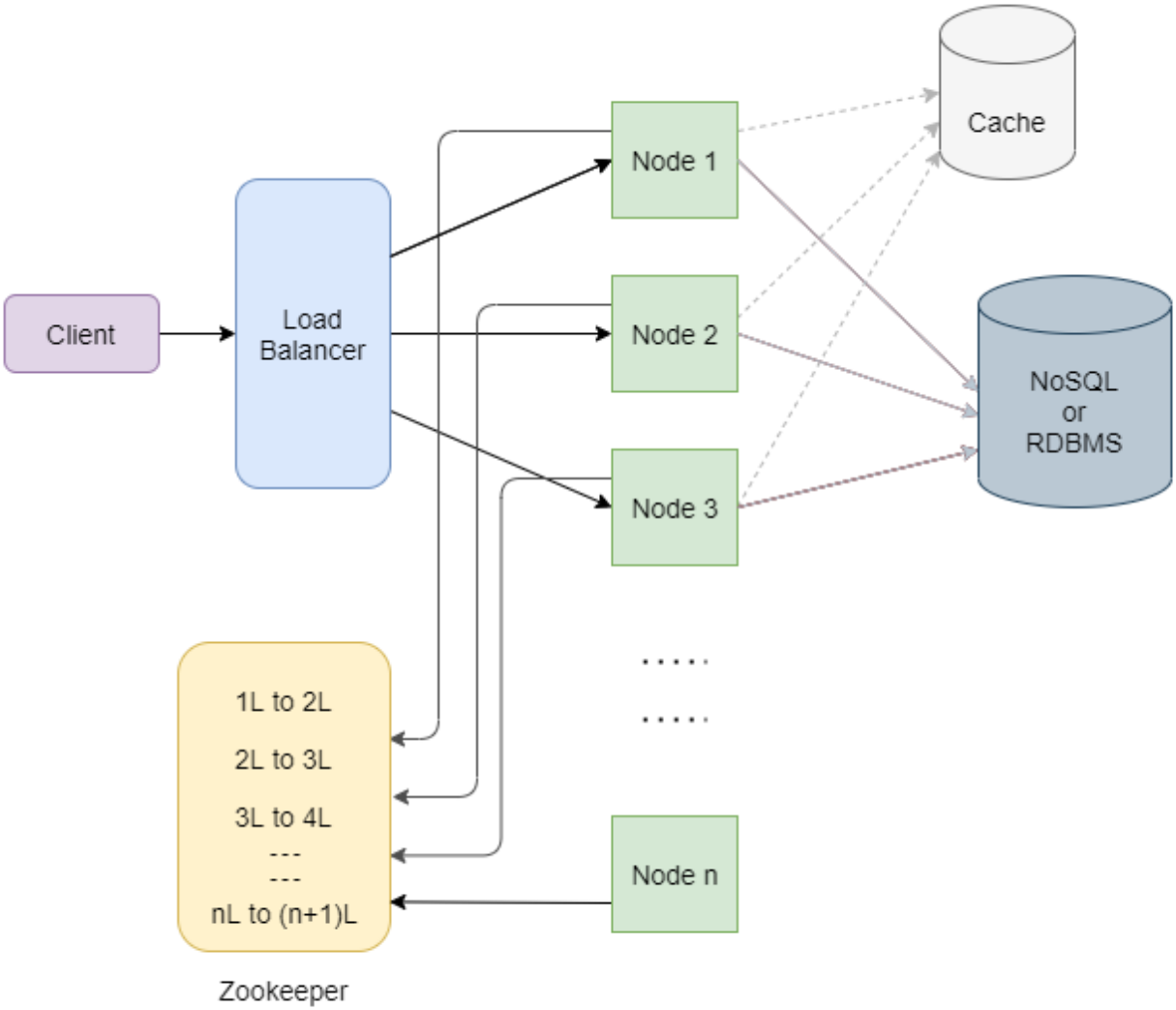
High Level Design



Initially, we can use a simple **Round Robin** approach that distributes incoming requests equally among backend servers. This would be the easiest to implement. A Round Robin LB does not consider server load, so our LB may still forward requests to an overloaded or slow server.

To distribute the load among servers, we will use the **Least Connection Method.** When a server is configured to use the least connection load balancing algorithm (or method), it selects the service with the fewest active connections.

Skeleton of the design



Hence in this way, we could design a highly scalable **Tiny-URL** service.

Idea to think!

1. Should we choose Consistency or Availability for our service? (Hint: **CAP Theorem**)
2. How would you shard the data if you were working with SQL DB? (Hint: **Consistent Hashing**)

Thanks to Chiranjeev and Navtosh for their contribution in creating the first version of this content. If you have any queries or feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!