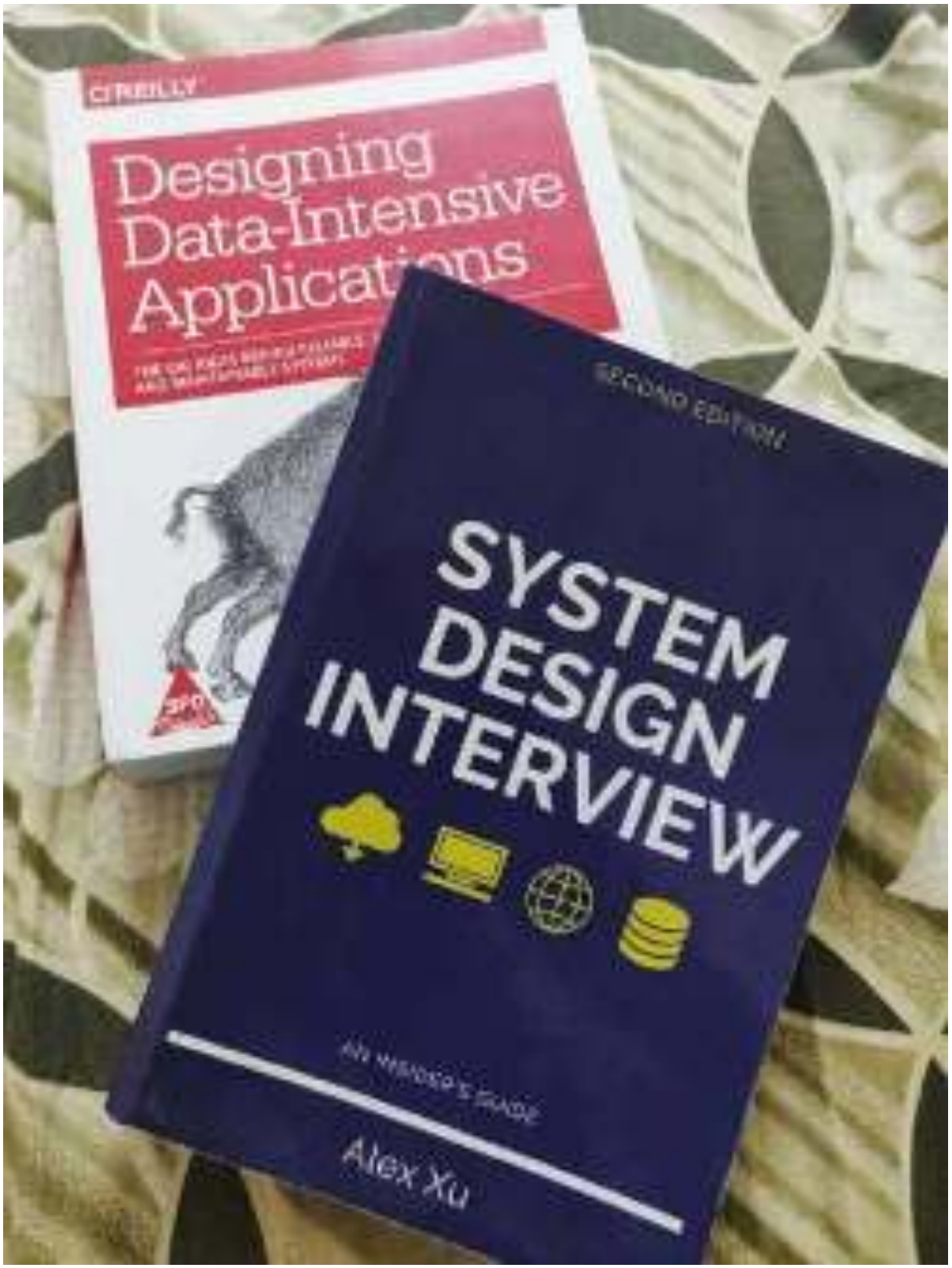

O'REILLY
Designing
Data-Intensive
Applications
SECOND EDITION
SYSTEM
DESIGN
INTERVIEW
Alex Xu

Tips for System Design Interview

Start with the Expectations, i.e. functional, non-functional requirements

1. Don't go into details immediately
   (e.g. Before, when it comes to DB design, ask if they want you to draw ER diagram right away (usually, we also want to do that))

2. Don't have a set architecture idea
   i.e. start caring about requirements and design as you've already read the question.

3. KEEP it Simple, Stupid (KISS)

4. Don't make statements without justification
   (e.g. saying you'll use a noSQL DB without any relevant statements to back the design

5. Be aware of current technologies
   (e.g. say you'll use a CDN, it might be nice to mention a real concrete like Akamai (or) cloudfront
   (or)
   for a message queue, say you'll use maybe Rabbit MQ / Kafka etc.

6) Don't do capacity estimation without any reason
   ↳ maybe speak out when interviewer tells you
   notice a particular service needs to know
   the capacity when you're 100x scale.

7) mention RPC frameworks like communication of the
   systems in architecture !! And mention pros & cons of
   using one over other (ex: HTTP vs FTP etc.)

8) Speak out internals of system (ex: why to choose
   DB like Cassandra over MySQL what does Cassandra
   bring to the architecture other DB's cannot)

9) TRADE OFFS !!

Terms & Analogy

Saturday, 25 December 2021 3:07 PM

1. Scaling → Horizontal / Vertical
2. Load Balancing
3. Backups (Avoid Single Point of failure)
4. Microservices Architecture (Isolate Responsibilities)
5. Distributed System (Partitioning)
6. Decoupling
7. Logging & Metrics
8. Caching
9. API (Application programmable interface)
10. CAP Theorem (Consistency, Availability, Partition Tolerance)
11. CDN (Content Delivery network)
12. DNS (Domain name system)
13. Ports (ex. HTTP - 80, HTTPS - 443)
14. Network Protocols (IP, HTTP, TCP)

Snippets

Friday, 14 January 2022 5:23 PM

Read heavy ?

↳ Throw a cache

write heavy ?

↳ Push write reqs to a Message Queue.

Search Data ?

↳ put a search index

More users ? Need to scale Database?

↳ need ACID properties ?

↳ Shard the RDBMS Database

↳ ACID properties not mandatory ?

↳ use NOSQL Database

High throughput ?

↳ scale application servers. & Put load balancer with Consistent hashing

Network Protocols

Wednesday, 12 January 2022 10:39 PM

↳ Just a set of rules to interact b/w 2 things.

↳ IP
↳ TCP
↳ HTTP

IP (Internet Protocol)

↳ 2^{16} bytes ≈ 0.06 MB

2 Main things in IP packet are

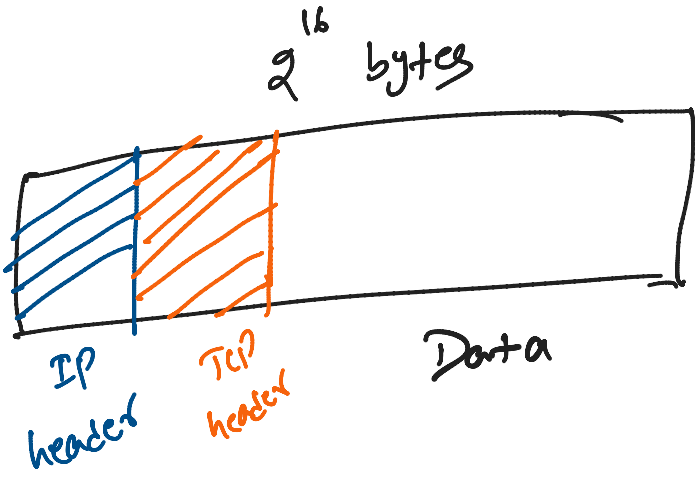
① Header (src, target, total size of packet, IP version (IPv4/IPv6 etc..))
↓
very small in general

② Data

TCP (Transmission Control Protocol)

↳ Built on top of Internet Protocol

-> Sends IP packets in order, reliable, error free way



HTTP (HyperText Transfer Protocol)

HTTP (Hyper Text Transfer Protocol)

↳ Built on top of TCP

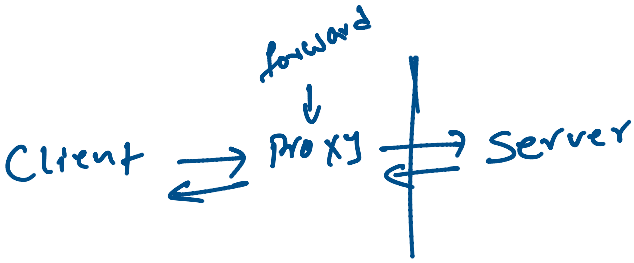
↳ This abstraction is called request-response paradigm.

Proxy (on Behalf)

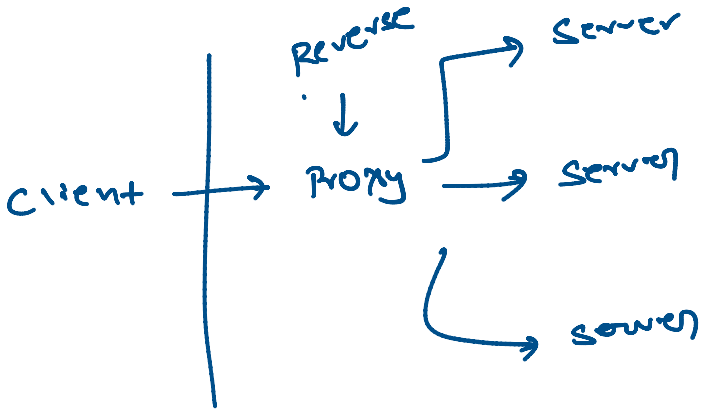
Friday, 31 December 2021 8:07 PM

Forward Proxy → Server doesn't know which client it is connecting from (VPN)

Reverse Proxy → Client doesn't know which server it is connecting to (balancer)



Forward Proxy :- Proxy sits b/w client & server and disguises client IP address, block malicious traffic from clients etc.



Reverse Proxy :- It's basically a server side proxy.

↳ can be used for traffic control, Load Balancing, caching response from servers etc...

CAP Theorem

↳ Consistency
↳ Availability
↳ Partition Tolerance

Any distributed system (distributed data store) cannot have all 3 guarantees (which is impossible), so we need to sacrifice one of them.

Consistency :- A read happening after a write should always return the latest value of the record after write operation.

Availability :- Every available node in system should respond in a non-error format to any read request, without the guarantee of returning the latest value.

Partition Tolerance :- System will be responding to all read & write requests even if the communication channel (or connections) b/w nodes is broken (or partitioned).

→ Partition tolerance mostly happens through node failures. In a distributed system, this is almost unavoidable. (i.e. we cannot give up on this property)

→ We can design a system with either

① Partition Tolerance + Availability (AP)

(or)

② Partition Tolerance + Consistency (CP)

Degrees of Consistency & Availability

→ So, we know that we cannot provide both consistency and best availability with partition tolerance in place.

→ But we can provide some amount or degree of consistency and availability!

↳ This absolutely depends on whether if we can acquire it or not!

Eg: In ATM machine we very we can be inconsistent, among so less consistent higher (or) Booking tickets system.

Eg:- 1) In ATM machine, no way we can [illegible]
have to be 100% consistent right? (or) Booking Tickets System.

2) Let's say, this time we're talking about No. of views for a youtube video. Here, you can forfeit the 100% consistency (The nodes will eventually reconcile & update anyways).

API (It's only a contract !!!)

Tuesday, 28 December 2021 7:44 PM

↳ A documented way in which external consumers know how to interact with your code (they don't know how code works, they only know how to interact with it).

Things to Remember when Designing API

1. Naming is important
2. Don't take additional parameters unless absolutely necessary. (Cost to optimise the call.)
3. Don't give more info. than caller needs
4. Error handling → Response codes (HTTP) → Respond with correct HTTP codes.
   (Don't over do it) Response messages → Don't give out sensitive info here!!!

Ex:- HTTP GET ← HTTP method

www.example.com / service / method

- where to? i.e. Route.
- Typically a micro service
- The functionality etc...

Content Delivery n/w

Wednesday, 29 December 2021 8:47 PM

↳ example Akamai, S3

CDN does: (Primarily focuses insanely fast data delivery by having multiple data centers geographically distributed !!!)

1. Host the data close to users
2. Follow Regulations
3. Provides some interface to update/remove? data from the data centers ?
   ↳ The CDN basically ?
4. can configure how much time to store some data in this CDN.

Cloudfront

↳ super cheap

↳ Reliable & easy to use

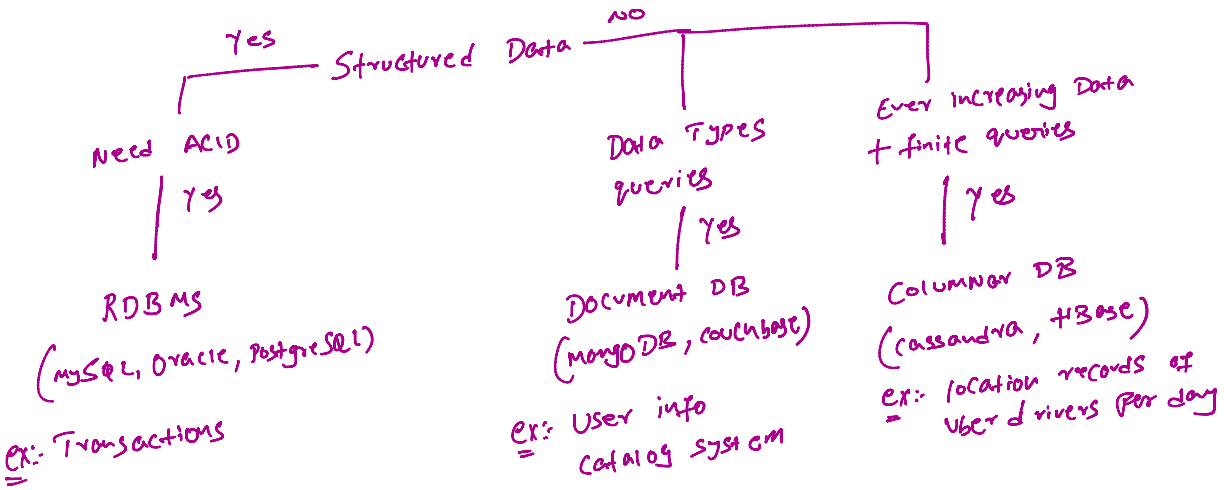
SQL vs NoSQL

Monday, 24 January 2022 1:35 AM

① Always think of caching to reduce latency.
- ↳ the server API call cache
- ↳ *Redis, MemCached (Key value stores) → non relational DBs

② Storing Images, Videos? These are blob storages & not queried upon.
- ↳ So use Amazon S3 to store
- ↳ use a CDN infront of S3 to distribute same data geographically! (faster way)

③ Text searching Feature (ex: Amazon product entire search / Netflix show title search etc..)
- ↳ ElasticSearch / Solr



-> Feel free to use a combination of database types based on the scenario!!!

① Relational Databases
- ↳ Tables & rows with a fixed schema
- ↳ ACID properties
- ↳ usecase → Entirely structured data / need acid properties
  - ex: Employee Table, Transactions etc..
- ↳ vertical scaling is easy (add more RAM, CPU etc)
- ↳ horizontal scaling is tough here (i.e. sharding)

Databases (Horizontal Partitioning inbuilt mostly)
↳ Built for scale

② Non relational Databases (Horizontal Partitioning inbuilt mostly)

↳ Key-value store (Redis, Memcache)
↳ fast & provide quick access & they are in memory
↳ used in caching solutions

↳ Document Based (mongoDB)
↳ used when not sure about structure of data & would evolve over time (i.e. No fixed schema, and can be changed easily)
↳ supports heavy reads & writes, provides sharding capabilities
↳ They have collections (like Tables) and documents (like rows)
↳ usecase → Product details of Amazon (since we want all data of one Product at one place, no joins etc..)

↳ Cons
↳ NO ACID Properties though we can handle it through application code, main loss is strong consistency.
↳ Joins are tough to achieve.

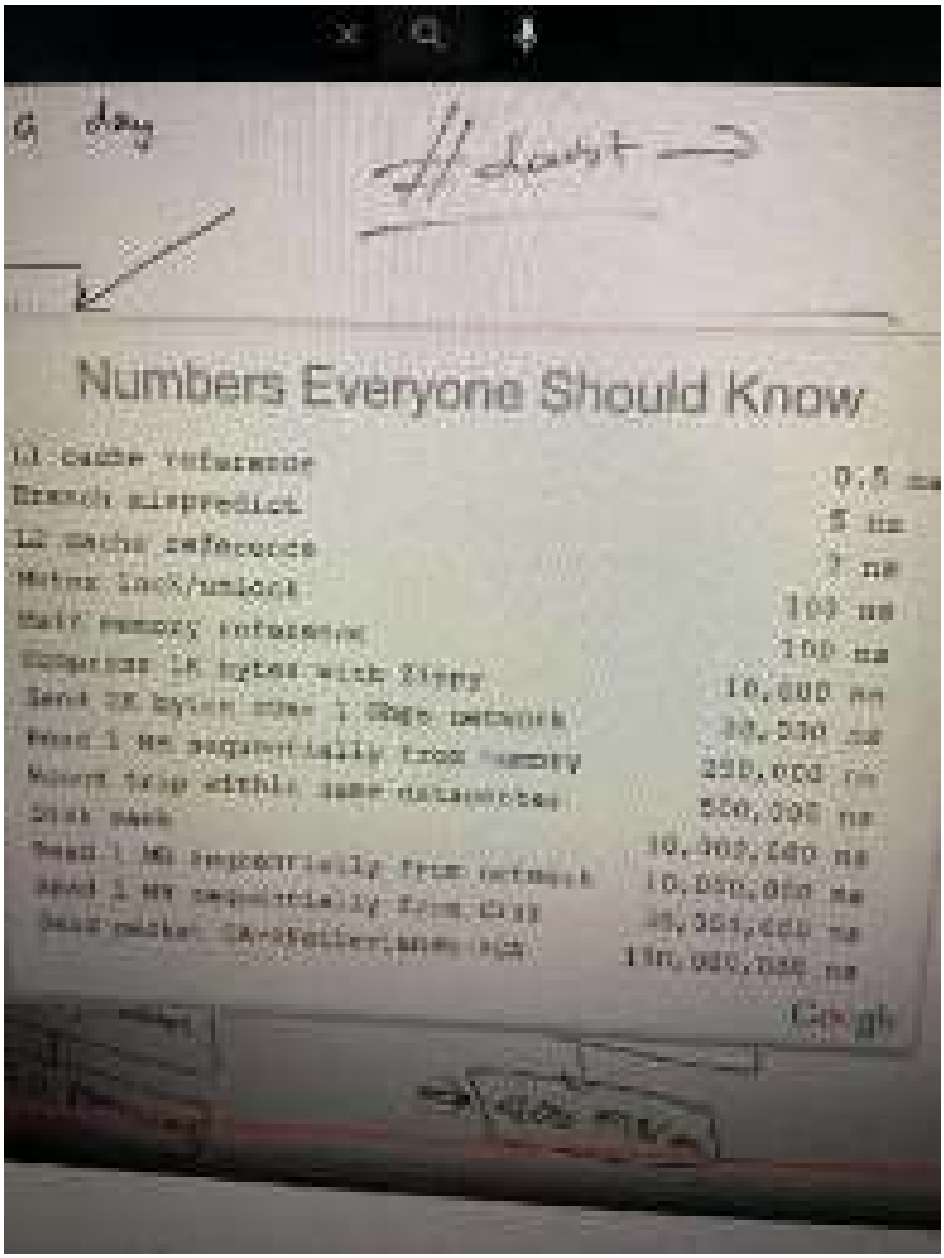
↳ Column DB (Cassandra, HBase, Scylla)
↳ Sort of mid way of relational & Document DBs
↳ Doesn't support ACID
↳ usecase → Streaming data / event data for analytics.
↳ Distributed databases in general

↳ Search Databases (Elasticsearch, Solr)
↳ Full text Search
↳ Fuzzy Search
↳ Keep in Mind, this is NOT PRIMARY DATABASE!

↳ Built for scale
↳ Built for aggregations

For Reference

Tuesday, 28 December 2021 9:46 PM



How to avoid Cascading failures

Friday, 31 December 2021 1:47 PM

Problems

1. Cascading failures → Rate limiting
2. Going viral → Pre scale
3. Predictable load increase → Auto scale
4. Bulk job scheduling → Batch processing
5. Popular posts → Approximate statistics.

Database Sharding

Problem → How to query huge data in DB?

1. Optimise queries (but we of data, even with optimised queries, it's not good enough)

2. Shard on table (fine, but got tons of data!!)

3. Sharding

→ Horizontal Partitioning

"Database Servers"

Takes a key and partitions the data to different servers

(The key is generally an attribute of data we're storing)

→ Practice of separating one table's rows into multiple different tables, known as partitions → sharding

Problems

1. Joins across multiple shards
2. Fixed no. of shards

↳ use hierarchical sharding

i.e. Let's say even after sharding the data in one shard is huge.

. . . in this shard we can again . . . in multiple

one shard

→ Now, in this shard we can again shard / partition the big shard into multiple shards.

② what happens if a shard goes down

↳ use master / slave architecture

↳ master is the most updated one & slaves keep reading from master for updates

↳ All writes go to master, Reads are distributed to slaves

↳ If master fails, one slave becomes master.

Distributed Caching

Tuesday 28 December 2021

→ Querying DB for commonly used data !!

↳ use cache! (In memory cache)

Helps in saving n/w calls

→ Avoid computations again & again

↳ instead of computing it every time from querying DB, query once & save in cache.

→ This also avoids load on DB

Caches generally run on SSD (Solid State Drive) which are expensive!!

① when to load data into cache
② when to evict/remove data from cache

} cache policy!

→ cache policy determines how cache performs !!

↳ ① LRU (Least Recently used) → mostly used !!
② LFU (Least frequently used) → not much used

Thrashing

↳ constantly loading & removing data from cache without ever using the results!

Problem

→ How can data be consistent in cache?

i.e. service A updating info in DB which is cached in service B cache, would what??

① Write Back Cache

↳ When updating data, hit cache (if the data exists then update it here in cache, then update DB)

↓ This method will have some data inconsistency if multiple caching are present

→ This should only be used if data update in cache is not of high importance.

→ Then all updating on cache can at once be sent as a batch to DB for update.

② Write Through Cache

↳ First hit the DB & then update in cache (synchronous)

↳ Reliable & consistent (but slow as update needs to happen in both places)

Consistent Hashing

Load Balancing

In this we have "n" servers

and we want to balance load across all

those "n" servers in our System

→ Consistent Hashing will be the load balancing algorithm

Example: hash(key) % n (Assume we have 3 servers and

↓ each key is assigned to)

hash function is uniformly distributed

so, data will be spread across

servers as well (load balancing)

Add one server will have uniform load

& Requests to load will be spread across

all servers

Problem?

When adding new servers (scale up/down)

the majority of the mapping gets

→ i.e. In most cases, the request is going

to different servers even though

they were cached

→ The only way to maintain cache is "n"

because of definition of which servers cache

what. Which is the same place servers "n"

(same hash is a function which servers)

Why? → Since, every time we change the

no. of servers

most of the values of hash % servers

also change so cache

Solution (use Consistent Hashing)

→ use [0, n] range

Solution (use consistent Hashing)

① Let's say our hash function outputs in $[0, M-1]$ range (Total)

② Servers themselves have Id's also, we hash these id's also and take remainder with M.

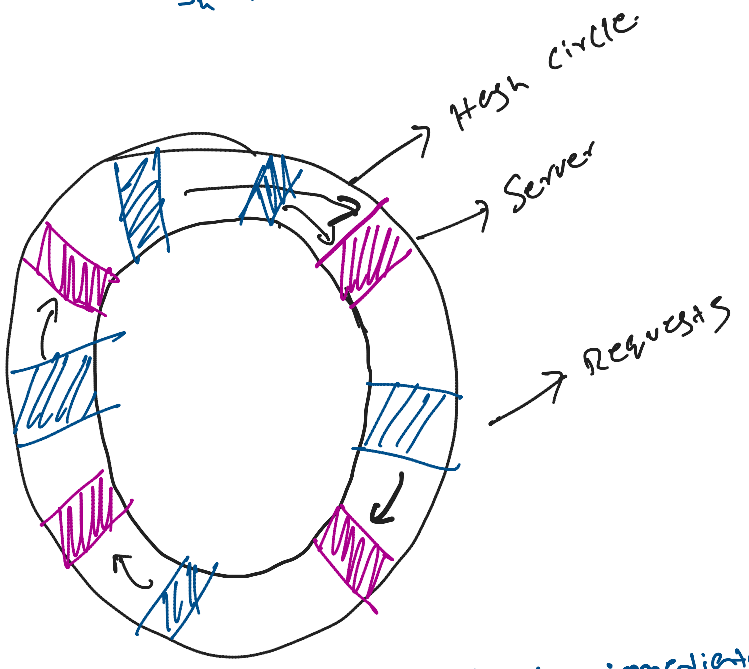
↳ i.e. $S_0 \rightarrow h(0) \% M$

$S_1 \rightarrow h(1) \% M$

$S_2 \rightarrow h(2) \% M$

$S_3 \rightarrow h(3) \% M$

$S_4 \rightarrow h(4) \% M$



- The request is served by the immediate server on the clockwise direction.
- If a newServer is added, it goes into the hash ring.
- If a server goes down, all requests that would go to this server will now be served by the next server on clockwise direction.

Problem

- If there are 5 requests that have Server "A" on clockwise direction. If this server "A" goes down,

- If the server "A" goes down in clockwise direction. If this server "A" goes down then all these 5 requests go to next server in clockwise direction, let's say server "B".
- Due to this server "B" load increased a lot!!!

Solution:-

use multiple hash functions for servers

i.e. $S_0 \rightarrow h_1(0)$ $h_2(0) \rightarrow ..$

$S_1 \rightarrow h_1(1)$ $h_2(0) \rightarrow ..$

- If we have "K" hash functions, we can have K points for each server.
- Likelihood of one server getting lot of load is now very less
- If a server is removed, then remove the K points of that server.

Message Queue

Sunday, 26 December 2021 3:34 PM

> Takes tasks, persists the tasks, assign tasks to right servers, waits for task completion, if taking long time/server dead assigns to another server

All of these are done By a "Message Queue"

Ex:- RabbitMQ, JMS (Java Messaging Service)

Horizontal vs Vertical Scaling

Saturday, 25 December 2021 2:00 PM

→ Most Services on Cloud (AWS/GCP/Azure)

↓

Can be Java Process/Service/anything

Problem → How to handle more requests
i.e. Scalability

① Buy bigger machine → vertical Scaling

② Buy more machines → Horizontal Scaling

Trade offs

Horizontal	vertical
① need load balancing	① n/A
② Resilient	② Single point failure
③ more calls (RPC) between servers	③ Limited process (hardware)
④ Data (in)Sistency issue	④ consistent
⑤ Scales well	⑤ Redundant limit

Solution

Horizontal scaling but each server is powerful enough