Introduction

SYSTEM DESIGN - CRASH COURSE

Monday, March 4, 2024 8:58 PM

① Scalability
② Reliability
③ Availability
④ Load balancing
⑤ Cache ⑥ CAP

NON FUNCTIONAL REQUIREMENTS

② Reliability
$1000 → $1000
cash A cash B
pertaining to accuracy
want to transfer A → B
$1000 $0
reliability > availability
instagram availability > reliability
old feed is also okay.

③ Availability
Percentage of uptime of system
99.9%
when you request your system

① Scalability
How scale the system
How it reacts for multiple inputs
Vertical scalability Horizontal
increase capacity or resources of machine - only hardware
3GB RAM - 5GB RAM can't do above 5GB
multiple servers are created

④ Load Balancing / Serviceability
refers to quickly managing my distributed system.
replica of servers what happens if one is down.
Sol: 1. Repair (more time spent, but less expensive)
2. Replace (Quicker but expensive)
3. Repeat as broken (engineers repair)
How does it happen
expensive - fault tolerant

LOAD BALANCING
client request multiple requests distributed over servers which request goes where backup
multiple webservers multiple AS's
WS web server AS application server DB

LB - can be hardware
Software Less expensive but not good/effective

How to decide on replicas - no of LB's needed
Budget constraints service
How many users are using at the same time.
beyond a specific requests get the other LB ready.

How load is distributed b/w servers?
1) Least no of connections (persistent connections are active connections)
2) least bandwidth → amount of network traffic
3) Least average response time T2 → How much time does it take to respond.
4) Round Robin
5) weighted round robin. (when capabilities of system are different)
passive load balancer

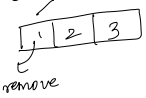
Sticky session

⑤ CACHE - locality preference
cache

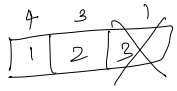
cache invalidation - data in cache is no longer valid.
① Write through cache (updates db and cache at once)
② Write around cache (updates db, you update cache when the data is different)
③ Write back cache (update the cache only, db is updated in a certain interval)

high throughput and less ... when cache is full

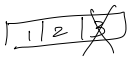
i) LRU — came in first



ii) LFU



iii) MRU — I just used this data



iv) Round Robin

v) Random replacement

vi) FIFO

vii) LIFO

12th March 2024

How do you handle a system design question.

Goal — what are you trying to achieve (what should the system do)

Scope — what all things I support (like functionalities) — am I supporting Followers, Follower

Estimation (Traffic, Storage, Bandwidth, memory, Cache) — Perform a couple of assumptions (capacity estimation, Traffic, storage, bandwidth, memory

Data Modelling — (How data flows, ER diagrams)

High level — How many web servers, App servers (can not guess at high level design)

Drill Down — (what specific load balancer I will be using, cache eviction techniques?)

Trade off — (checking if one technique is better than others)

EXTENSIBILITY — How many features can we add more

DESIGNING URL SHORTENING SERVICE (TINY URL)

Examples: google, youtube, bit.ly, tiny url

Long URL → Short URL

Functional Requirements: (what does the system supposed to do)

suggested by interviewer:

1) Long URL → Short URL → vice versa.
2) Redirect to correct long URL
3) Expiration Date (to expire short URLs after some time)
4) Custom alias

youtube.com/watch?...

tinyurl.com/aBf4G9 ↙ tinyurl.com/myweb short

NON-FUNCTIONAL REQUIREMENTS.

? will come as a part

we should think
1) Availability over consistency
2) Low latency.
3) Reliability. (recover from the fault)
4) Security. (DDOS)
5) Scalability (How does system scale in terms of raising requests)

3. CAPACITY ESTIMATION

10^6 - Million 10^3 - Kilo
10^9 - Billion 10^6 - Mega
10^{12} - Trillion 10^9 - Giga
10^{15} - Quadrillion 10^{12} - Tera
10^{15} - Peta
10^{18} - Exa

i) Traffic : queries per second.
ii) storage
iii) Bandwidth
iv) Memory (RAM, cache)

assumption : 500M users/month → arbitrary number ⟹ new requests 5 URLs per person per month.
= 500M x 5 ⟹ 2.5B write requests per month.

Is it read heavy or write heavy (short to long / long to short)
reads : writes (Read heavy)
100 : 1
50B x 5 = 250B read requests / month.

[EXPIRATION : 5 YEARS
storage/URL : 500 bytes] assumption

WRITES
$\frac{500M \times 5 \text{ (per month)}}{30 \times 24 \times 60 \times 60}$ = 1000 queries per second
(days, hours, min, sec)

READS 1000 queries per second writes x100 ⟹ 100k qps

STORAGE. 500M x 5 req/user x 500 byte x 5 years x 12 (months) ≅ 75 TB storage
(how huge the database should be)

BANDWIDTH : 500 bytes/request x 1000 qps [write requests] incoming.
= 500 k bps

500 bytes/req x 100 kqps ⟹ [read requests] outgoing.
50M bps

cache: 80:20 rule
80% of traffic will be from 20% of url

MEMORY :
read requests $\frac{250B \text{ read reqs/m} \times 0.2}{30 \text{ days}}$ → 80-20 rule
= (1.6) read reqs/day.
+ buffer (1.5)

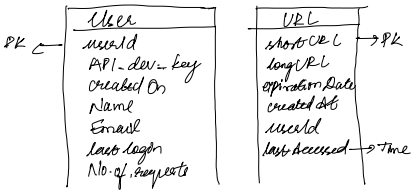
Cache $= \frac{250B \text{ read reqs/m}}{30 \text{ days}}$
= 8B read reqs/day
= 8B x 0.2 x 500 (bytes)
= 400B x 0.2
= 800B bytes

API – will have inputs and outputs.
(1) shorten URL create - URL (long_URL, expiry date, customAlias, createdBy, API_dev key, userId)
↑
To remove this URL or information

to prevent [illegible] of applications.

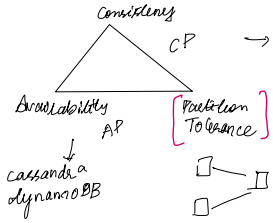
API 2) success/failure deleteURL(API_dev_key, shortURL, userId)

① DATABASE ENTITIES



SQL vs NO SQL (Read Heavy) relationship b/w users (not many joins) billions of records → NOSQL

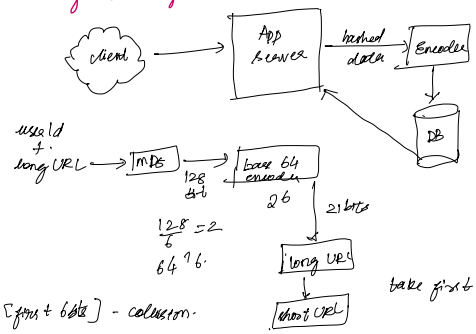
CAP theorem (NO-SQL)



should be given to these for time

How to generate unique short URL

Hashing + Encoding

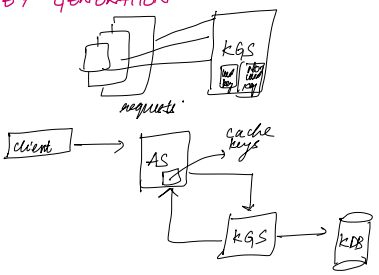


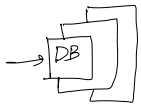
How to avoid it.
2. Timestamp append it to URL.

we can use a key generation service

longURL → Short URL.

2. KEY GENERATION





If request go to same DB the load on DB increases

another approach

tinyURL / a a 2 6 4 X

a–d → DB

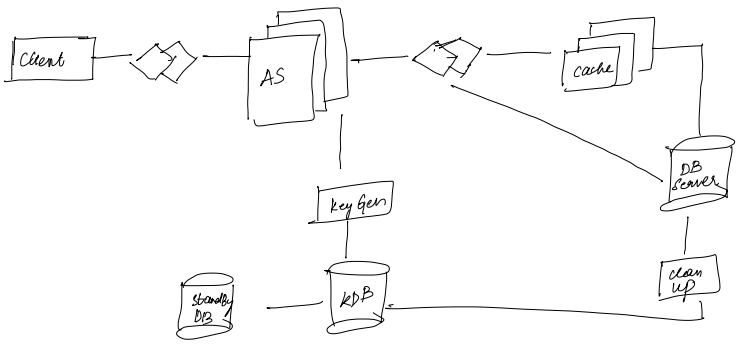
e–k → DB

l–z → DB

requests from a–d are served by DB?

shortURL → Hash code % 4

Component design.


weighted round robin
Client
AS
cache
DB Server
key Gen
standby DB
KDB
clean up
when does the cleanup happen ?
1. a job scheduled regularly checks for expiry and removes it
2. when requester comes, check for expiry
we both