**Tiny URL**

Functional Requirements:

1. Given a long URL, return a short URL.
2. Given a short url, return a long URL.

Non-Functional Requirements:

1. Very low latency
2. Very high Availability

**Length of the URL / Scale**

1. What is the traffic? How many urls would come for shortening & for how much duration I need to save it?
   1. X number of requests come every sec & we want to store the URL for 10 years
   2. X\*60\*60\*24\*365\*10 = Y = System should handle these many unique requests
   3. Characters which can be included in the url = a-z, A-Z, 0-9 = 62 characters.
   4. We need to find a n for which 62^n > Y
      1. n = log(base 62)(Y)
   5. Let’s say that n turns out to be 5, then we can create Y number of URLs for the next 10 years in just a length of 5.
   6. 62^6 = 58 billion
   7. 62^7 = 3.5 trillion

**System Design:**

A picture containing text, whiteboard

Description automatically generated

1. UI gives a long URL and forwards it to the short URL service which returns the short url to the user.
2. UI gives a short URL and forwards it to the short URL service which fetches the long url from the database and redirect the user to the long URL.
3. Short URL service saves the mapping into Cassandra database.
4. **Short URL service:** There are multiple instances of this service and what if some of them have a request at the same time and they generate the same short URL causing collision?
   1. **Solution 1:** We can first check in the database if this URL is not taken, then only keep it, otherwise keep generating a new one. This way is not so efficient since it involves a call to the database.
   2. **Solution 2:**

A picture containing text, whiteboard

Description automatically generated

* + 1. Using a feature of REDIS which generates an auto incrementing number. We ask all the services to talk with the REDIS service asking for a number and we are sure that Redis would provide a unique number to them. The service can then convert the base 10 number returned by Redis into base 62 and this would be a unique URL. The issue with this is
    2. that all the instances of short url service will contact REDIS and it will be under huge amount of load.
    3. REDIS becomes a single point of failure which is bad design.
    4. Multiple REDIS is also a problem since then we will have to synchronize between them somehow on that number since if they generate duplicate numbers then that is a problem.
  1. **Solution 3:**

Diagram

Description automatically generated

* + 1. We introduce a Token Generator which will generate a range and give it back to a service instance. The Short URL service may request this token generator occasionally when it exhausts its tokens and then it gets a new range which it starts using. Hence, the token service will be having few requests only occasionally.
    2. This token generator will have a MySQL DB at the back to keep track of the last range it assigned and will respond to a request with a new range and update the record so that the next request can have another range. MySQL since data needs to be consistent here. We can’t have two services working on the same range of tokens.
    3. We’ve divided the system into two since short to Long is simple. Just accessing the Cassandra and returning it. Long to Short URL is also simple just that the Short URL service will check if it has tokens, then it will consume a token and convert it into base 62. But if it has exhausted the tokens, it will ask for a new range and then convert a token from it. After that, it saves it into a Cassandra and returns the short URL code.
    4. Why Cassandra? -> Use any Database which can handle 3.5Trillion records. MySQL may have some problems with such large number of records but can be dealt with (by sharding).
    5. To scale this up,
       1. We can have multiple instances of Token Generator.
       2. We can assign bigger ranges to the service instances.
    6. Cons of this architecture:
       1. Wen a service goes down, we lose all the “remaining numbers” from that range since they are lost now. Still, this would be very few compared to the 3.5Trillion records we have prepared our design for.
       2. This doesn’t give us any metrics on how our system is being used, what kind of geographies people come from, what kind of urls are topmost used urls.
       3. If we have 4 datacenters and we want to make 2 as primary and 2 as secondary. We can take this decision based on some data which we do not have. It would be nice if we have this data and then we can use it to select 2 primary data centers around which we are getting more requests.
  1. **Solution 4:** When a client sends us a request, we check the attributes from that request and try to get some meaningful information from there (IP, device: ios, android, etc)
     1. We can put these requests “asynchronously” in a Kafka so that it doesn’t impact our performance. The only downside of this is that we may lose out some records in case of failure in kafka insertion but that okay for us.
     2. If there are too many network I/O calls to Kafka we can do batch processing. We can store all the requests in a queue and then when it reaches some size of crosses a threshold of X secs, we can then push all of them in Kafka. The con here is that we may lose out on a whole batch of requests.
     3. Kafka can dump all this data into Hadoop and we can create some HIVE queries on top of that which will generate aggregate results.