**URL Shortner System Design:**

**URL Shortener Service:**

* Takes in a URL and produces a shorter link.
* The shortened URL redirects the user to the original URL
* Examples: bit.y, tinyurl, goo.gl
* Use cases:
  + Embed on business card
  + Share on social media

**Functional requirements:**

* Input original URL, get a shorter link.
* When accessing short link, user is redirected to the original URL.
* Short Link must be human-readable (if user is typing in browser, then there is less room for errors).
* All links expire after 2 years of inactivity.
* The original and short URLs will be stored for 5 years.
* Creation of 300 Million new short URL per month.
* 200 redirection requests per short URL per month.

**Non-Functional Requirements:**

* We focus on
  + Response time
  + Availability
* Short URLs must be random
* Short URLs must be unique

**Questions can be asked:**

* What is length of URL which is shortened
* What is traffic/volume of traffic per second
* Is system single instance or should we scale it?

**Assumptions made/Capacity planning:**

* Volume/traffic: Twitter- 300M users/month

So, we can assume 30M users/month. i.e. 1 M users per day

* URL Length- e.g. [www.us.com](http://www.us.com) + 7 characters

**Data Capacity Model:**

|  |  |  |  |
| --- | --- | --- | --- |
| Column name | Datatype | Size | description |
| Long URL |  | 2kb (2048 characters) |  |
| Short URL |  | 17 bytes (17 characters) | [www.us.com](http://www.us.com) + 7 characters |
| Created at | Datetime (epoch time) | 7 bytes (7 characters) |  |
| Expired at | Datetime (epoch time) | 7 bytes (7 characters) |  |
|  |  | 2.031KB | Total size |

For 30 Million users, we will save 60.7 GB per month, which will be 0.7 TB per year and 3.6 TB data for 5 years.

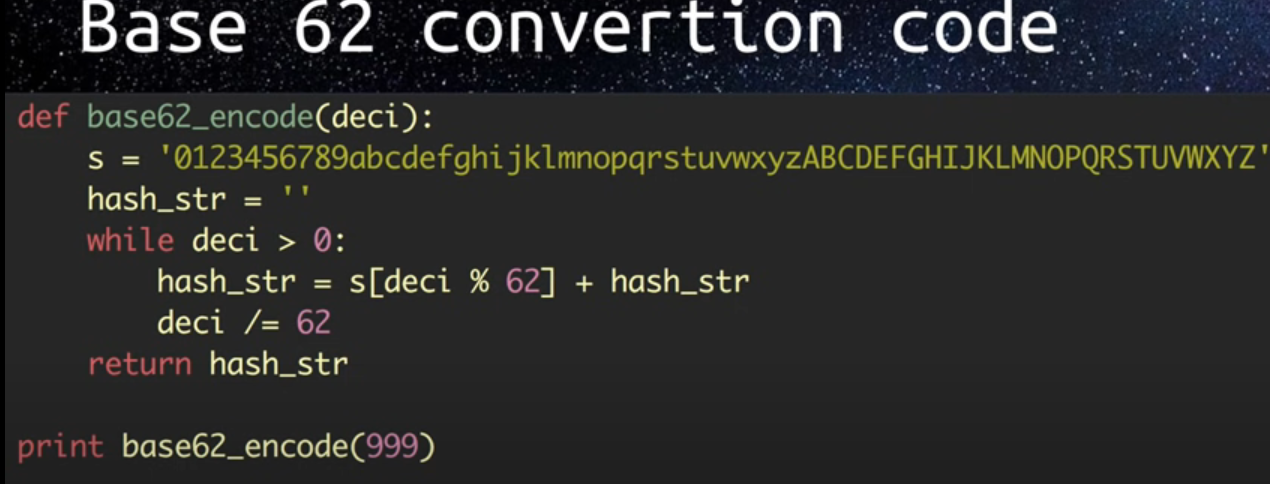
URL shortening logic:

e.g. www.us.com/ABC12qc

* 2 algorithms used to calculate unique id(7 character random ID) for Shortened URL which will redirect to long URL
  + Base62
  + MD5 hash
* Base 62: It takes decimal no/integer/long and it gives out alphanumeric (A-Z, a-z,0-9)
* MD5 hash: We also get same output same as base62, but problem is it gives very long output around 22-25 characters. So, we have to take only 1st 7 characters which can cause lot of collisions and data corruption.

**Why use base62:**

* We can base10 which will generate unique id with only digits from 0-9. But,it will generate only 10^7 combinations of unique ids which is 10 Million only.
* Whereas in base62, it will generate unique id using A-Z(26)+a-z(26)+0-9(10)=62. So it will generate 62^7 unique id combinations. i.e. 3.5 Trillion

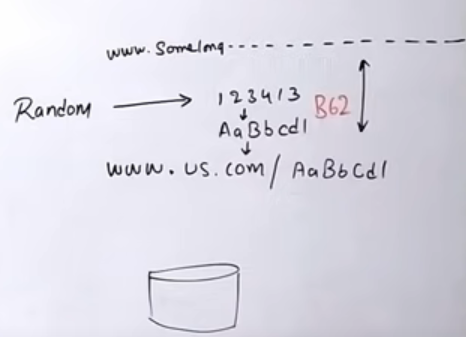


**What database to choose for storage:**

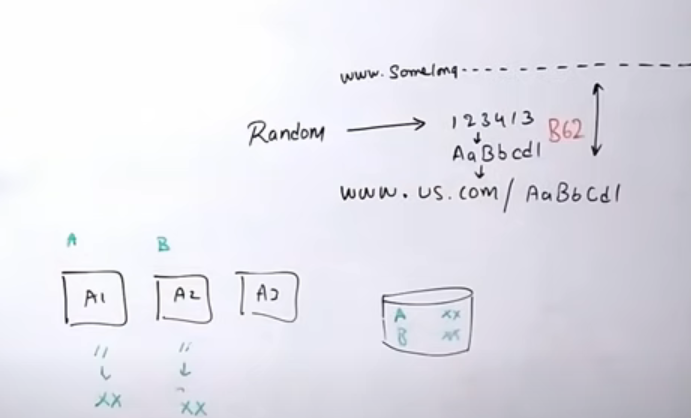
* In case of RDMBS, it is good for ACID property, but problem is scaling. If we talk about 30 M users per month, then there will be lot of reads and writes which is pretty hard to scale RDBMS.
* In case of NoSQL, the problem is eventual consistency (we write something, which will take time to replicate to different nodes). But good thing is high availability and easy to scaling. (keep adding nodes when we want to expand storage).

**technique1- Inserting short url mapping to long url in Database (Base62):**

Random number is an input to the base62 which will generate unique\_id to create short URL.



* This short unique URL might be already present in the DB cz same random no could be re-generated for another long URL.
* First, we need to check if the Short URL which was generated is already present in database. If present, then we should generate another random number and again check in database. IF not present, then insert mapping of long and short urls to the database.
* But this technique is feasible for simple systems where you have 1 app server and no parallel processing that inserts data into database.
* In case of multiple app servers inserting data into database, if multiple app servers generate same short URL for different Long URLs, then there will be data corruption.
* In this situation, we can use putIfabsent() or insertIfabsent() technique. But the issue is functionality is not available in NoSQL database.



**Technique2- Inserting short url mapping to long url in Database (Md5 hash):**

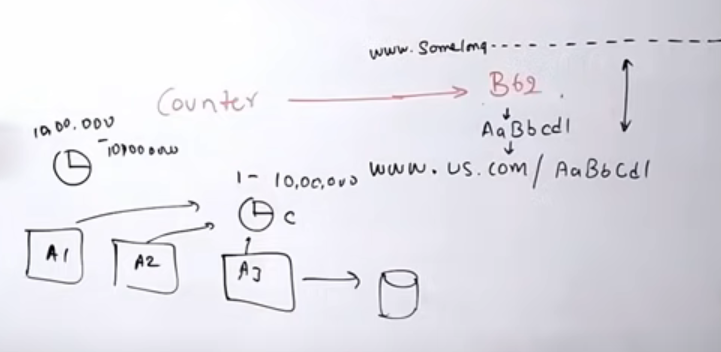
Long URL is an input to the MD5 which will generate long characters out of which we can select 1st 7 characters as a random number and it will be passed to base62 to create unique\_id for short URL. In this case, collision can be reduced to some extent but still collision issue will be there. Because, it could be possible that MD5 will generate 1st 7 characters for another long URL. This will cause same problem as that of technique1 of data corruption.

**Technique3 (Using Counter and Zookeeper):**

We can maintain counter which will be provide next counter to each of app server. That counter can be use as random number to generate short url.

We can maintain 2 different counters in different regions (Aisa, EU). One generating 1 to 1 Million counters while other generates 1Million to 2 Million. This will be better approach but the problem is what if counter goes out of range, who will reset the counter and how will it get resolved?

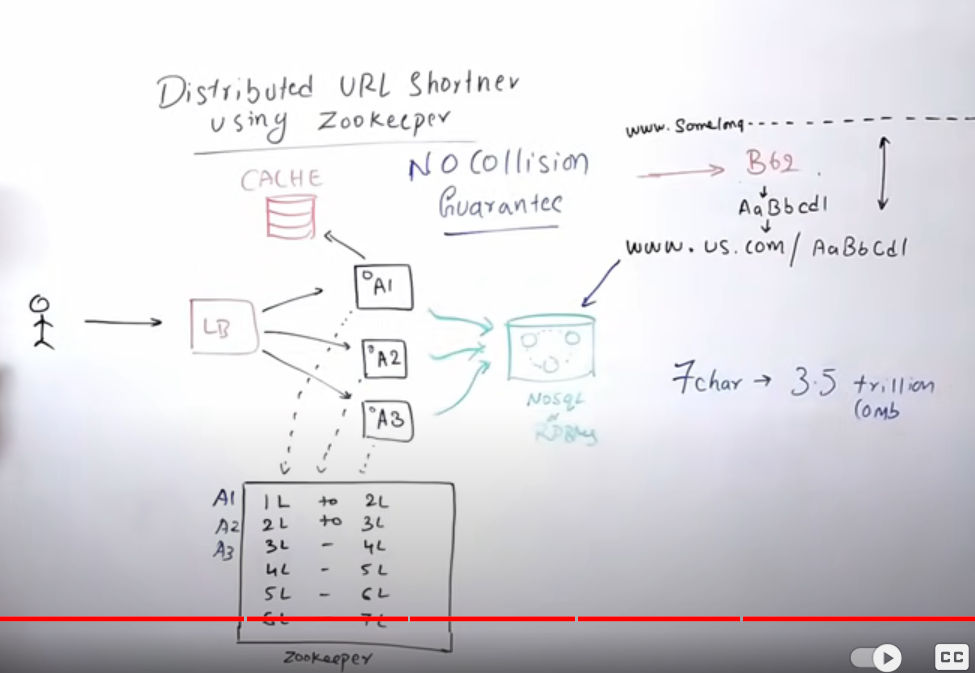
Also, if counter service will become single point failure for each region.

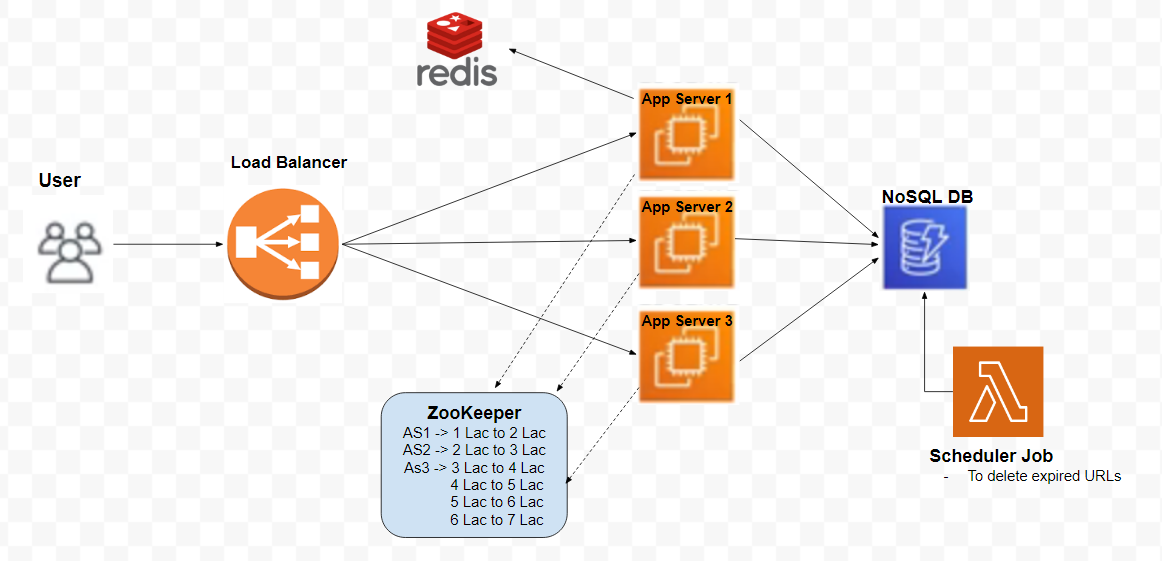


We can introduce Zookeeper to resolve this issue.

* It takes care of coordination between multiple services in a distributed env. So, developer hasn’t worry about coordination and focus on core development.
* All host/app servers are registered with Zookeeper. All host/app servers information/configurations whether they are active or down is available with Zookeeper. It also co-ordindates between other app servers. It also maintains synchronization between all app servers.

Distributed URL Shortner System design using Zookeeper:





* Zookeeper maintains ranges for counters.
* App servers will communicate with zookeeper to get their range.
* When zookeeper gets calls from app servers, it will assign range to app servers.
* Now app servers are responsible to generate counters between ranges assigned to them.
* If any new app server is added to the system, then first it will get registered with zookeeper and get new range.
* If any app server goes down, zookeeper will come to know that server is down. If that server has generated a few counters then loosing a few counters will not be a loss. Zookeeper will erase that range.
* If app server has consumed its whole range, then it will call the zookeeper to get next range. Zookeeper will erase range which was assigned earlier and then will assign new range to server.
* So, this is guaranteed no collision solution.
* Reddis cache can be used here which is centralized cache db for all app servers.
* Database NoSQL is highly scalable, Load balancer is scalable and reddis is also scalable.
* Zookeeper is highly available.

APIs needed:

* Createshorturl (LongURL) – We need to pass longurl to createshortl api.
* Getlongurl (ShortURL) – We need to pass shorturl to getlongurl api.

In order to have analytics to know how application is performing, we can use elasticSearch/ openSearch which comes with analytics functionality.

