System Design - Tiny URL

Functional Requirements

1. The system should generate a url shortening service for the long url that the user inputs.
2. The system should redirect the traffic to the domain of the long url when the user sends read request of the short url.
3. The system should consider the expiration date as input for the long url.
4. It should have a clean/purge service to delete the expired long url mapped records.

Non-Functional Requirements

1. Scalability: The system should be scalable to process the increasing number of new requests and data. This scalability should be at computational and database level. So, the backend server that processes the requests and storage including both cache and database should be scalable.
2. Availability: Every non-failing node in the system should be able to respond to the user's request. For this system availability is achieved at the cost of consistency during a partition. Hence, the database selection should be the one that provides availability during a partition.
3. Security : The system will be prone to extensive abuse from the attackers with fake huge number of requests from the same attacker causing denial of service attack(DoS). Hence, it is essential to consider this in the design to avoid it.
4. Reliability : Every user that requests to access the system should get a response even if one or several of the nodes fail in the system. The system should have high throughput with a decent latency.
5. Effectiveness: The effectiveness to perform the functional requirements of the system.

API Services:

CreateURL(original_url, expiration_date, user_id, api_key)

DeleteURL(original_url,api_key)

ReadURL(short_url,api_key)

Traffic:

This system will be read heavy. The number of read request to write request is estimated to be in the ratio 100:1 for read: write. The read traffic estimated is about 500 million/month requests, which will be 500 * 10^6/30*24*3600 qps in traffic which is 200 qps for read. The write will have 2 qps. Since the write to read ratio is 1:100.

Storage:

Assuming the url data will expire in 1 year and Length of the tiny url is 6

If we have 5 million write per month. That amounts to be 5 million * 12 *5 url for 5 years -> if each url requires about 500 Bytes to store the url data in the database. The total storage data will be 5 * 10^6 * 12 * 5 * 500 Bytes -> 15 GB database storage will be needed for 5 years. To keep some buffer for increase in clients let's consider 20 GB of database storage for 5 years.

Bandwidth:

Incoming traffic -> 500 Bytes * 200 qps = 100 KB/s bandwidth for incoming traffic is needed

Outgoing traffic -> 500 Bytes * 20 qps = 500 * 20 = 10 KB/s bandwidth for outgoing traffic is needed

Memory:

Memory is required for caching the most often used urls which are also called hot urls. Caching is used for the read requests only. From 200 pqs. 20 % of the read requests will be cached. 40 qps will be cached. So, the cache memory required will be 40 * 500 Bytes = 20 KB. If we plan to refresh the cache each day. For 1 day the cache memory required will be 20 KB * 24 hours * 3600 secs/hour = 2 * 24 * 36 MB -> 1.7 GB

Database Design:

A highly available database will be used for the system. The service will get the original url from the url mapping from the URL table and redirect the read traffic to the original URL domain. For write traffic, the system will assign the key randomly to the original url and make an entry in the url table for the mapping of url key to the original url which will used for read requests for the same URL.

User Table

Name

Email

Last_Login

Account

URL Table

URL_Key(Primary)

Original_URL

Expiry_Date

User_id

Key_Pool

Unique Keys(size of 6)

[a-z][0-9][A-Z]

Architecture