System Design

4 components required to power a system.
architecture
modules

Set of Requirements

- Decide Architecture
- Decide components
- decide modules

How they interact with each other
↓
"To solve a problem"

"PRODUCT DEVELOPEMENT".

What will we do when we design a system?

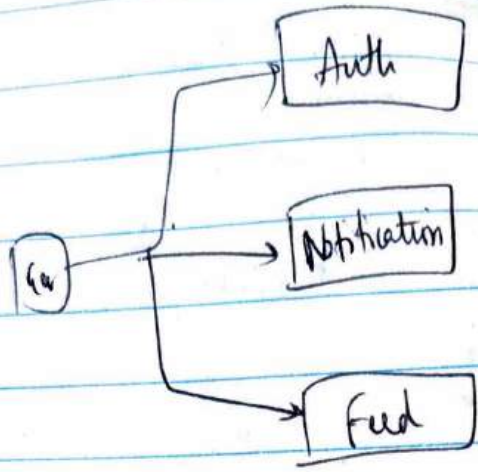
- Break down problem statement into solvable sub-problems
- Decide on key components & responsibilities
- Decide on boundaries
- key challenges in scaling
- Make our architecture fault tolerant & availability.

How to approach system design?

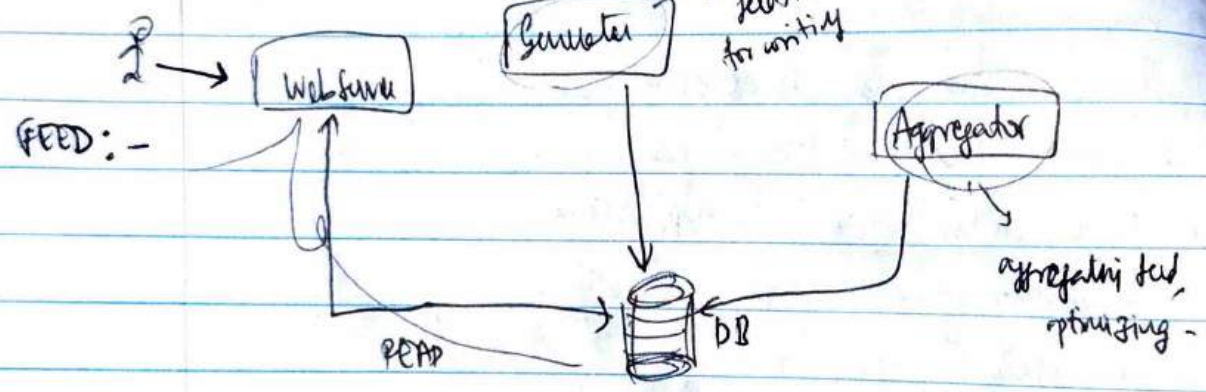
- Take Baby steps after understanding the problem statement
- Break the problem statement down into components (essential)

  Note: Create components that are a must and do not create for the sake of it.
- Design Facebook

  Auth, Notification, feed, gamification 4 components/features



Now Dissect each component



For each sub-component look into

1. Database & caching
2. Scaling & Fault tolerance
3. Async processing (delegation)
4. Communication (how diff component communicate)

Repeat for each subcomponent one by one

where to place them etc?

Add more subcomponents

Try splitting the components into sub-components.

- ~~Big Event problem~~

- Specific problem (small Details)
  ↳ ~~Down~~ Bottom Up.
  (Database first)

Big problem → Top Down
↓
Features (Code first)

4. "How to evaluate that Built system is a good system?"

Every System is infinitely buildable.

- Some pointers to help:-

1. Break system into components
2. Each component should have clear set of responsibilities

3. → Feed webServer → serves feed over HTTP.
   → Feed generator → pulls data from multiple services (posts, friends, recommendations) & puts them in DB.
   → Feed Aggregator → combines candidate items fetch be generator filters out redundant, ranks, and creates a consumable feed.

For each component, we should figure out technical insights.

Each component (in isolation) is :-

- Scalable - Horizontal scaling
- fault tolerant → Recovery in case of failure
- available → component working even in on server / DB goes down..

"This is how we would tackle every single system structured and detailed

Relational Database

- Most brittle, most response time, most latency (or) anything that breaks is always because of the DB.
- Data is stored & represented in rows & columns.

History of RDBMS

Databases were developed to support accounting.

Hence, key properties were:-

1. Data consistency
2. Data durability
3. Data integrity
4. Constraints
5. Everything in one place.

Because of these reasons, relational database provides "Transaction"!

A C I D

A → Atomicity
C → Consistency
I → Isolation
D → Durability.

...ure / DB goes down...

...icity :- All statements within a transaction take effect (or) none

Eg :- No matter how many statements are run, either all are executed (or) none.

Eg :-
```
start transaction
insert into posts values (..);
update stats set total_post = total_posts + 1
where user_id = 100;
commit
```

publish a post and increase total post count

→ can also be created using triggers.

DB remains consistent before and after the transaction.

- Consistency :- Data will never go incorrect, no matter what constraints, cascades, triggers.

Eg :- Foreign keys checks do not allow you to delete parent if child exists

Cascades suggests if we delete a row then we have to delete everything related to it.

triggers → On update trigger this etc.

Durability

when transaction commits, the change outlives outages.

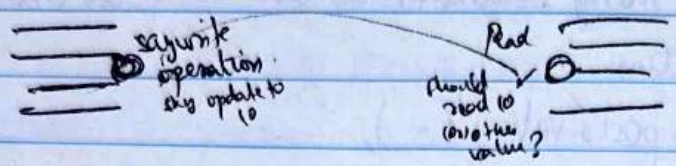
"The change of a successful transaction occurs even if the system failure occurs".

Isolation

when multiple transaction are executing parallely,

the isolation level determines how much changes of one transaction are visible to other.

Txn 1 Txn 2



Should changes done at this line in Txn1 be visible to Txn2 before Txn1 commits?

Remember

You pick relational databases for relations and acid properties.

4 types of isolation levels :-

(i) Read uncommitted
(ii) Read committed
(iii) Repeatable Read
(iv) Serializable

6. Database Isolation levels

- Isolation level dictates how much one transaction knows about the other

→ Repeatable Reads :-

consistent reads within same transaction
even if other transaction committed 1st transaction would not see the changes (by value already read).

Gives guarantee that we see the same value.

Transaction _ISOLATION

SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

Read Committed

- Reads within the same transaction always reads fresh value
  con:- multiple reads within same transaction inconsistent.

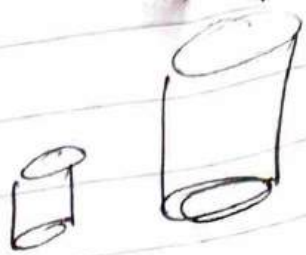
Read Uncommitted

- Reads even uncommitted values from other Txns.
- "Dirty Read" Problem.
  "Has a better throughput"

Serializable (strictest & slowest)

- Every read is a locking read
  and while one txn reads, others will have to wait
- Only after 1 txn commits, the other txn gets to read.

Scaling Databases

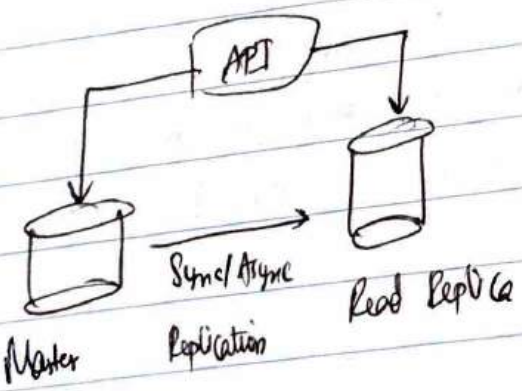


Vertical scaling

- Add more CPU, RAM, DISK to the DB
- Requires downtime during reboot.
- Vertical scaling has a physical hardware limitation

Horizontal Scaling : Read Replicas

- when read : write = 90 : 10
- So you move reads to other database so that master is free to do writes.

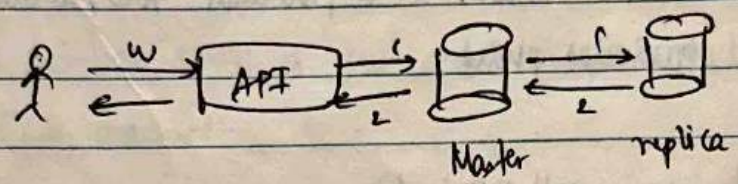


- How to know reads to navigate to Read replicas and write to the Master.
  Do this on the API level.

There are 2 types of replication
SYNC and ASYNC replication.

- Replication → Changes on DB(master) needs to be sent to replica to maintain consistency

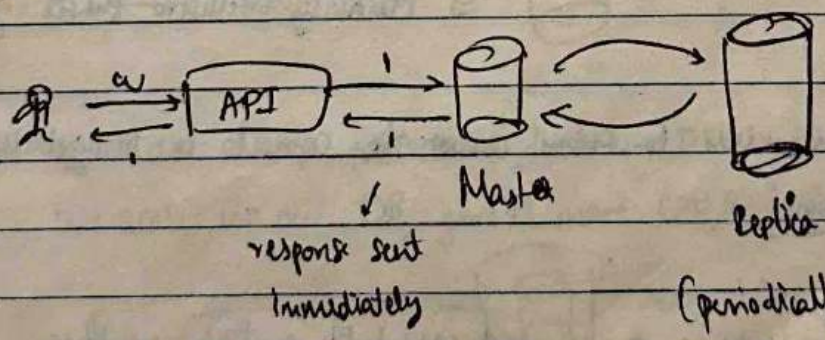
1. <u>Synchronous replication</u>



Written on both the DB's. Slow as latency is increased.

- Strong consistency
- zero replication lag.
- Slower writes.

2. <u>Asynchronous replication</u> (V. common)



- eventual consistency
- some replication lag
- faster writes

* We can add many read replicas.

*

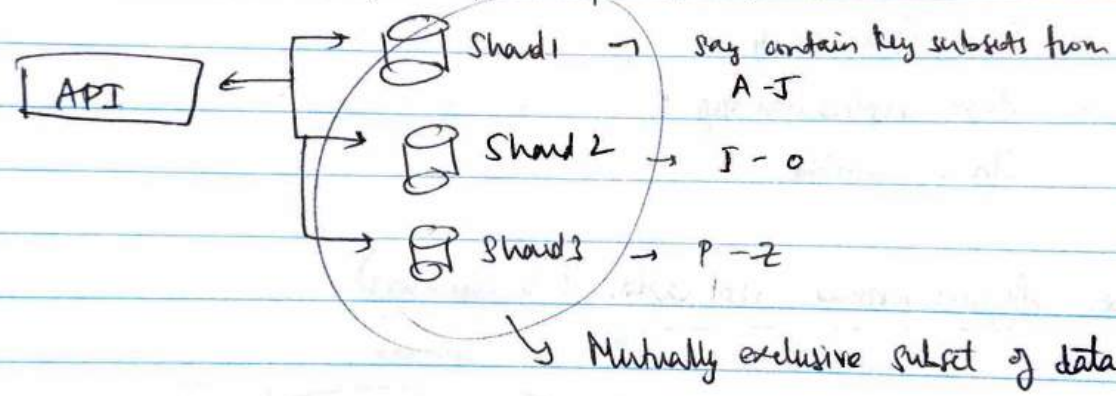
Horizontal Scaling: Sharding

What if we want to do more writes?

Because one node cannot handle the write loads. We split them into multiple exclusive subsets. Writes on a particular row/document will go to one particular shard.

This way we scale our overall DB load:-

Note:- Shards are independent, no replication b/w them.



API server needs to know whom to connect to, to get things done "Routing logic".

Note:- Some DB's have a proxy that ~~cate~~ takes care of routing.

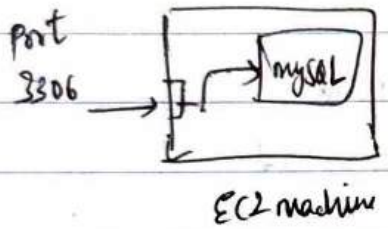
* Each shard can have its own replica (if needed).

Sharding and Partioning

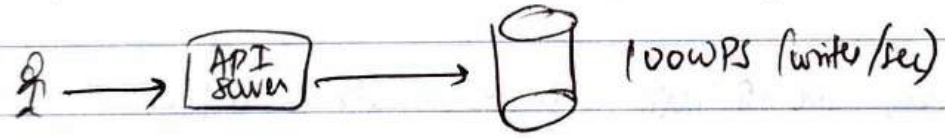
- Sharding :- Method of distributing data across multiple machines
- Partioning : Splitting a subset of data within the same instance

How a database is scaled?

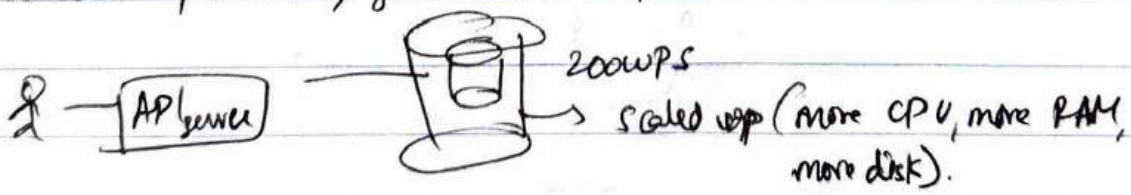
A database server is just a DB process (mysql, mongodb) running on an EC2 machine.



Say,
You put your database in production, serving real traffic



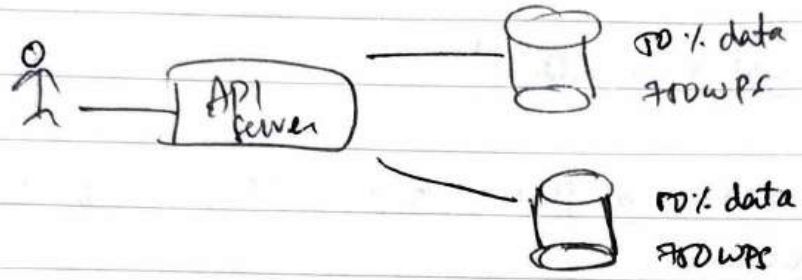
Now, we are getting more users, that our DB is unable to manage
We scale up our DB, give it more CPU, RAM & Disk



Now, our product went viral y 1000 WPS, so unable to handle the load. So we scaled up again.

After certain stage we cannot "scale up" DB. so ve[illegible] a limit.

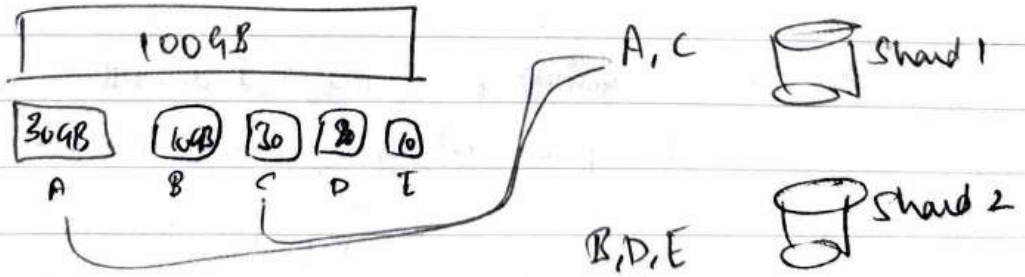
So we have to resort to Horizontal scaling for 1000WPS



So Each DB server is thus a shard.
and we say that the data is partitioned.

"Overall the DB is sharded while the data is partioned"
↑
split across.

Say 100GB data → partioned into 5 exclusive partitions.

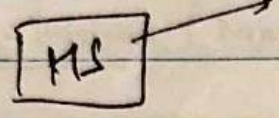


DB - do Vertical scaling for 1000PS

w to partition the Data?

Horizontal partioning (v. common)

Vertical Partitioning



pick rows and place in DB's
(taking bunch of rows & placing)
(slicing it horizontally)

Some tables in 1DB & other tables in other DB → Vertical scaling.
↳ Monolith to Microservices - most commonly used.

Partioning

Sharding	NO	YES
NO	(Monolith)	
YES		

within 1DB creating 2 databases.
say CREATE AUTHENTICATION & CREATE PROFILE in same DB server

Read Replica

Advantages of Sharding

(i) Handle large Reads & Writes
(ii) Increased overall storage capacity
(iii) Higher availability

Disadvantages of Sharding

(i) Operationally complex
(ii) ~~complex~~ cross-shard queries expensive.

Hence we need to try to minimize this as it reduces performance.

Non-Relational Databases

what makes NRD interesting?
Most NRD shard out of the box!!
↳ Horizontal scaling

We talk about 3 most important types of NoSQL databases

- Document DBs (closest to RD)

1) MongoDB
2) Elastic Search

- Mostly JSON based
- Supports complex queries → almost like Relational (sql) databases
- Partial updates to documents possible
- closest to relational databases

Use Cases = In-app notification service, catalog service.

{
"user_id":
"total_posts": 270
}

can do total_posts += without re-writing the entire document.

- Key-Value Stores (Redis, DynamoDB, Aerospike)
- extremely simple databases
- limited functionalities (GET, PUT, DEL)
- Meant for Key-based access pattern
- Does not support complex queries (aggregations)
- Can be heavily sharded and partioned
- Use Cases: Profile data, order data, auth data

- You can you use relational databases and document DBs as KV stores.

Graph Databases (Neo4j, Neptune, Dgraph)

- What if our graph D.S had a database.
- It stores data that are represented as nodes, edges and relations

Eg: A —— B
follows

- Great for running complex graph algorithms
- Powerful to model social networks, recommendations and fraud detection.

10. Picking the right Database?

- A DB is designed to solve a particular problem really well
- Each kind of DB picks a segment with a slight overlap

Common Misconception :- Picking NRD because RD do not scale.

Why NRD Scale?

- There are no relations or constraints.
- Data is modelled to be sharded.
  ↳ split across multiple nodes.

If we relax the above on RDB, we can scale it too !!

- do not use foreign key checks
- do not use cross shard transaction.

Eg:- How does this help in designing system,

1. Understand what data you are storing
2. Understand how much of data you will be storing.
3. Understand how you will be accessing the data.
4. What kind of queries you will be firing.
5. any special feature you expect
   Eg:- Expiration

- Redis → cache & advanced data structure
-

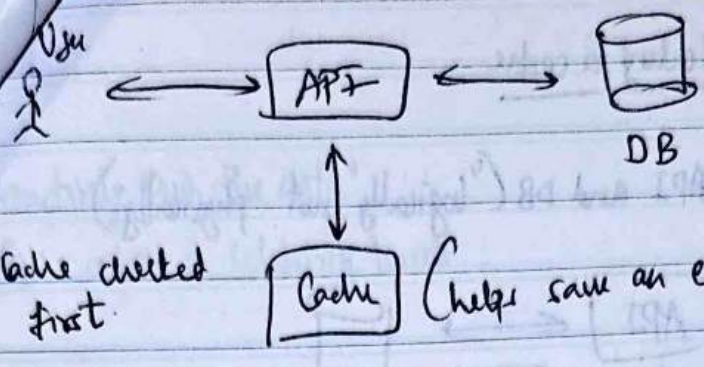
11. What is Caching?

Caches are the place where we store data - Caches helps you avoid an expensive n/w I/O, disk I/O, CPU I/O or computation.

- API call to get profile information
- Reading a specific line from a file
- doing multiple table joins

* Performance improvement

Store frequently accessed data in a temp. storage location.



* Cache checked first.

(helps save an expensive I/O operation)

* Caches are faster and expensive
- Hence we do not cache all the data (just a subset of it which is most likely to be accessed).

2 famous caches - Redis, Memcache

- Caches are not restricted to RAM based storage
  Only Redis stores in RAM.

"Any storage that is nearer and helps you avoid something expensive" ↳ cache.

- Caches are glorified hash tables (key-value) pairs.

<u>Some Examples</u> :-

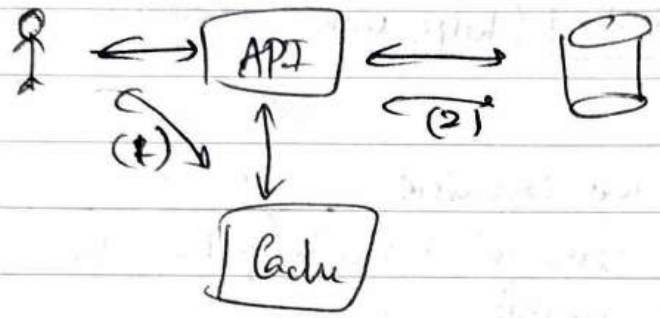
(i) Google News :- Most recent news articles are to be accessed, hence saved from cache.

(ii) Auth Tokens :- Authentications are cached in "cache" to avoid load on DB.

(iii) Live Stream : last 10min of live stream is cached as it will be accessed the most.

12. Populating and Scaling a cache

Cache sits b/w API and DB ("logically" not physically)



1. Lazy population (most popular)
Read first then go to the cache if data exists, return data else go to DB, do heavy operation, persist in the cache, return data..
Eg:- Caching Blogs
Fetching a blog from DB is expensive (multiple join.)
Hence, when someone accesses it we fetch from DB and cache it on Redis.
Subsequent requests are served from cache.
"JSON ~~when obtained~~ is stored ~~as a~~ in cache."

* Whenever we set something in cache, we set an expiry.

If we don't set expiry then it leads to memory leak.

2. Eager Population (proactively trying to push in the cache)
Writes go to both database and cache in the same request call.
Eg:- Live cricket score, 1000's of users reading cric commentary
You will be serving it from cache. So why not update cache & db at once and save the cache miss.

Proactively pull the data

Eg:-

2. Proactively Push the data
Eg:- when a celebrity tweets.

Scaling a cache
Cache is similar to a DB, so scaling is also similar to db.

1. Vertical Scaling
2. Horizontal scaling - Read Replica

3. Horizontal scaling - sharding (Distributed cache)

13- Caching at Different Levels

Most common cache we saw was Redis. (store data in Memory) But that is not the only type of cache out there or not the only place that can be used as cache.

Literally every piece/component in your infrastructure can cache something for you. But should you?
↓
Depends

topic has n-partitions.

Too much caching is bad → stale data and invalidation

(Most accurate / consistent data takes time to come to cache).

Eg:- Banking Transaction.

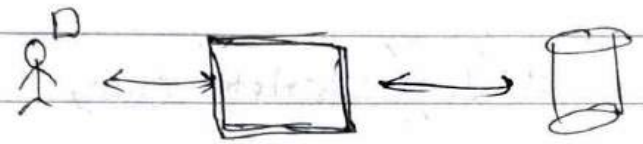
Financial applications always require consistent data.

→ 2 different places where we can cache:-

- Client side caching

Storing freq. accessed data on client side

Eg:- Browser, mobile devices etc.



cache near constant data. (Eg:- Images, js files, User info etc.)

It should be okay serving cache (stale)

invalidation by time (expiry)

Eg:- Amazon prime caches recently watched in the browser side.

- Content Delivery Network (CDN's) → live streaming, images & videos

set of servers distributed across the world. → used for caching.

Request from a user goes to the nearest CDN server and hence user gets a quick response.

CDN does lazy cache population

"Origin Server"

If CDN doesn't have it takes from origin server

CDN → always a lazy population.

user's request comes to CDN → closest server
CDN server checks if it has the data. If yes, return the data.
else,
CDN makes the same request.

Eg of CDN → Cloudflare, Akamai

Remote Cache (Redis) → expensive and stored in memory
→ It is not distributed

It is a centralized cache that we most commonly use. Multiple API servers use it to store freq. accessed data.

- Every key stored should have an expiration (memory leak)

Database caching
Instead of computing total posts by users everytime.
we store total_pos as column & update in once a while.

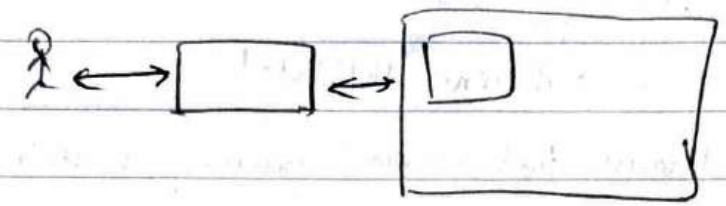
So we can save a expensive query soln in an separate column in same dB.

There are other places like load Balancer where we can cache.
Note :- we can cache some data at every single component in the system.

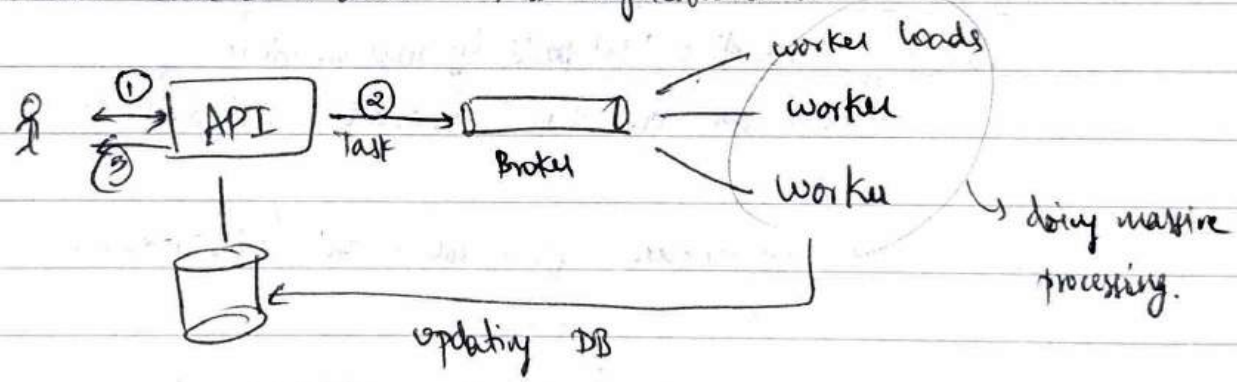
Asynchronous processing

User sent the request and you immediately handle it is synchronous
loading insta feed is synchronous. } Most instruction on web are
Login on website is synchronous. } synchronous

But somethings should not be synchronous.



Spinning up a VM takes min, and user will not wait on the same page, waiting for the response.
Instead he/she would low to move around and keep checking status once in a while. This is asynchronous.

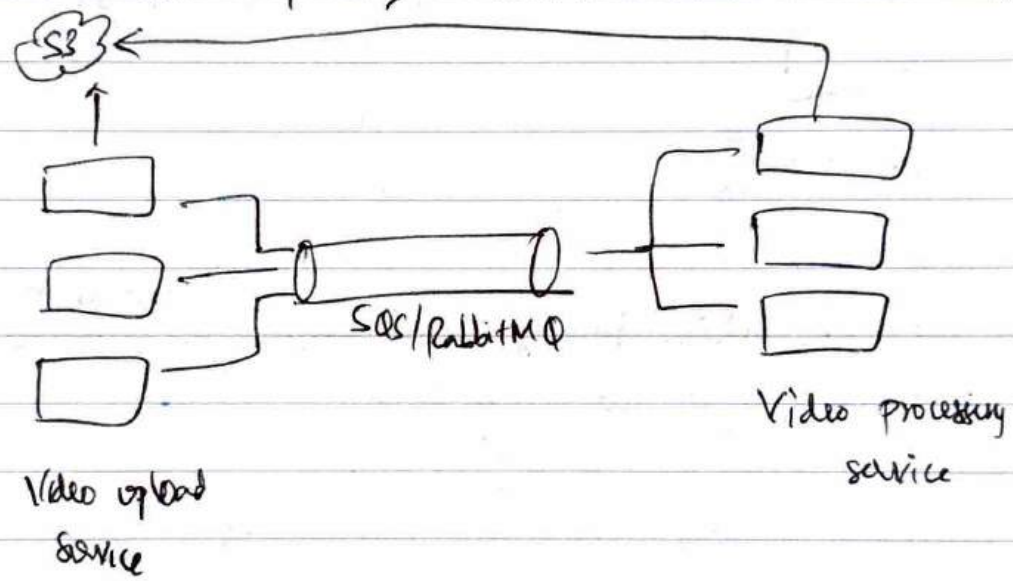


Message Queues

Brokers help 2 services/applications communicate through messages
We can use message brokers when we want to do something asynchronously:-

1. Long-running tasks
2. Trigger dependent tasks across machines.

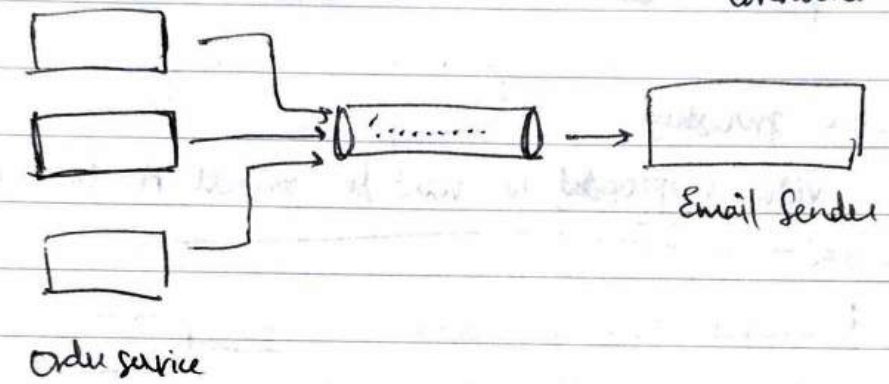
Eg:- Video processing.
Once the video is uploaded, we need to convert it to 360p, 480p, 720p.



Message queues are called message broker.

- Features of Message Broker

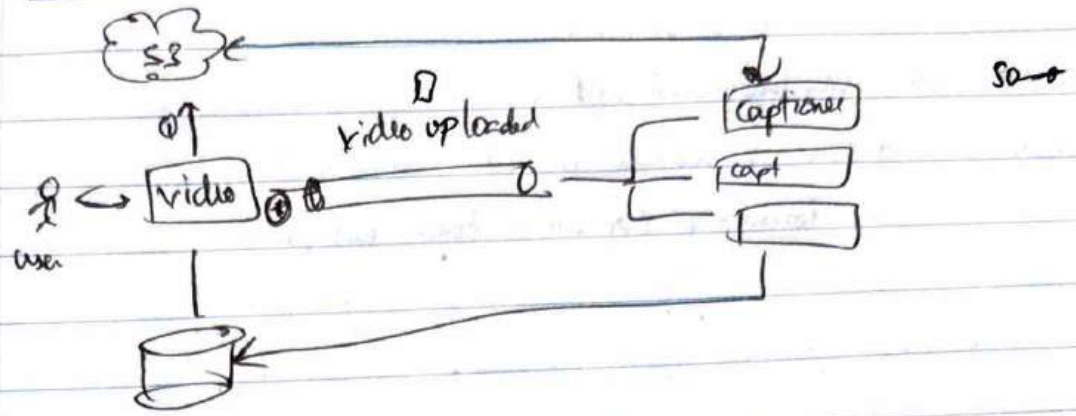
1. Brokers help us connect different sub-systems.
2. Brokers act as a buffer for the messages.
   i.e consumers can consume at their own pace.
   Eg:- Notification system → no synchronous load on connected systems.



1. Brokers can retain messages for 'n' days → depends on broker you use
   Eg:- SQS can hold msg for 14 days.
2. Brokers can requeue the message if not deleted.
   Eg:- consumer read the message but before it could delete it, it crashed.
   Broker calls API to delete a message'.

- Some messages may be delivered to you more than once → side effect

Example : Auto-subtitle (auto-captioning)



1. user uploads video to S3 through video service
2. video service puts a message (after upload complete) to broker
3. and returns response to the user
   user sees upload complete
4. message is asynchronously read by captioner.
5. captioner downloads the video.
6. captioner generates captions and updates in the DB.
7. User now can see subtitle / caption button enabled.

* Message queue & message streams (both are different).

topic has n-partitions.

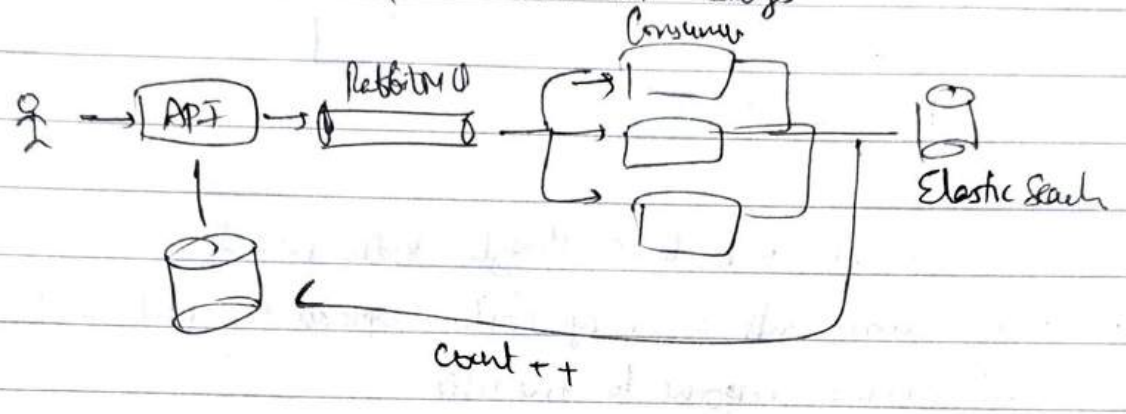
Message Streams

We are building Medium.com

and upon every blog published, use

- need to index it in search engine (Elastic search)
- need to do count++ for user's total blogs

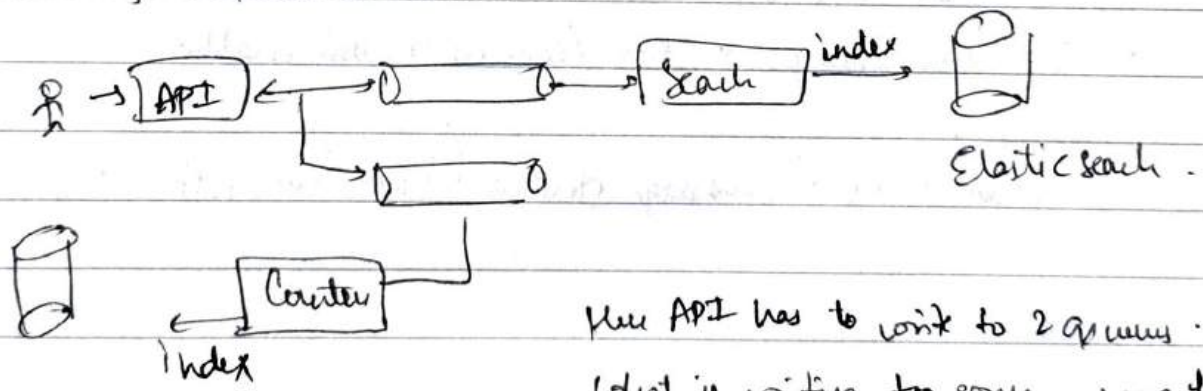
Approach 1



Consumers are doing 2 things → count++ and index

Issue: What if write to one succeeds but other one failed.

What if 2 queues are used now



Here API has to write to 2 queues.

What if writing to some queue fails.

we end up in the same spot.

~~Multiple~~ Root cause is we do not want any multiple things.

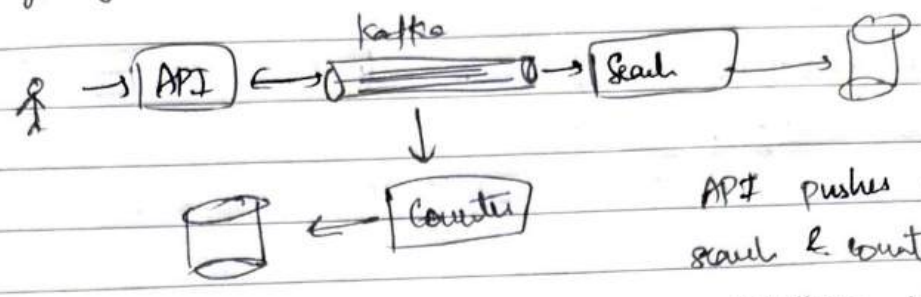
Hence, we want to "write to one" and "read by many" semantics.

This is where message streams come into picture.

↳ Kafka, Kinesis.

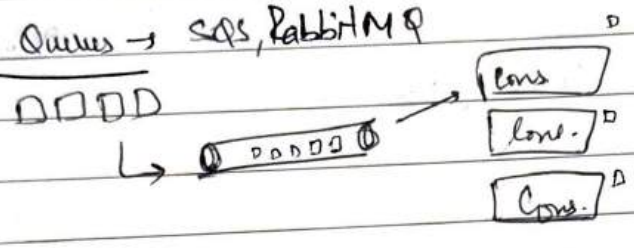
In Message queue consumers are homogenous, all consumers do the exact same thing

Message streams are similar to message brokers with one change: multiple type of consumers read the same message.

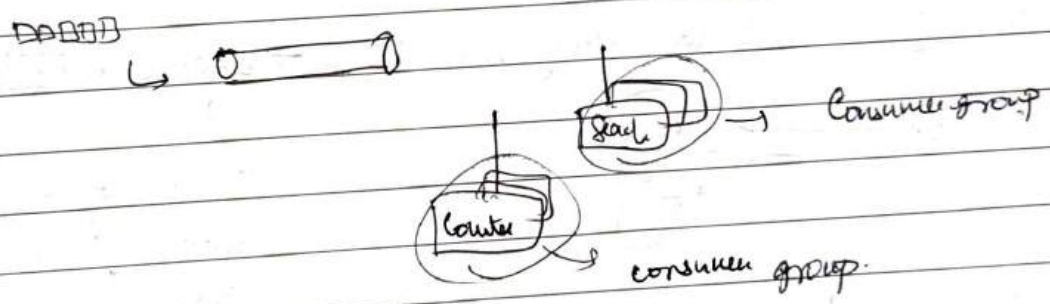


API pushes one message in Kafka, search & counter service both reads does their work

Message Queues → SQS, RabbitMQ



Message Streams
Kafka, Kinesis

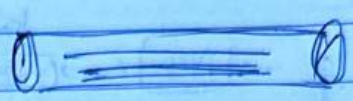


Kafka → messages lies forever
Here, each consumer iterates and consumes each message

Kafka Essentials

Kafka is a message that holds the message
Internally Kafka has topic
Every topic has n-partitions.

Hash based partitioning
Message is sent to a topic and depending on the configured hash key it is put into partition.
Within partition, messages are ordered
↳ no ordering guarantees across partitions

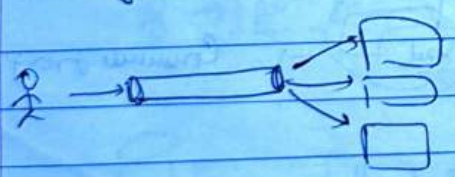


Limitation of Kafka
consumers = # partitions, If only 1 consumer is present then it can consume from any partition.

Parallelism in kafka is limited to no. of partitions.

16. Real-time pubsub
Message brokers and message streams requires consumers to pull.
Advantage :- Consumers can pull at their own pace. Consumers do not get overwhelmed.
Disadvantage :- Consumption lag when high ingestion.

What if we want low latency and 0 lag? → Realtime pubsub
↓



Instead of consumers pulling the message, the message is pushed to them

Eg :- Redis Pub-Sub

Still really fast really time.
It broadcasts to all consumers and the only consumers where it is required consumes.
Especially messaging apps use this

Say some update to be pushed, then we push to all the server/consumers.

Real-time pub-sub do not store anything.

Load Balancers

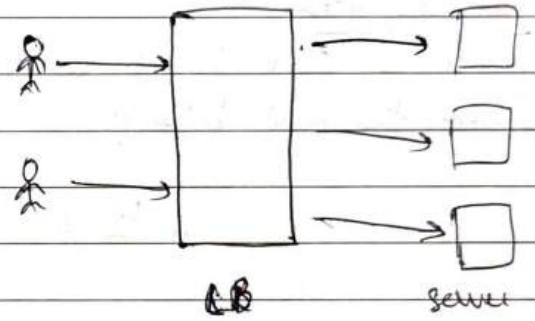
One of the most Infra component of D.S which makes it easy to scale horizontally.

* LB is the only P.O.C

Every LB has other

1. Static IP
2. Static DNS names

} allowings clients to talk to it (users / servers)



Request / Response flow

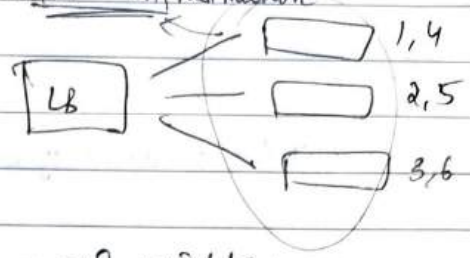
1. Client already have ip/domain of load balancer
   Eg :- auth.example.com
2. Client makes API call and it comes to load Balancer
   Eg - GET auth.example.com/login
3. LB picks one server and makes the same req/
4. LB gets the response from the server
5. LB respons back to the client

Load Balancing Algorithm

Picking one server to forward to

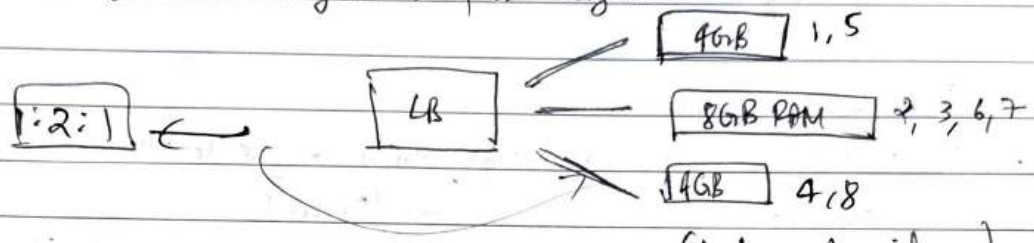
1. Round Robin

Distribute the load internally - Uniform Infrastructure



2. Weighted Round Robin

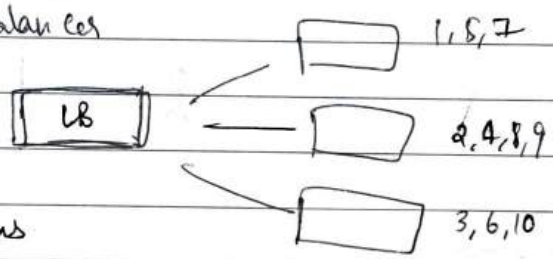
Distribute the load iteratively but as per weights.



(In this not uniform)

3. Least connections

Pick the server having the least connection from the load balancer



Degree of freeness ∝ no: of connections

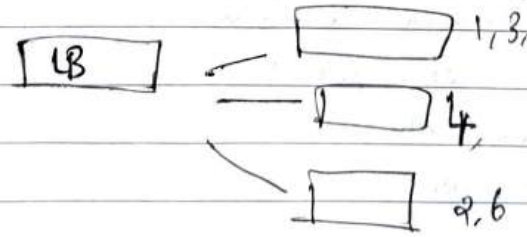
Minimizing response time / load

~~low~~ response times

↳ when we have ~~request~~ high variance

4. Hash based Routing → random enough

Hash of some attribute (ip, user_id, url) determines which server to pick.

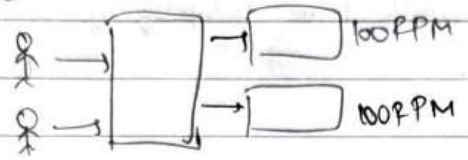


and % no: of servers

Advantage → Everytime same user will come to the specific server no matter what so sticky sessions are enabled in hash-based routing.

Key Advantages of LB → Abstracting the distributedness of the system.

1. Scalability :- with more servers behind LB, more request can be handled



we can handle 200 RPM

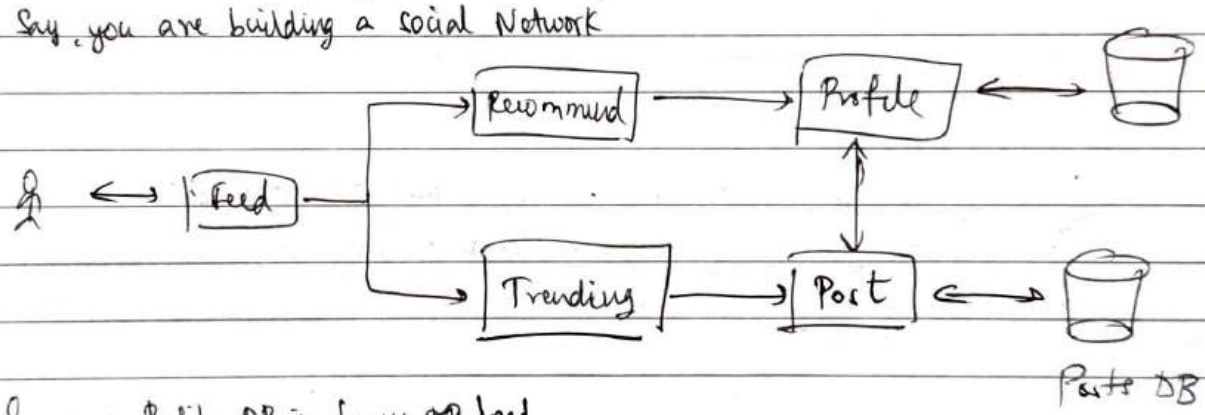
2. Availability :- Even if one of the servers crash, it does not take down the entire system. LB will forward to other healthy servers. Improving the availability.

8. CIRCUIT BREAKERS

They prevent cascade failures

V. Imp which we design for resilliency.

Say, you are building a social Network



- Say our Profile DB in heavy ~~DB~~ load.
- Then it starts getting slow and transitively it leads to a big outage when 1 service is dependent on others.

HTTP connection with some service → there is always some timeout.

1. Complete outage
2. Unresponsiveness → poor user experience.

Idea: What if we make a call to a service, only if it is healthy

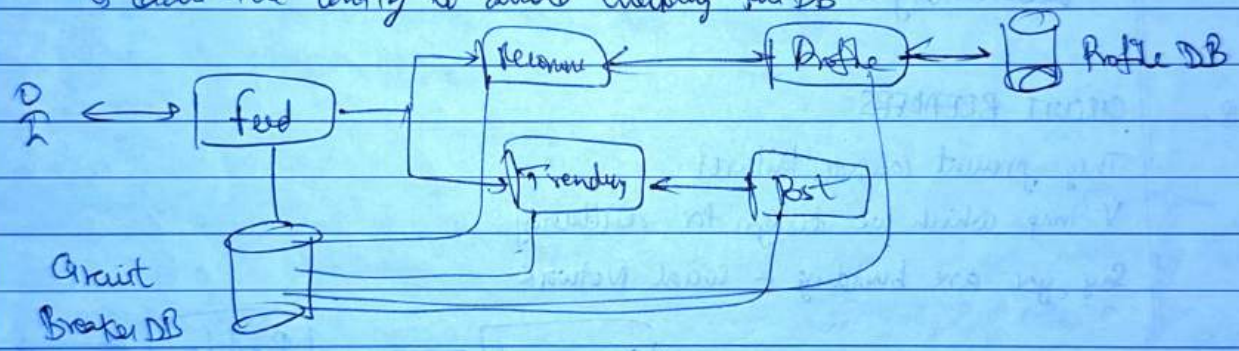
↑

This is circuit breaker

↳ Prevent entire product from collapsing by preventing cascading failures.

How is it Implemented? (Practically)

- A common DB holds the settings for each breaker (column for healthy (or) unhealthy)
- Services, before making calls to others, checks the config.
  ↳ cache the config to avoid checking the DB.



In case of outage, the circuit is tripped and DB updated.
Services will periodically check and stop sending req. to affected services

19. <u>Data Redundancy & Recovery</u>

- API Servers are "stateless" but databases are "stateful"
  ↳ API Servers going down is fine because
  a new one will be spin up almost instantly. API server gets request and it does not matter which one handles it.
- Database going down is catastrophic; almost always an outage!!
  worst: - Disk crash leading to loss of data

A good system always takes care of catastrophic situation?

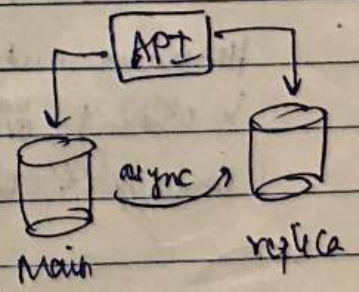
* The only way to protect ourselves against loss of data is to create multiple copies of it → Data Redundancy.
  Redundancy can be implemented at row/document level, table levels or DB levels.

<u>Backup and Restore</u>

- Daily backup of data (incremental)
- Weekly complete backup
- Storing one copy across region → Disaster recovery
- When something goes wrong, we can restore from last backup.
- Almost always the easiest thing to do.

Continous Redundancy

- Setup Replicas of the DB
  and writes go to both DB (sync/async)

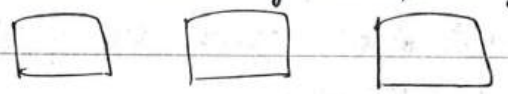


- API server writing to both DB's
- API writes to one and is copied to other asynchronously

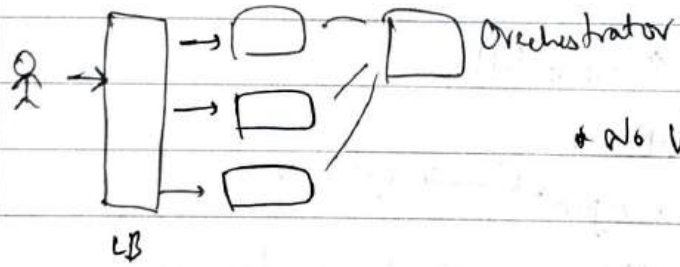
- If the main DB gets down, replicas can take its
  note:- this replica may just be a
  stand-by. and not serve any traffic.

<u>Leader Election for Auto Recovery</u>

say we have a bunch of servers, serving HTTP requests.



When one of them goes down, we have to spin up a new one.
Doing this is the responsibility of another module, say orchestrator



* No human intervention

But who is keeping an eye on the orchestrator! Who monitors it?
Another orchestrator!?
Hence we need automated way to recover the system
↳ when 1 orchestrator is down, somehow other orchestrator
backs up & takes responsibilities

So we run orchestration in leader-follower mode. Multiple nodes running the orchestrator code. One leader while others are workers

- Orchestrator leader keeps an eye on the workers.
- Orchestrator workers keeps an eye on the servers.

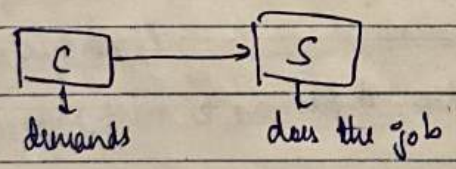
Workers ping the servers and checks if they are healthy.

If orch worker finds a backend is unhealthy, it spins up a new one.
If orch-leader finds that orch worker is unhealthy. it spins a new one.

- When orch leader goes down?

Thats when we do leader election and system is auto-recovered.
How workers will choose the leader depends on the leader-election algorithm.

Q1. <u>CLIENT - SERVER MODEL</u>



Most common way 2 machines talk to each other.

The communication happens over the common n/w connecting the 2.
2 protocols to exchange data - TCP & UDP.
↳ 99.99% we use this.

<u>Some imp. properties of TCP</u> (All backend APIs, web browsers)

1.

1. TCP connection requires 3-way handshake for setup.

→
← Ack
→ confirmation

packets send thrice (expensive) but reliable

2. TCP connection requires 2-way handshake for teardown

→
←

3. TCP does not break immediately after data is exchanged.
↳ breaks because of new interruption.
↳ breaks because server/client initiated it.

Hence connection remains... almost 'forever'!

Protocol over TCP

TCP does not dictate what data can be sent over it.
Common format agreed upon by client & server is called a protocol: HTTP

~~Properties of HTTP 1.1~~

HTTP is just a format that client & server understands.

* You can define your own format, and make
1. your client send data in it
2. your server parses and processes it

Properties of HTTP 1.1

There are many version of it → HTTP 1.1 / HTTP-2 / HTTP 3

HTTP 1.1 is the most commonly used one

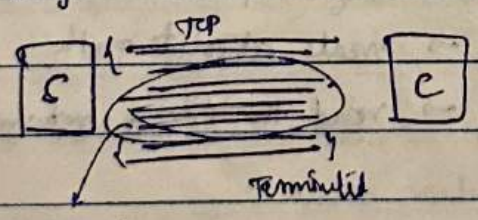
1. For client and server to talk over HTTP 1.1,
   * they need to establish TCP connection.
2. Connection is terminated when response is sent to client.
3. Almost new connection for every request/response
   ↳ little expensive.
4. Hence people pass → "Connection: keep-alive" header
   which tells client and server to not close the connection.
   * depends if server follows it or not.

WebSocket

Meant to do Bi-Directional communication.

Key-feature Server can pro-actively send data to client, without client asking for it.

Because, there is no need for setting up TCP every single time.



Bi-directional

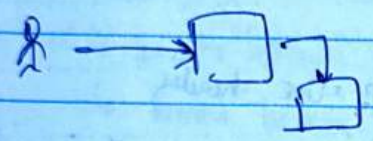
WebSockets

SocketIO

Anywhere you need "real-time", "low-latency" communication over the internet then we think about websockets.

22. BLOB storage and S3 → managed service by Amazon
↳ Binary large objects

Earlier when people uploaded any files, they uploaded it to "server" and were stored on the hard disk attached to it.



getting a file was simple, make an API call the handler reads the file & return.

This is precisely how /static folders/ routed worked.

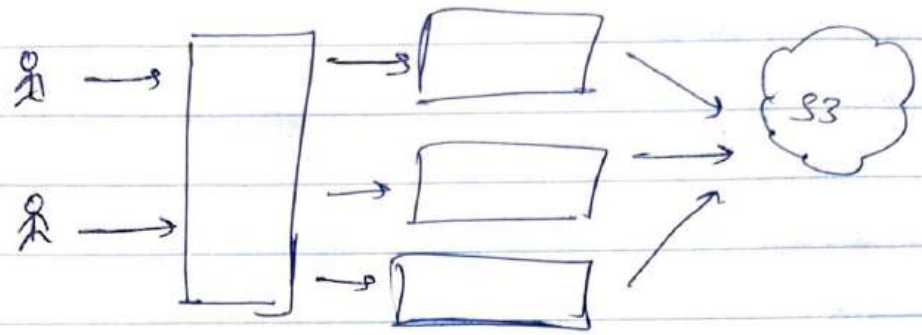
when user uploads the file → accept it on HTTP POST
'a.txt' → create absolute path using folder mapped to static.
/home/praddy/www/static/a.txt
→ store the file at ↑ location.

when user requests '/static/a.txt' → get the path from URL.
→ create absolute path
→ read the file & return.

This worked well for something, but won't work with multiple servers. because each server will have its own disk.

Now, in distributed systems, when there are multiple servers

So, it is stored now remotely:



Any "file" that needs to be accessible by any server, is stored at a place accessible by all.

On S3 you have

buckets : (namespace) eg :- insta-images, my-bucket → any unique-name.

keys :- path of the file within bucket

s3://insta-images/ $\underbrace{\text{uyu123/72896.jpg}}_{\text{key}}$

you can seamlessly create the file, replace the file, delete the file, read the entire file.

Advantages - Cheap, durable, storage
- can store literally any file
- scalable and available
- integration with a lot of AWS & Big Data services.

Disadvantages - reads on S3 are slow
- quick reads required means, you should not use S3 & SSD & HDD attached to instances are better
- not a fully fledged file system.

You should use S3 when you want to store 'blob' that is centrally accessible :-

- DB backups
- static website hosting
- Big Data Storage
- Logs archival
- Infrequently accessed data

23- <u>BLOOM FILTERS</u>

Bloom filters are approximate/probabilistic Data structure says with 100% certainity that an element does not belong to a set.

<u>for example</u>

Insta wants to recommend reels, but it does not want to recommend something you saw already.

Naive way - keep track of everything that a user saw in a set.

u1 <p1, p4, p7, p1024, p10565>
↑
user

ppl watch 1000s of reels every day

overtime the set will become huge.

so what?

To check the existence of a key in set - We have to load the entire set in memory and then check.

↑ super expensive and time consuming

So can we do better?

Key insight: - When something is "watched", you cannot take it back.

↳ once a post/reel is watched by you. We do not take it out of the set!!.

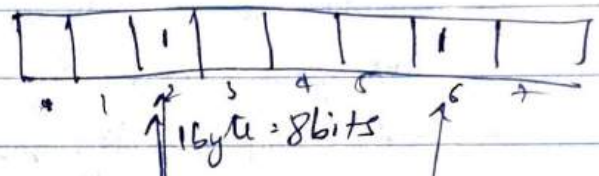
This means storing actual data is not worth it.

This is the crux of Bloom filters

Bloom filters

Filter ≙ bit array

(eg:- we take 8-bit array)



words i want to put in

apple → f(apple) % 8 → 2

↳ Hash function

ball → f(ball) % 8 → 6

cat → f(cat) % 8 → 2

dog → f(dog) % 8 → 5 filter[5] = 0 → dog not present

elephant → f(elephant) % 8 → 6 filter[6] = 1 → but it is one - so elephant may be present

Thus, Bloom filter NO → 100% certainity

YES → Maybe there.

1. Space Efficiency
2. Very efficient in checking existence (just an array lookup).

False positivity rate

loosing on correctness, when it is YES. (elephant, example).

When we add more words, false positivity rate increases as all bits could be filled.

So, in such cases

1. we have to re-create bloom filter with larger size and populate the keys again from the DB as the hash function has changed.
2. Estimate the max keys and provision a large one to start with.

Practical Bloom filter

Libraries present

Redis has it as one of its core feature default 100 or 1000.

↑ now-a-days, mostly people go for this

Practical Application

1. Insert but do not remove data.
2. Need a NO with 100% certainity Eg:- Tinder feed.
3. False positivity should be okay.

Consistent Hashing

It is an algorithm that solves data ownership.

Ownership → who owns this part (or) who will handle this request.

How does LB know which server to route to - it is done using hash based routing

fn(user.id) % 3 (no: of servers)

SHA-256 or SHA-128 are popular choices.

SHA256(u_i) = x

server = x % 3 = i

- Hashing logic is not a service, just a simple function.
  It is handled in the Load Balancer.
- If one of the server is taken down, then the routing function changes now the request will be evenly distributed b/w remaining 2 servers.

  fn(r.id) % 2

  ↳ a user token in the request.

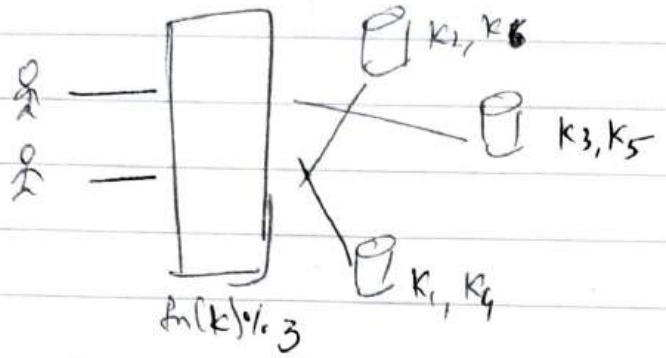
LB are stateless

Because the API servers are stateless; which means any server is equally capable of handling requests it does not matter if:-

request that was handled by server 1

now starts going to server 0.

Hash based 'routing' for distributed storage.



Picking which node "owns" the data depends on hash based ownership.

Say we want to store 6 keys - K_1, K_2, K_3, K_4, K_5 & K_6.

When we remove a node then the data goes into different nodes.

- Can we minimize the data movement?

This is where consistent hashing comes in

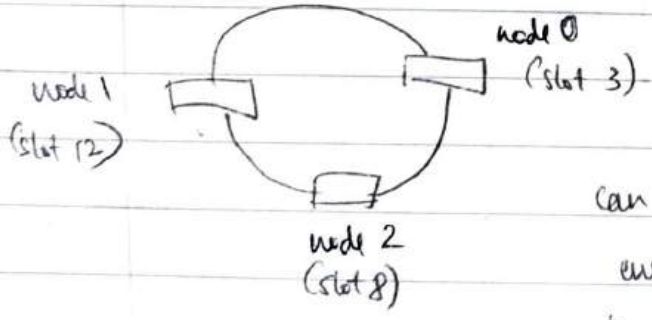
This is the limitati of hash based routing.

What consistent hashing is?

An algorithm (function) in determining who owns it.

Consistent Hashing

simple visualization



Hash function (SHA 128)

Given hash functions are cyclic we can visualize it as a ring of integers. every node occupies one slot in the ring; the slot is calculated by passing node's

Array of 3 elements

the ring can be modelled as a simple array and

Here's how the ownership is determined.

$K_1 \rightarrow$ hash $\rightarrow 0 \rightarrow$ node to right $\rightarrow$ Node 0
$K_2 \rightarrow$ hash $\rightarrow 10 \rightarrow$ node to right $\rightarrow$ node 3

Consistent Hashing tells you who owns it.

<u>Scaling up</u>
When we add new node to the ring
Say node 3 hashes to slot 1.
The keys that hashed between slot 12 and slot 1 will now be owned by node 3 instead of node 0.
Other keys continue to remain at the respective nodes
↳ Minimal data movement.

<u>Scaling down</u>
say we scale down and remove node 0. All the keys that were owned by node 0 will now be owned by node 2 (next in the ring)
<u>Minimal data transfer</u>
Operationally,
1. copy everything from node 0 to node 2
2. Remove node 2.

Introduction to Big Data Tools

When one machine is not enough to process the data we divide and conquer → This is Big Data processing.
Companies use this to process massive amount of data, and extract insights out of it, train ML models, move data across databases and much more.

- when too much data needs to be processed quickly, use BD tools.
- all these fancy processing on commodity hardware.

Counting word frequency

Given a 1TB text dataset find frequency of each word

Approach 1: Simple

1. Load data on one machine (disk)
2. Read character by character
3. when space is encountered
   update a in-mem hashmap.
   do count ++.

Simple approach that runs in O(n) time approx.
Can we parallelize it.

Approach 2: Threads

Each thread can handle a chunk of data
But, what if the data is 100TB?
↳ what if it does not fit in one machine.

- threads are bounded by CPU cores

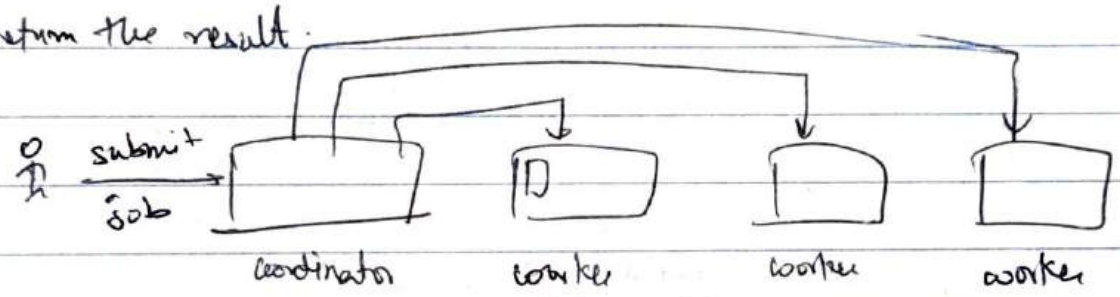
Instead of one machine, can we distribute the workload across multiple machines

↳ and leverage parallelism.

Approach 3: Distributed computing

More computers, more CPU, more processing.

Idea: Split the file into 'partitions' distribute the partitions across multiple servers, then all sent to one machine to combine and then return the result.



Challenges:-

1. What about failures
2. What about recovery
3. What about completion
4. What about scaling & distribution

Big Data tools manage these complexities for us:-

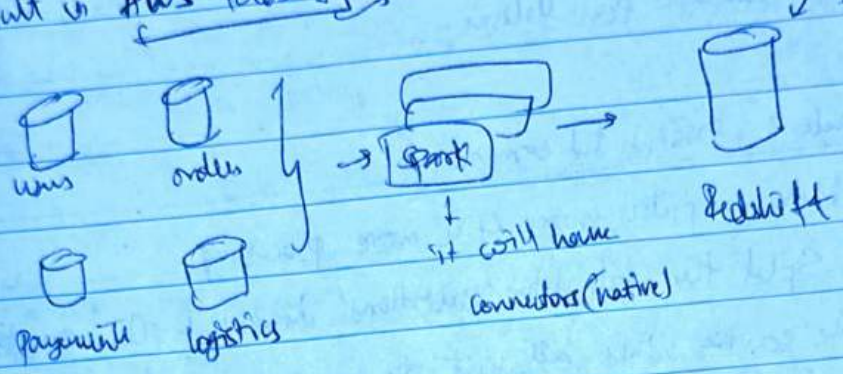
- distribute across machines
- Knowing which machines are doing what
- retrying in case of failures
- reprocessing in cases of crashes (or) completion
- cleaning up the resource

Apache Airflow, Apache Spark, Hadoop.

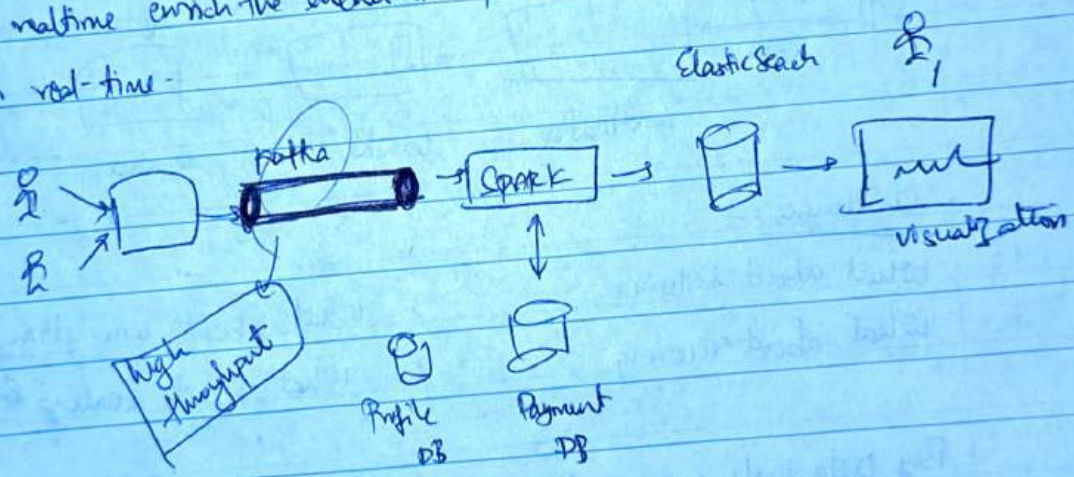
Spark & Flink

Large scale data processing on commodity hardware

Eg:- combine user, order, payments & logistics DB and put the result in AWS Redshift → Data warehouse



Eg:- when any activity is happening, event are streamed to Kafka. In realtime enrich the events and put them in analytic DB. to track in real-time.



Design e-commerce product listing

For a shop design a system through which owner can

1. Add a new item
2. update/delete products on the website
3. List all products on the website
4. Customers should be able to quickly access the catalog.
5. Payment is out of scope for this problem statement.

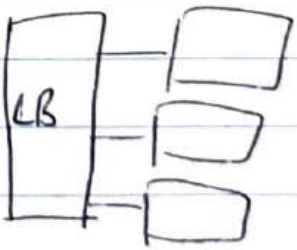
Storage

- not a huge data only 100 rows
- there seems to be a structure
  we can go ahead with SQL DB.

a 'Product' table to hold all the catalog.

Serving the data

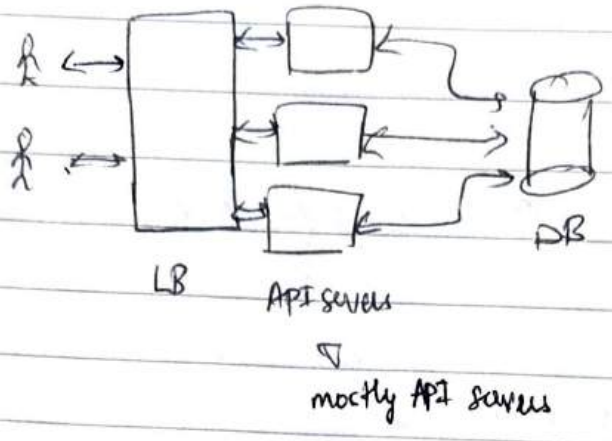
- Simple REST based HTTP webserver is fine.
- we will need many (handle low request).
- hence put a load balancer.



load Balancer will have a DNS like api.mystore.com

storing the data ✓

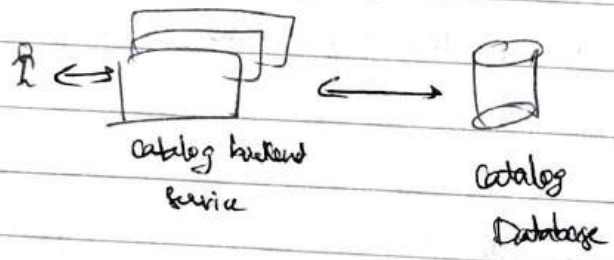
serving the data ✓



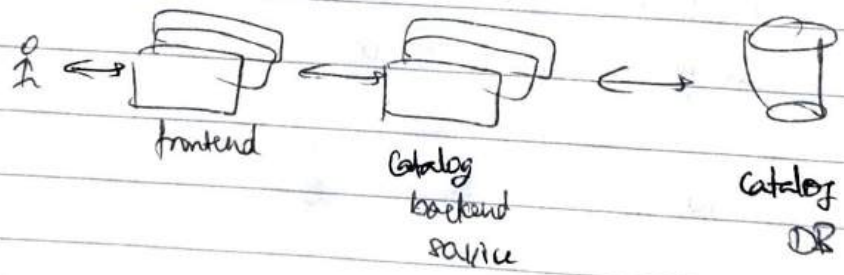
→ 3 tier architecture

- UI
- compute
- storage

Instead of drawing LB & multiple same everywhere everytime, we simply draw the above diagram as:-



End user needs a frontend.



<u>Shop owner's admin interface</u>

Shop owner needs admin interface/console to manage the console. we can keep the Admin UI seperate frontend service, which interacts with the same backend service to do admin things.

Let's understand load on each component &

Load on frontend?

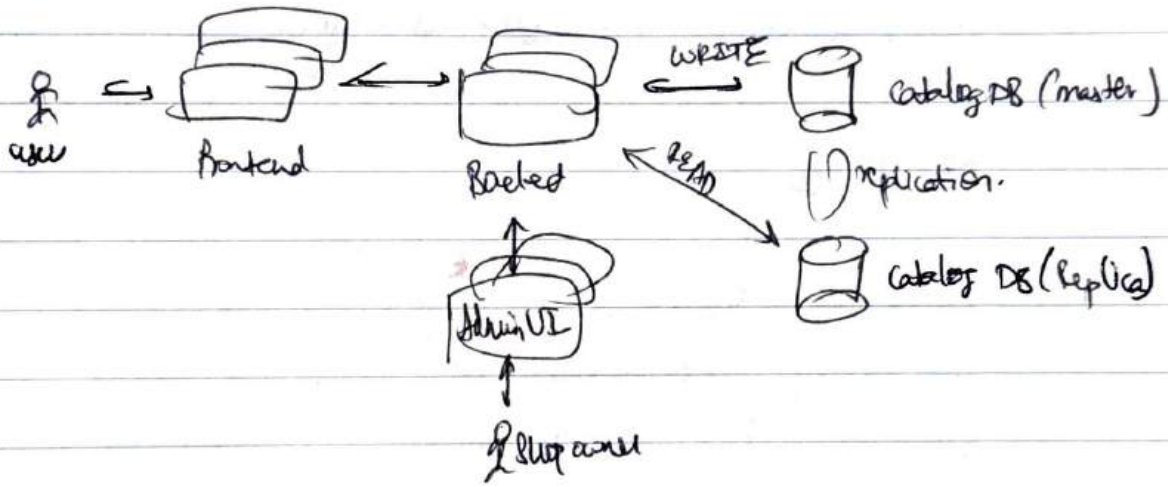
Load on AdminUI → Secure enough as only one admin.

<u>Scaling DB</u>

when a lot of load comes in, the DB will be under heavy load.

Given the use case is to list the pages,

the load is certainly Read load, hence we add read replicas.



- Design DB schema for the system?
-
- * Even masters can serve Read requests, there is no rule that there should be no reads on masters.
- If we add cache, we need to think of cache invalidation

Q7. Designing an API Rate Limiter

Systems break under tremendous load and we need to ensure that doesn't happen.

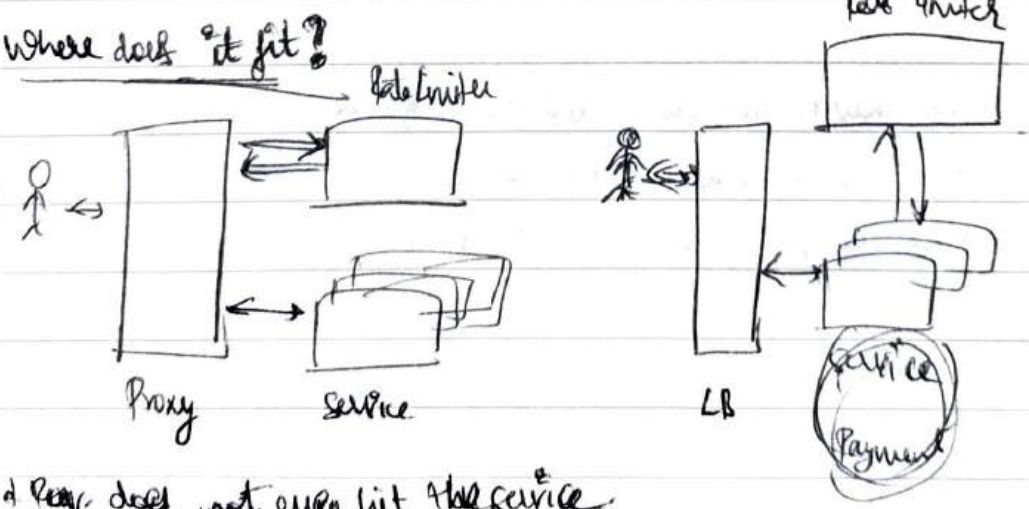
Design a rate limiter, that: -

- limits the number of requests in a given period.
- allows developers to configure threshold at a granular level.
- does not add a massive additional overhead.

Rate Limiter

- First line of defense
- Any incoming request is first consulted against rate limiter.
- If we are under limits, go through.
- Otherwise, reject with error (429)

Where does it fit?



A Proxy does not even list the service.

Dissecting Rate Limiter

Rate limiter needs to track # requests in a given period. Hence we need a database to hold the count

1. for every incoming request, we should update the db.
2. for every incoming req., we would read from db (aggregate)
3. we need ability to "clock" the time.

Say, we rate limit per user.

1. We store # request in current period. user_id → count
2. Once the period is over we reset the counter.

Resetting the counter ≈ expiring the key.

KV store + Expiration → Redis

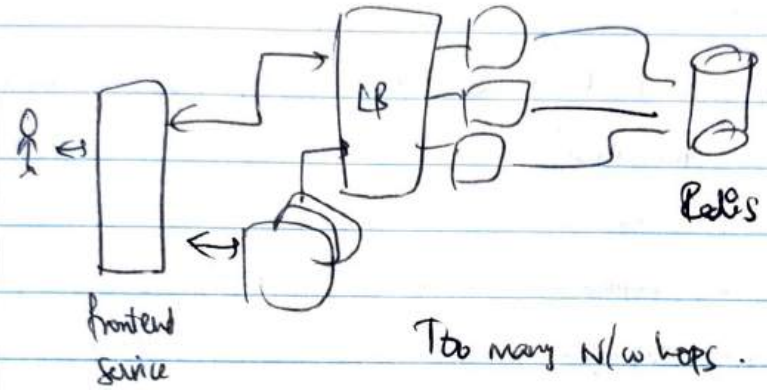
↓

also gives us fast in-mem writes + periodic persistence.

Checking and Updating

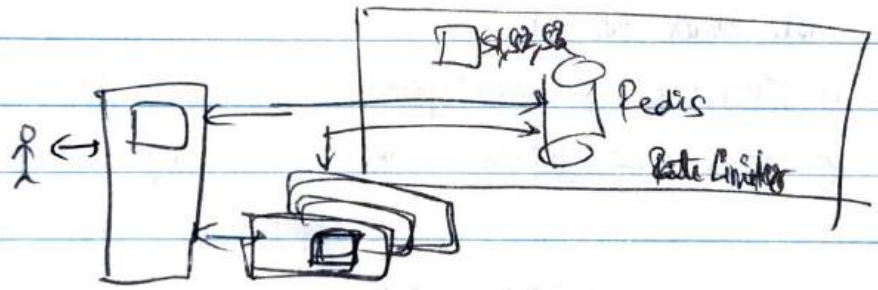
- we expose service (HTTP endpoint) that talks to RL DB.
- We let the services directly talk to RL DB.
  - ↳ saves one N/w hop
  - ↳ but expose critical internal details

27. ~~Designing an API Rate limiter~~ Backend server g RL



Too many N/w hops.

so

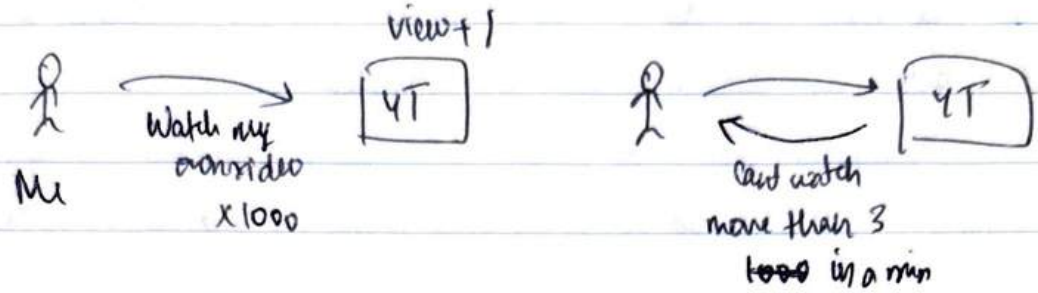


Making RL as a library and not a service
↳ it contains all the business logic.

Scaling a Rate limiter

Given the service are directly talking to the rate limiter DB.
Scaling Rate Limiter = scaling the DB.
Should use vertical scaling? YES
Should make Read replicas? No traffic is not read heavy.
Should we shard the DB? YES.
↳ any fn in the backend service.

Design a Rate Limiter



Problem req.

1. Stop malicious users from submitting too many requests to our services
2. Support a variety of different rate limiting technique

Capacity estimates

1) 1 billion users
2) 20 services to rate limit
3) User Id = 8 bytes, int to keep track of count = 4 bytes
4) 1 billion x 20x(8 + 4) = 240gb

→ If we plan ~~to~~ storing data in memory, we will need to partition.

What to Rate Limit On?

1) User Id

\+ Easy to track, bad users will be blocked

\- Users can create many accounts, not all services need authentication.

DDOS attack.

2) IP address

\+ Easy to track, no need to be signed on, prevents multiple accounts

\- Good actors on some n/w will be throttled.

Hybrid approach? User Id when authenticated, IP address otherwise.

Rate Limiting Interface

We will be able to support multiple different implementations of the rate limiter interface.

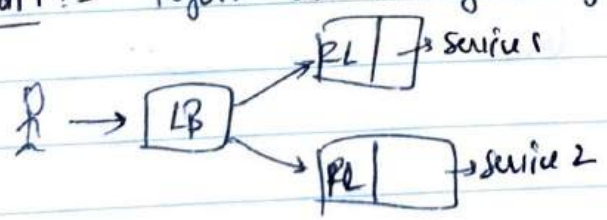
boolean rateLimit (long userId, string ipAddress, string ServiceName, Date requestTime)

→ Returns true if request should be rate limited.

Where does the Rate Limiter go?

Note:- The amount of data that we need to store is too big to put on our load balancer

Option 1:- Perform rate limiting locally on each Service



Option 2 → Dedicated distributed Rate limiting

Service-Local rate limiting

Pros:

1) No extra n/w calls required to perform rate limiting
2) Fewer types of components to manage.

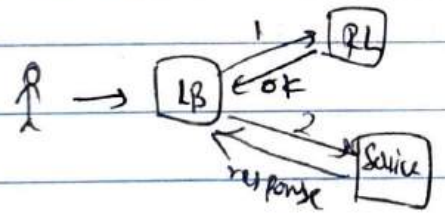
Cons:

1) If we are getting spammed (call hits the service) can use up server n/w bandwidth
2) Our application and rate limiting scaling are tightly coupled.

↓

Strong not enough, requires more replicas.
Just have to scale for the sake of RL
If one service goes down, we lose RL into with it.

Dedicated distributed Rate Limiter



Pros:

+ Shields application servers from large bursts of n/w traffic
+ Scales up or down ~~[illegible]~~ independently.

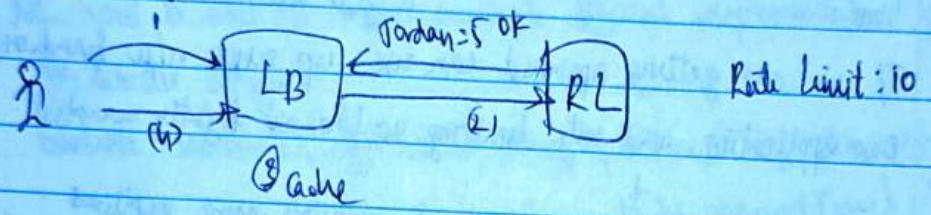
Cons:

- Introduces extra n/w call to API requests

Rate Limiting Caching

Since our rate limiter now requires an extra n/w call, it would potentially be very useful to avoid this for users who submit lots of requests

Idea: Write back cache on the load balancer once reaches certain threshold.



Rate Limiting Database choice

Recall: We want both reads and writes to be as fast as possible and we dont have that much data to store.

→ No time to access disk.

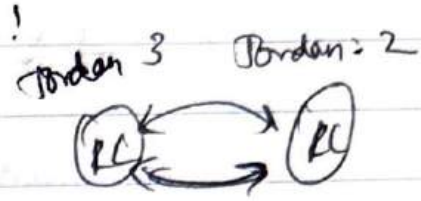
So Redis/memcached (so some in memory DB)

↳ can take advantage of existing DS.

Replication also fairly easy to set-up.

Replication Choice?

We need our rate limiter to be fault tolerant!



Multi leader / leaderless replication?

→ Could increase write throughput, lower write load on each system

→ Have to wait for anti entropy for counts to sync.
(could use CRDT)

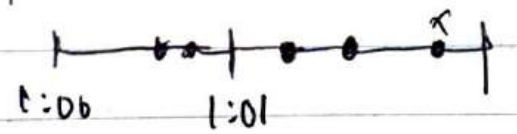
Single Leader replication

→ Counts of requests on master accurate, can do all reads on master

→ If we have enough shards should be able to have ~~reasonable~~ reasonable R/W throughput.

Fixed Window Rate Limiting

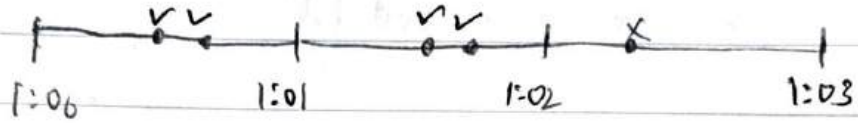
Example: 2 requests, resets every min → 2req/min



Example: user 22

(1:00, 1) ✓
(1:00, 2) ✓
(1:01, 1) ✓
(1:01, 2) ✓
(1:01, 3) ✗

Sliding Window Rate Limiting



We want to use a linkedlist

1) 1:00:10
2) 1:00:10 → 1:00:20 } outside of active time, so purge
3) 1:01:40
4) 1:01:40 → 1:01:50
5) 1:02:10 } recent enough so dont add.

Pseudocode

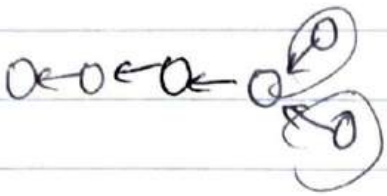
```
Requests : Map < Service, Map < UserId, LinkedList <Date>>
LIMIT : 2
WINDOW_SIZE : 60

boolean rateLimit (long userId, String serviceName, Date requestTime) :
    requestsWindow = Requests.get(serviceName).get(userId)
    cutOffTime = requestTime - WINDOW_SIZE
    while (requestsWindow.head().earlierThan(cutOffTime)) →
                                        requestWindow.popHead()
    result = requestWindow.size > LIMIT
    if result == false → requestWindow.pushTail(requestTime)
    return result
```

Concurrency consideration?

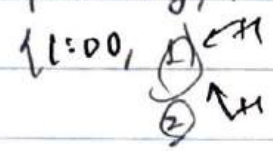
Are there potential for race condition? YES
Threading rear → YES

Do we need to perform any locking? YES

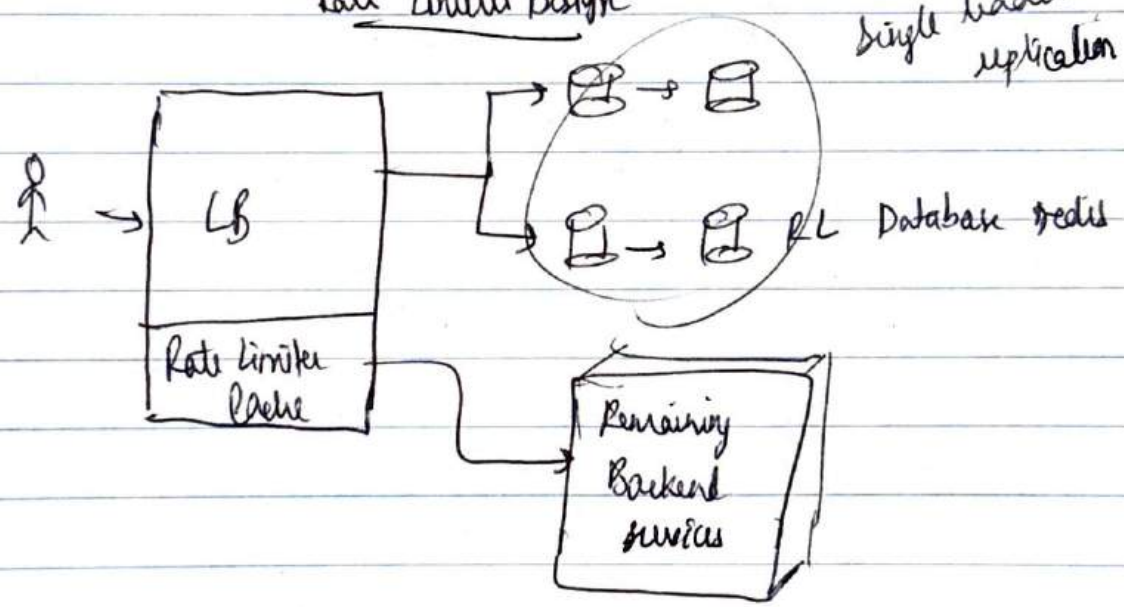


} Need to lock linked list head when inserting new elements

→ Also need to have locks in place for deletes!
Unfortunately, there doesn't seem to be a good way around this.



Rate Limiter Design



Microservices - Decomposition pattern

- Breaks down a monolithic system into smaller, independent services, each responsible of specific piece of complexity.

① Decompose by business capability

② Decompose by Subdomain

(say Banking then it has subdomains → Payment, Loans, Accounts)

③ Decompose by Vertical Slice

④ Decompose by Transaction boundaries

(ensure each services are designed ~~around the~~ can perform tasks without requiring co-ordination (or) locking from other services).

⑤ Decompose by Scalability / Work load

⑥ Decompose by User (or) client type. (Vendor Service, Customer Service, Admin Service).

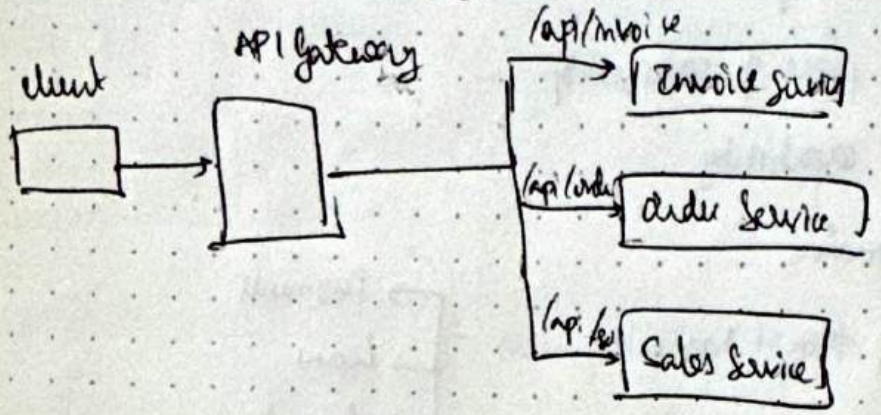
⑦ Decompose by Data ownership.

API Gateway

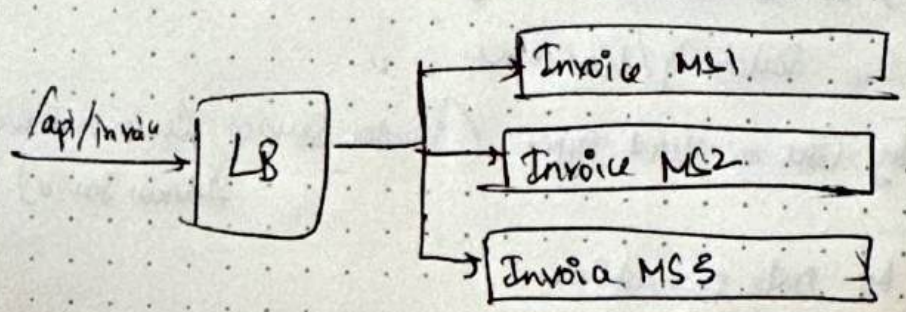
- A single entry-point, how does it handle millions of reqs/second?
- How is API Gateway different from Load balancer? Does it come before (or) after the load balancer?

What is API Gateway?

→ API Gateway accepts the client API request and routes them to correct backend service based on API endpoint.



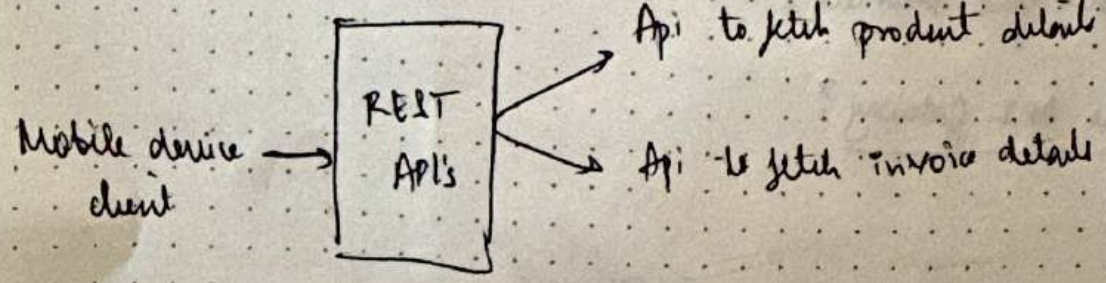
→ Load balancer simply distributes the traffic to the multiple instances of microservice based on various service like health, traffic load etc.
But load balancers dont have ability to understand an API and then take decision, where to route it.

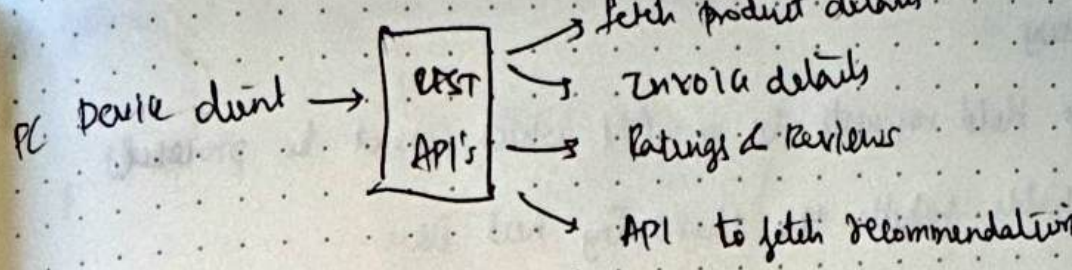


Apart from routing, other capabilities of API Gateway are :-

1) API Composition

Let's say, we are visiting "My orders" page.



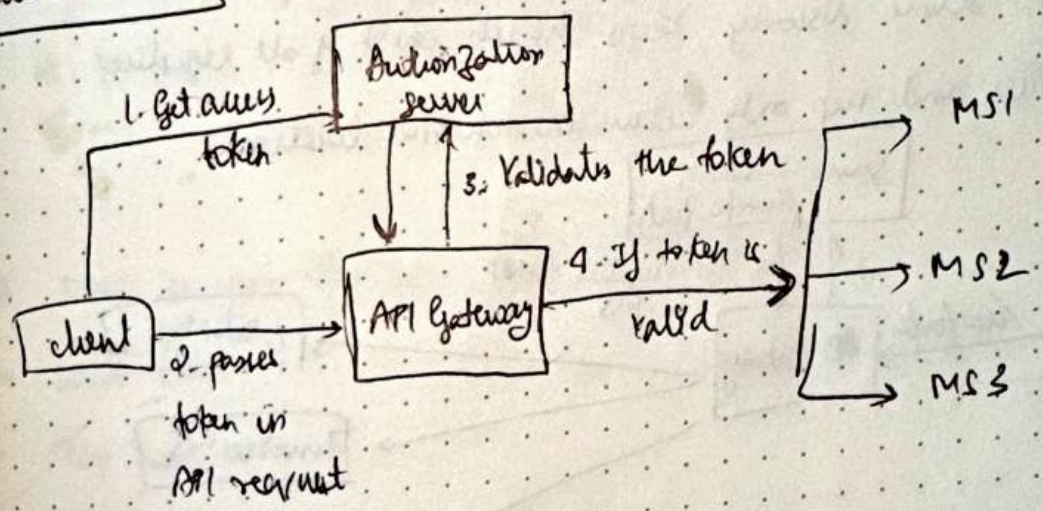


So, based on the client type, the API Gateway can show the same page differently. So, API Gateway has this intelligence.

"API Composition" → NETFLIX uses

(single authentication & authorization)

2) Authentication



3) Rate limiting

→ Manage Burst limit, means no. of concurrent request that API Gateway can handle before it returns 429 (too many requests).

→ API throttling, limiting the no. of requests from an individual (or) an application by temp. blocking, once they cross the allowed request rate.

/api/invoice → should not get invoked more than 10 req/min.

→ IP based blocking

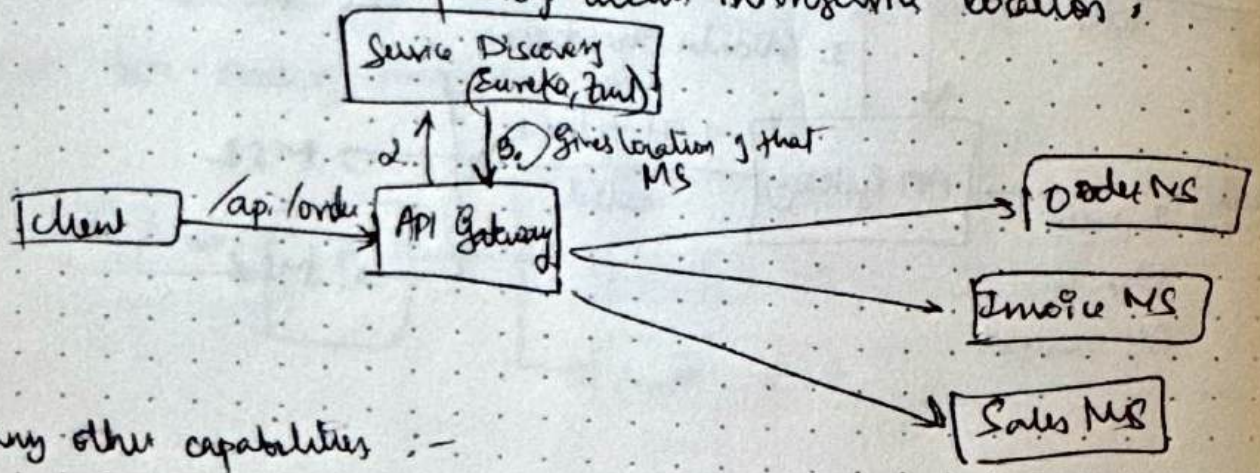
→ API Queues → Hold requests to an API, which cannot be processed immediately, helps handle the thundering herd issue.

4) Service Discovery

As microservices scales up and down, its necessary to know the location (IP address & port). Service discovery keeps track of those :-

Approach 1 → Each microservice register / de-register themselves.

Approach 2 → Service discovery keeps health check of all registered microservices and keep only active microservice location.

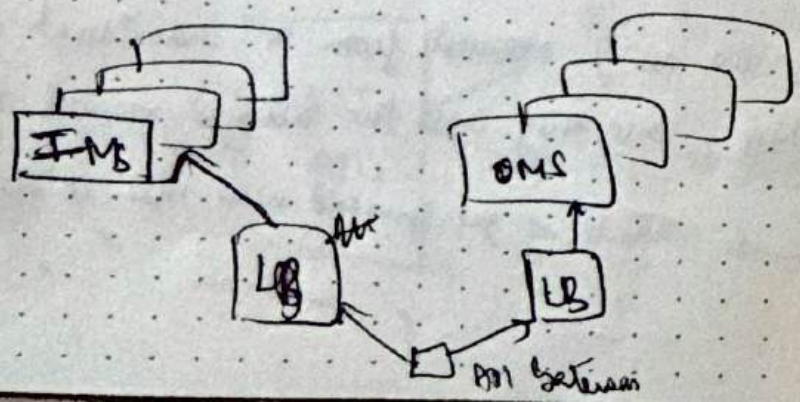


Many other capabilities :-

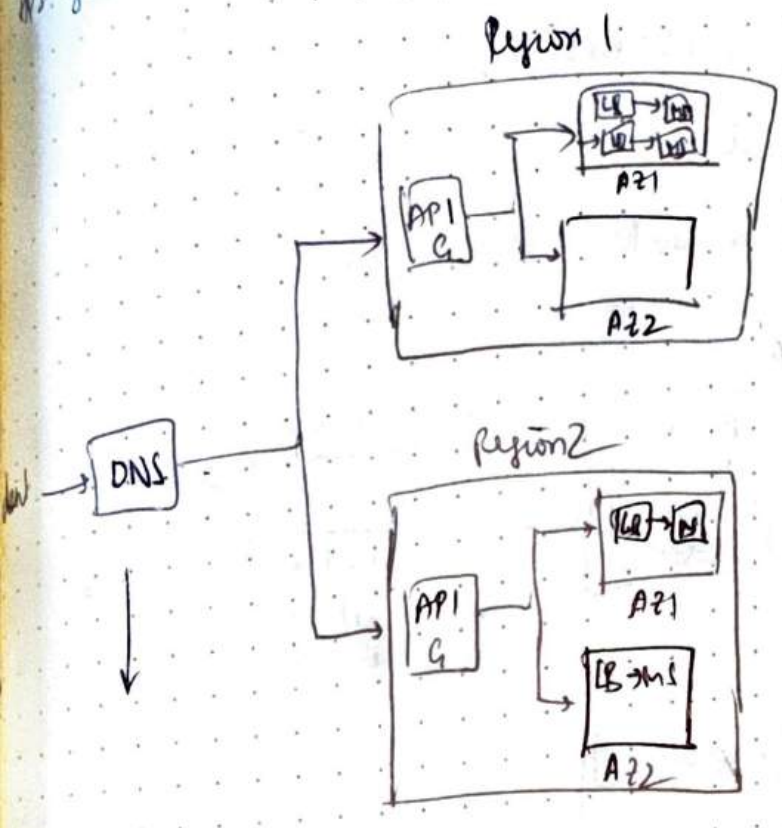
(i) Response caching

(ii) Logging

(iii) Request - response transformation



API Gateway is on regional level / zone level.



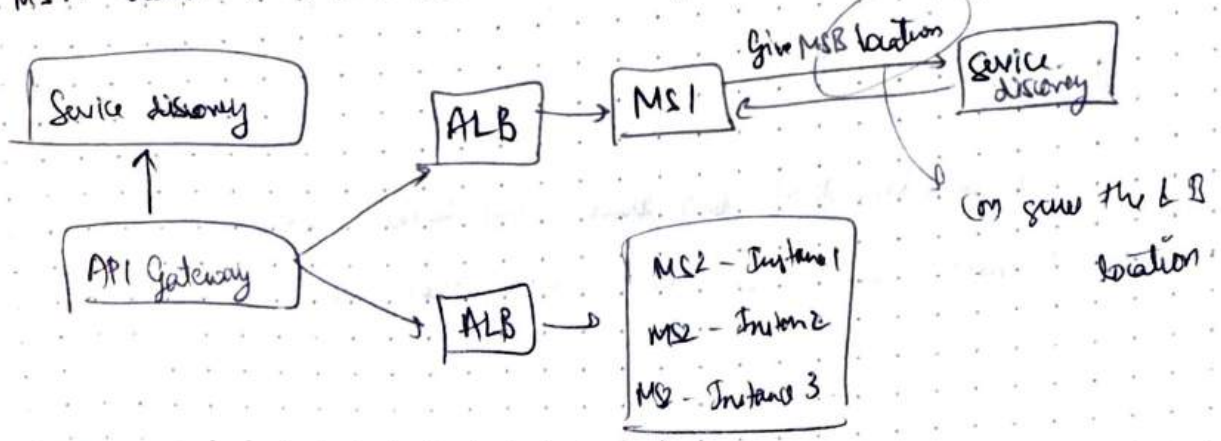
But DNS is not the single point of failure

- Local DNS
- Top level DNS
- Group DNS

It is hierarchical.

How do Microservices communicate with each other?

① MSA wants to communicate with MSB, then it needs its location (address & port)



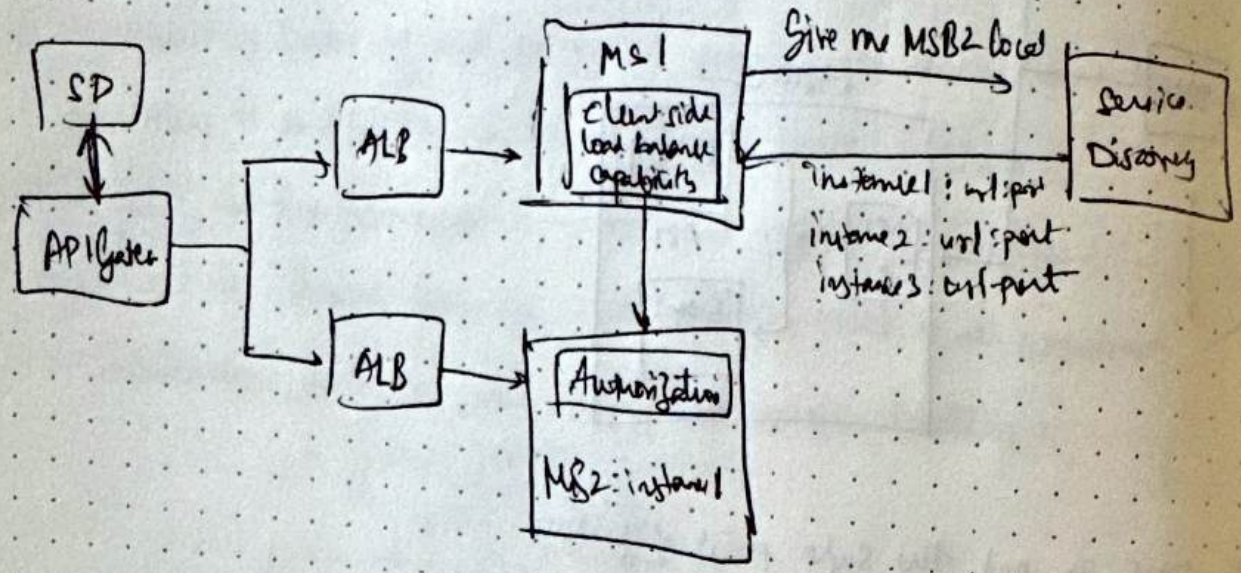
In MS1,

application.properties

microservice.url = {instance-url} : {port-no}

Client-side load balance capability.

→ Microservice 1 calling ALB form another n/w hop, hence its not usually used.



Circuit Breaker capability

Say I have M1, and I want to call M2.
Let's say that we are calling 10 times and all requests failed. Then CB tell M1 not to invoke M2 for some-time say 1 min. After one minute we can open the circuit break. (Hystrix)

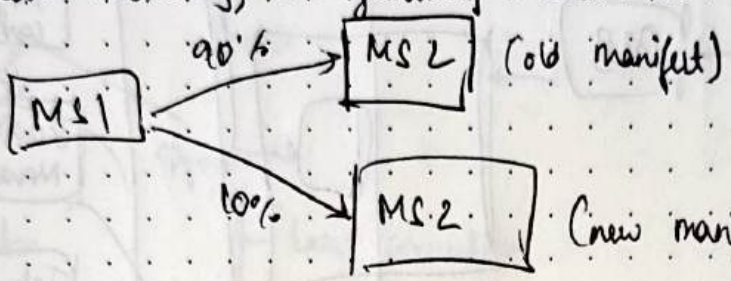
Retry capability

If MSA is invoking MSB, but there was some issue, so once you retry it might come. For eg:- 5xxx errors helps us to retry.

Deployment Strategy

Canary, Red-Black & Blue-green deployment strategy.

for example :- In Canary, I want to 90% traffic on old and 10% traffic on new initially, and gradually increase the number of new manifests.



Generally, load balancer helps to control this, means "client side load balancer" also has to handle this different deployment strategy.

Telemetry Capability

We need to collect data which will help in analysis and monitoring of a component. Like say what APIs are getting traffic?

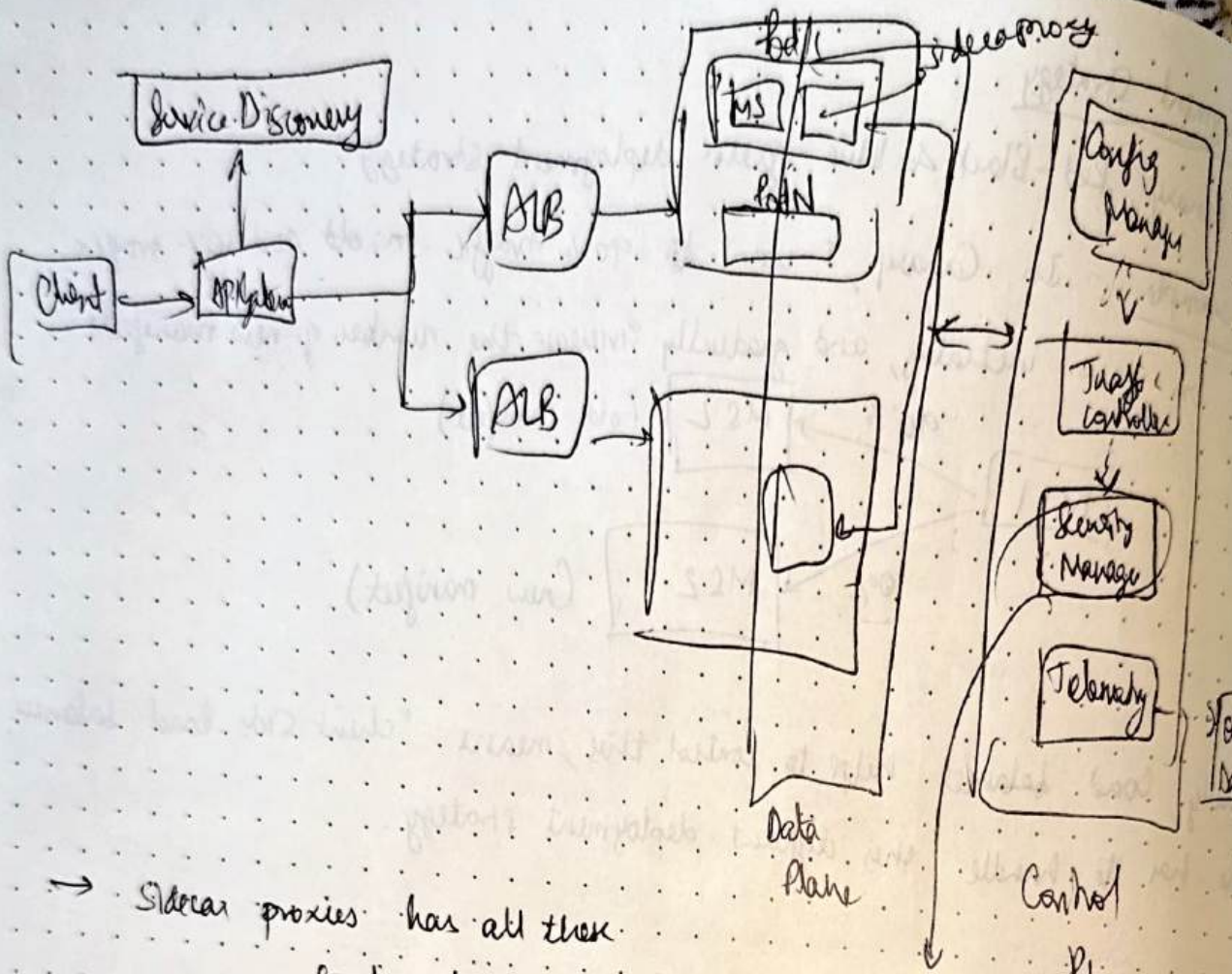
"latency"

"error rate"

"Traffic"

* MA → MB, Just for communication, we required so many things. So, the better solution is Service Mesh.

Each pod hosts a micro-service instance, sidecar proxy also is added on the same pod.



→ Sidecar proxies has all these configurations

for authorization & authentication.

Istio → Service Mesh

Sidecar Proxy known as Envoy.

Configuration Manager → Galley

Traffic controller → Pilot

Security Manager → Citadel

MA → MB

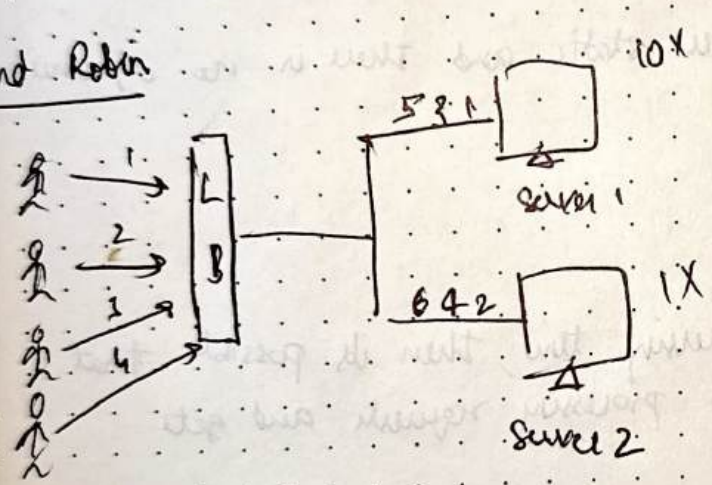
no need of ip (or) port

sidecar proxy, uses SD data from configuration manager & forwards to the best instance in MSB.

Load Balancer

- Network load Balancer Type (L4 Balancer)
  - faster
  - Static
    - Round Robin
    - Weighted Round Robin
    - IP Hash
  - Dynamic
    - Least connection
    - Weighted least connection
    - least response time
- Application load Balancer Type (L7 Balancer)
  - Caching
  - Advanced

Round Robin



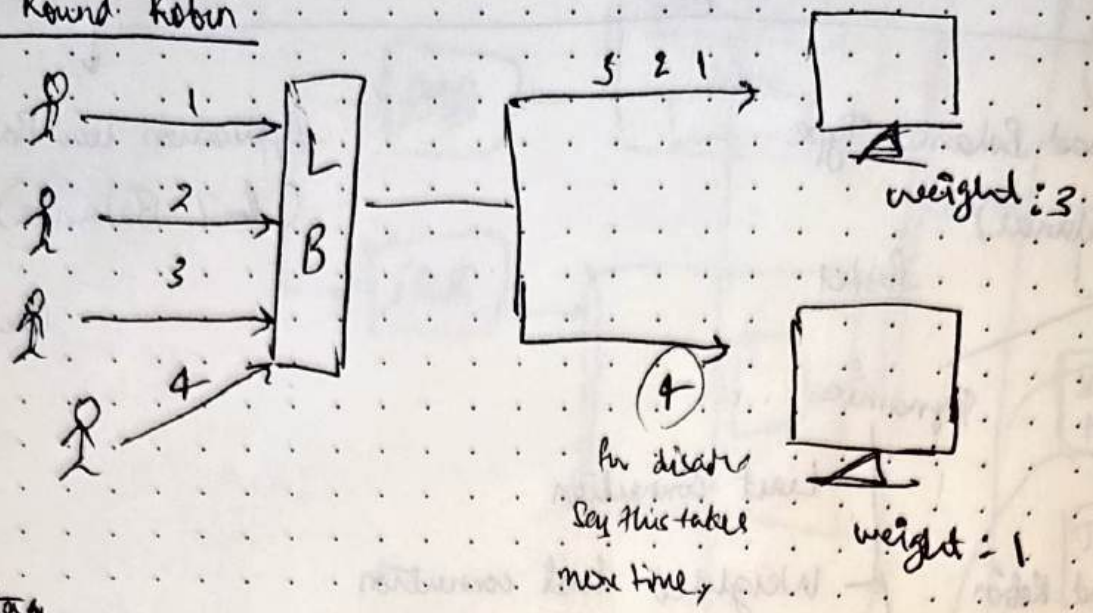
Advantages

- Very easy to implement
- Equal load distribution to all servers.

Disadvantages

- One server with low capacity & another with high will be treated as same.
- Chance that low capacity server will go down because of overload of requests.

Weighted Round Robin



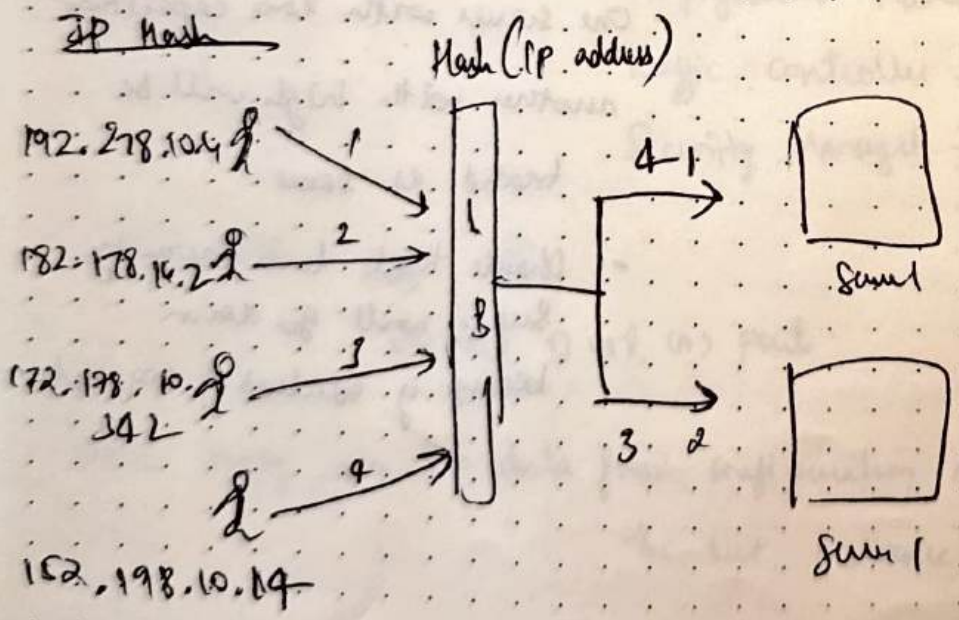
Advantage

- Low capacity server will get less requests itself.
- Easy to implement as weight is static and there is no dynamic computation.

Disadvantage

- If request have different processing time, then its possible that low capacity server gets high processing requests and gets overburdened.

IP Hash



Same client, same server then this algo is good.

Advantages :

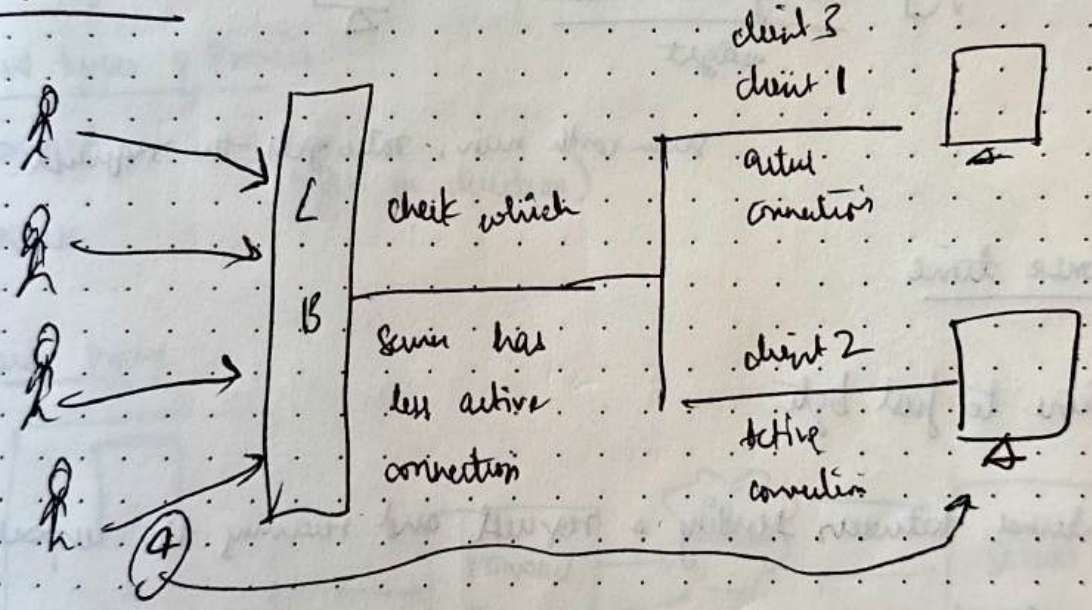
- Good for use cases, where same client needs to same server
- Easy to implement

Disadvantages

- If client request is coming through proxy, then all the clients will have same source IP address, and will overload one server.
- Cannot ensure equal distribution

DYNAMIC LOAD BALANCING

Least Connection



Advantages :-

- Dynamic, as it considers the load on each server, so chance of overburdened of the server is less when each server has equal capacity.

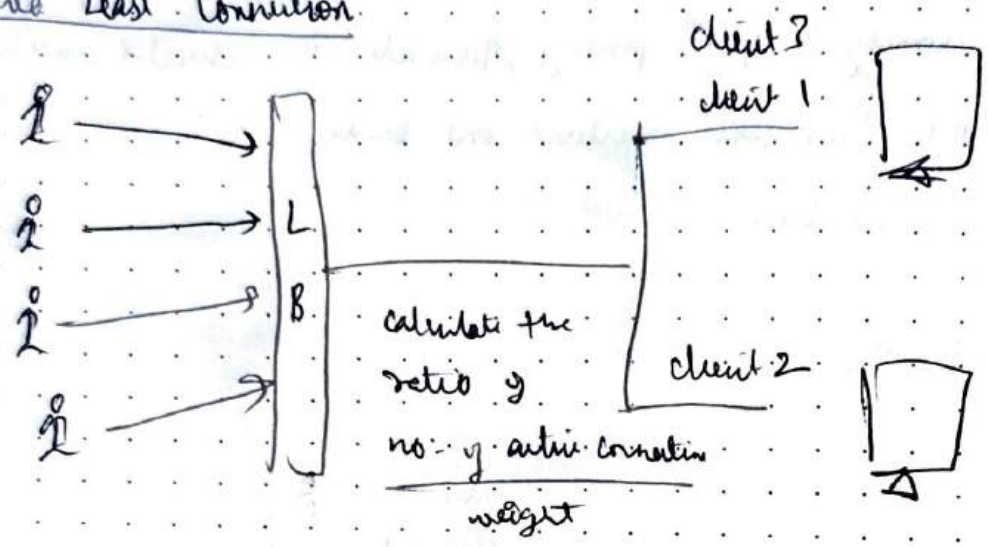
Disadvantages

- TCP connections can be active but possibly have no traffic. So purpose is failed.

Disadvantages

- No difference between low capacity and high capacity server. Chances that low capacity server will get over burdened.

Weighted Least Connection



Server with min. ratio gets the request.

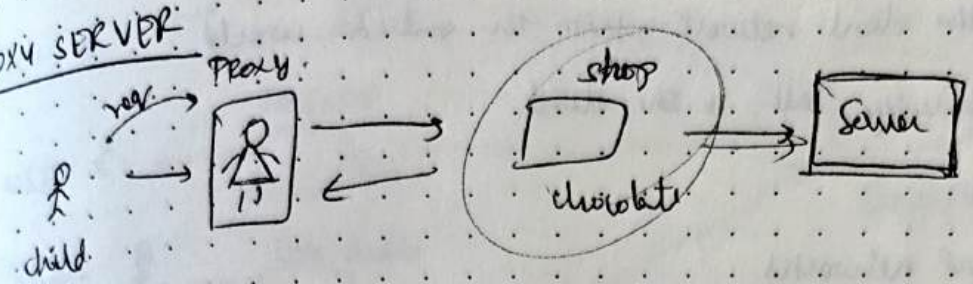
Least response time

TTFB → Time to first byte.

↓

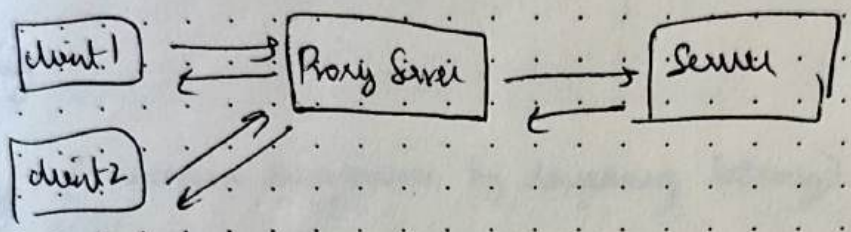
The time interval between sending a request and receiving the response from the server.

PROXY SERVER



(client) child is not directly talking to the shop-keeper.

So, here Mom acts as a proxy.



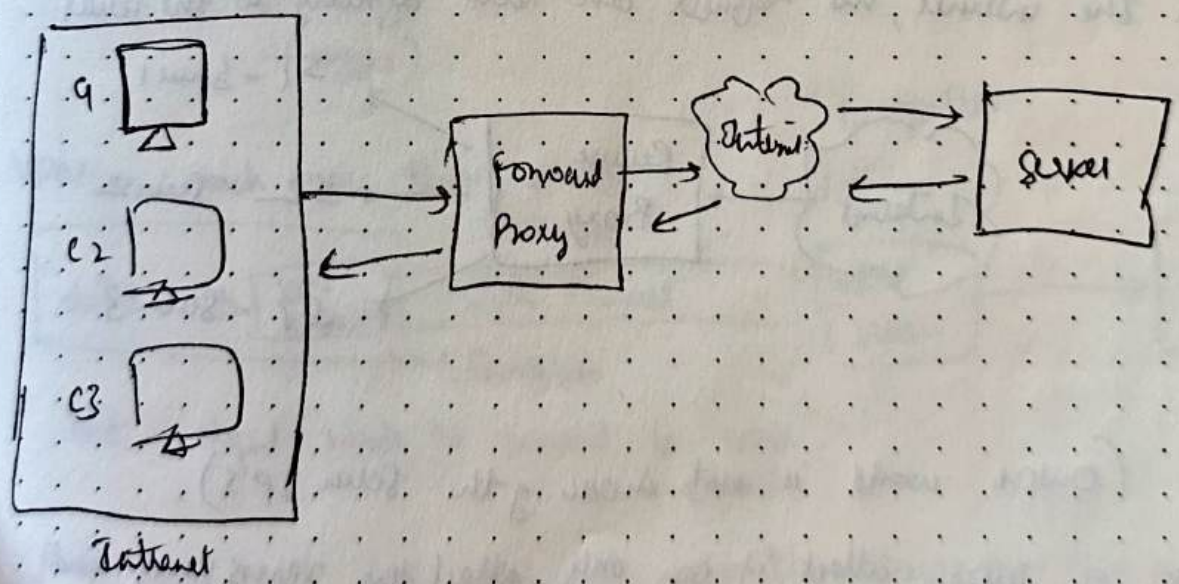
Proxy server acts as an intermediate between client and server.

Different types of Proxies :-

1. Forward
2. Reverse

(differs in direction)

forward proxy



→ Forward proxy hides the client network, from the outside world

No server can directly talk to the client.

Advantage

(i) Anonymity to the networks

(ii) Grouping of requests (can club requests and send as one)

(iii) Access restricted data / content

(iv) Brings Security.

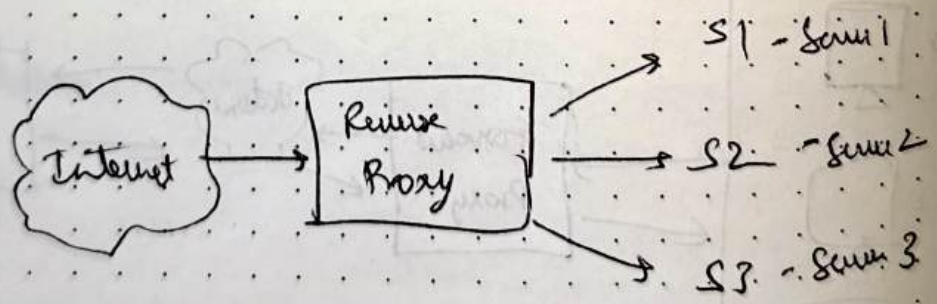
(v) Caching

Disadvantage

→ Application level (say if we have 10 applications, then for each application we need to setup proxy)

Reverse PROXY

From the internet, no request can talk directly to the servers

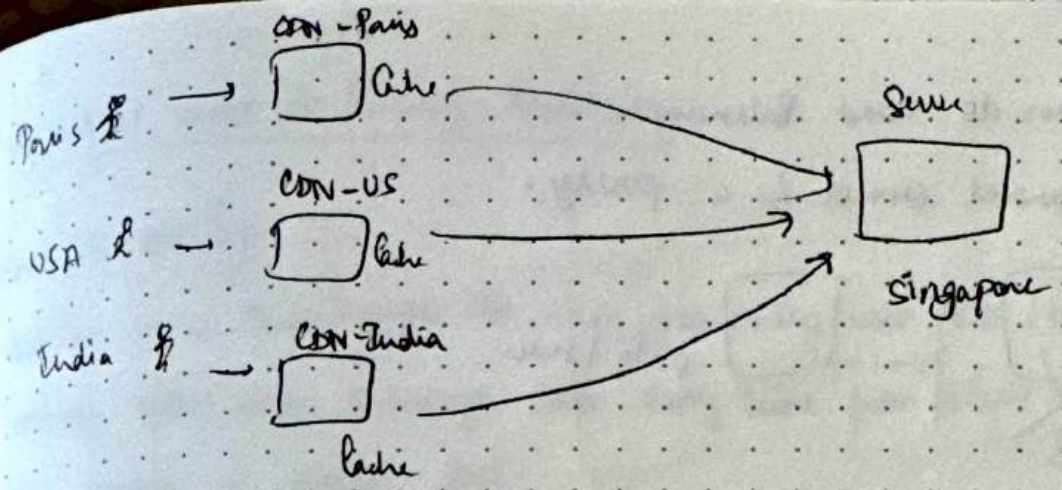


Advantages

(i) Security (Outside world is not aware of the server IP's).

(In case of DDOS attack, it can only attack the reverse proxy and not servers)

Eg :- CDN is nothing but a reverse proxy



(ii) Caching

(iii) Latency (increases performance by decreasing latency)

(iv) Load Balancing capability (It can do load-balancing as-well).

So, what is <u>Proxy vs VPN</u> ?

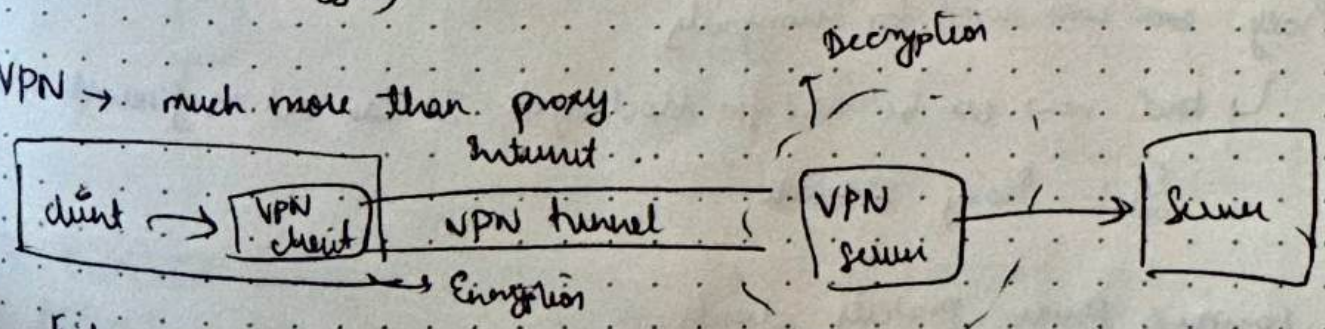
(i) Proxy acts as a by-pass

(anonymous) (IP address)

(caching)

(logger)

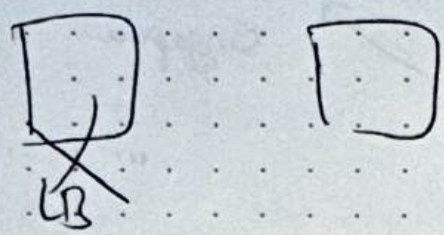
VPN → much more than proxy

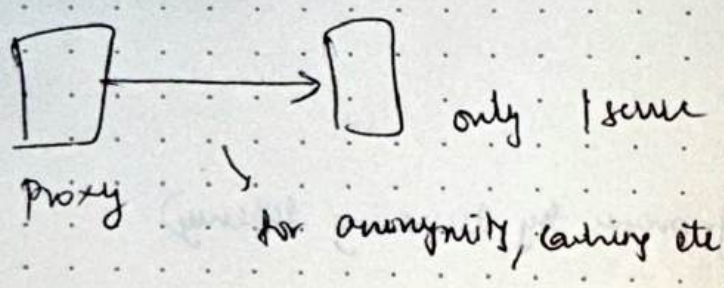


First client needs to connect to VPN client

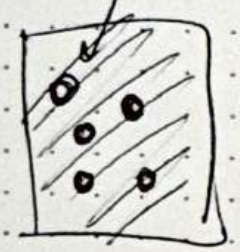
And VPN server does decryption and then sends to the server.

→ Reverse Proxy can do Load Balancing
but LB cannot cannot be a proxy.





→ Proxy vs Firewall



Works on packet scanning → headers
Port no, IP, destination

Proxy ~~can~~ was used for anonymity
↳ but now can be used for blocking etc. It can act as firewall also.
Eg :- Proxy Firewall.

→ Forward proxy protects clients.
→ Reverse proxy protects servers.

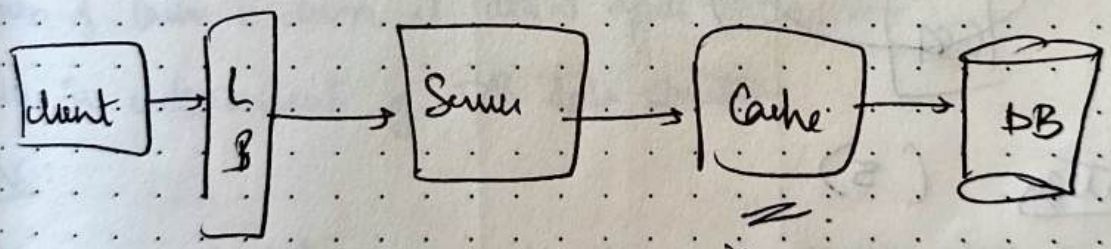
Distributed Cache & Caching Strategies

What is caching?

- Caching is a technique to store frequently used data in fast access memory rather than accessing data every time from slow access memory.
- This makes our system fast.
- It helps to reduce the latency.
- It also helps to achieve the fault tolerance. IP

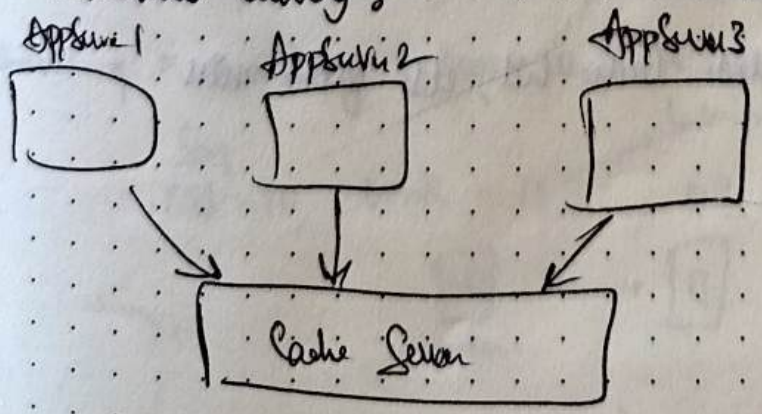
There are different types of caching present at different layers of the system like -

- client-side caching (browser caching)
- CDN (used to store static data)
- Load balancer
- Server-side application caching (like Redis etc.)] IP



sits between server and DB

What is Distributed Caching?

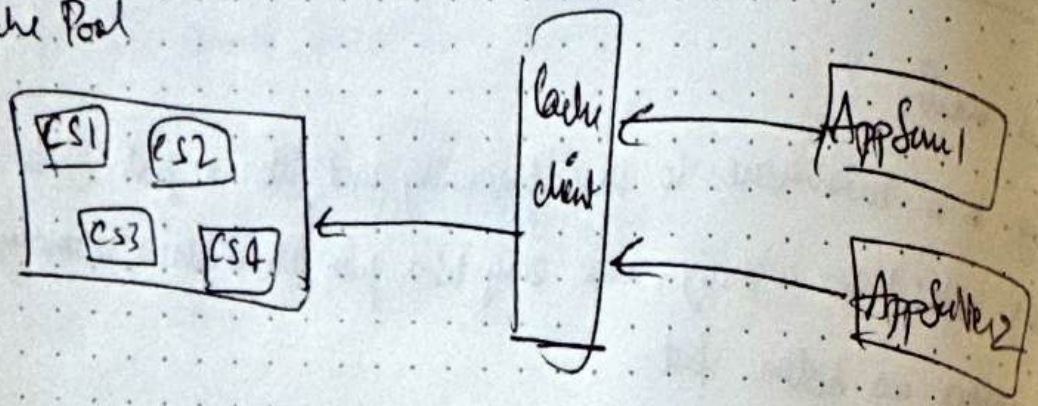


Limitations?

1. Single point of failure
2. Scalability is an issue

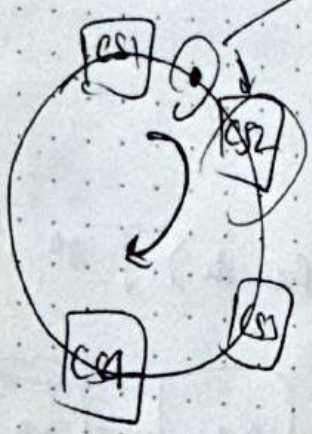
So,

Cache Pool



→ How one server is allotted?, it uses consistent Hashing technique.

In consistent hashing



Based on the hash, it'll get the point.

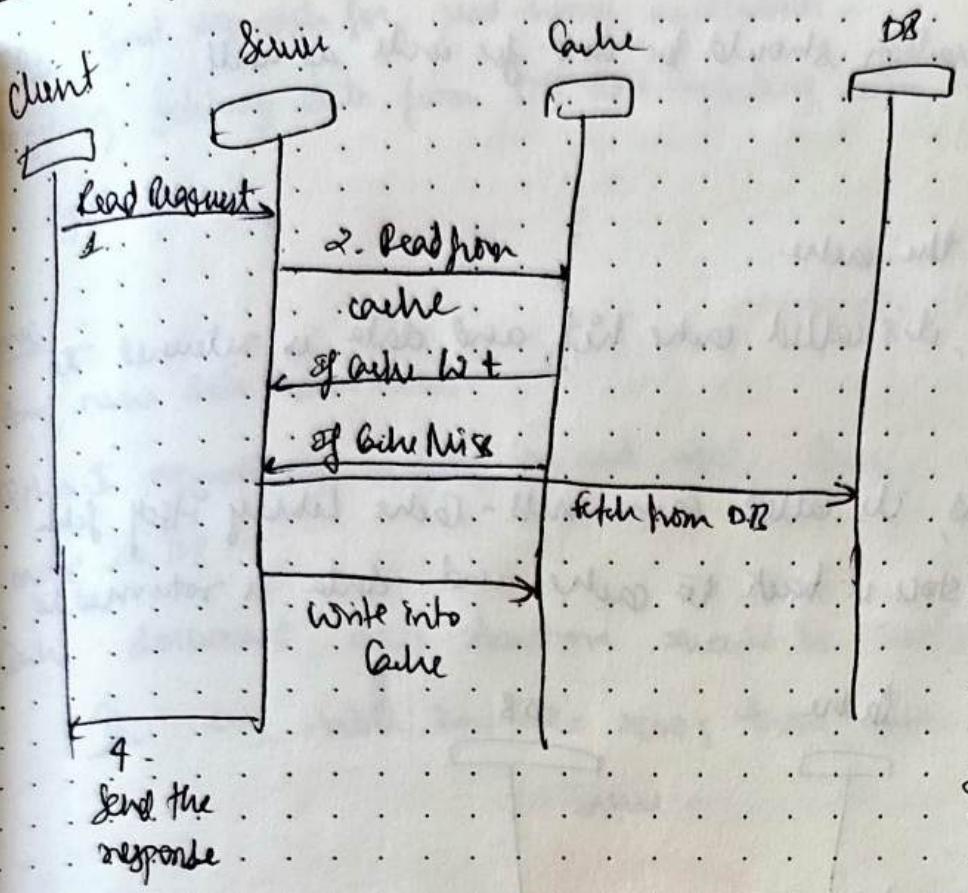
say we want to put ~~into CS1~~ data into a cache server from AppServer1.

Cache Strategy (5)

1. Cache Aside

→ Application first checks the cache

→ If data found, then called Cache hit, else cache miss.



* Write → directly writes to the DB

PROS

→ Good approach for heavy-read application.

→ Even if cache is down, it doesn't affect the system.

→ Cache is independent of DB data structure

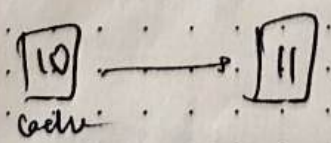
CONS:

→ For new read data, there will always be CACHE-MISS first (to resolve this, generally we can pre-heat the cache).

→ Without appropriate cache, not used during write operation, then there is a chance of inconsistency between cache & DB

Say
Id - 10, cache - 11 → but previously it had 10, so next next time it'll fetch 10.

DB



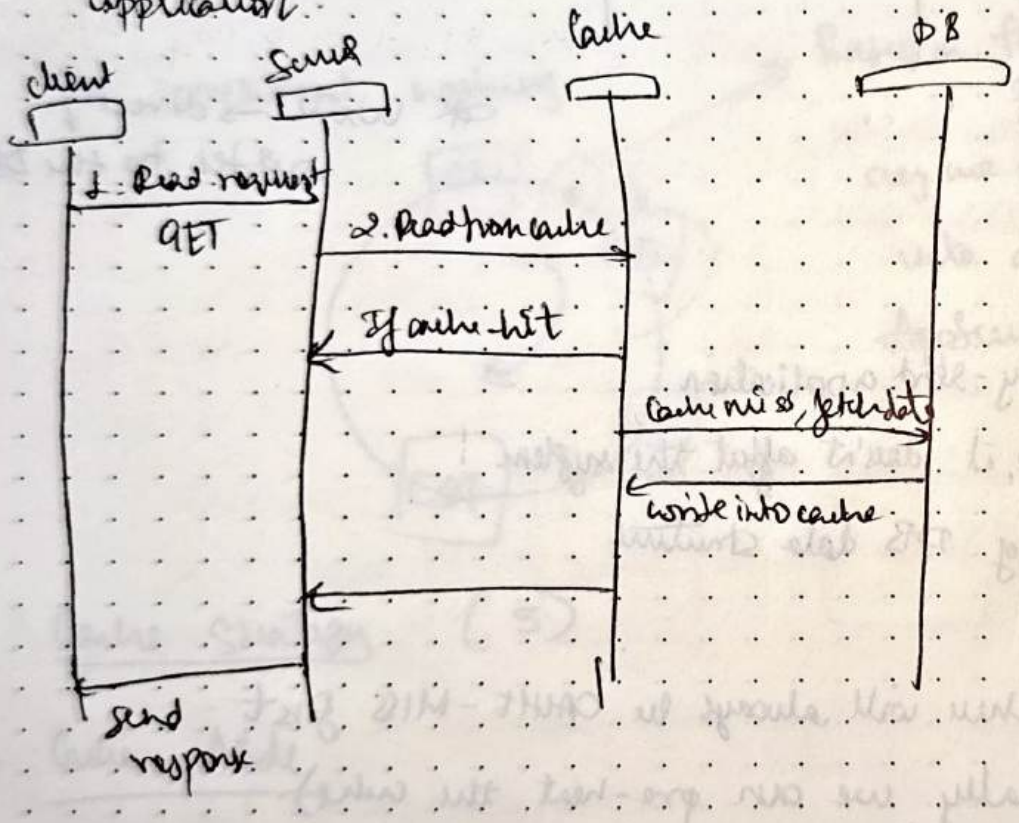
Hence, some caching strategy should be used for write as well

2. <u>Read through cache</u>

→ Application first checks the cache

→ If data is found in cache, it's called cache hit and data is returned to the client

→ If data is not found, its called cache miss. Cache Library itself fetch the data from the DB, store it back to cache and data is returned to application



PROS:- Good approach for read-heavy application.

→ Logic of fetching data from DB and updating cache is seperated from the application.

CONS:-

→ Any new data cache-miss.

→ Without appropriate caching is not used, There is chance of inconsistency b/w cache & DB.

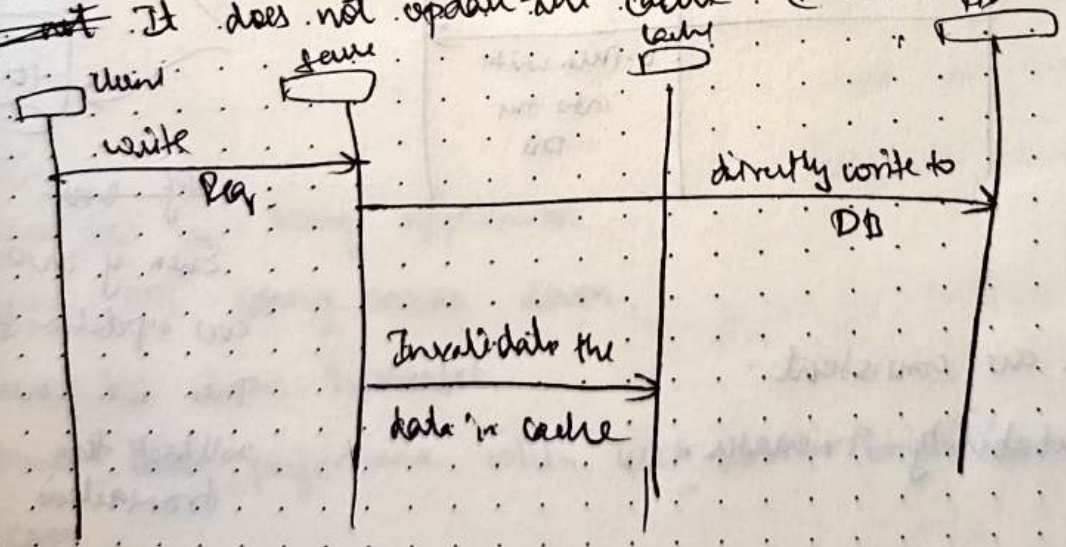
→ Cache document data-structure should be similar to the table now.

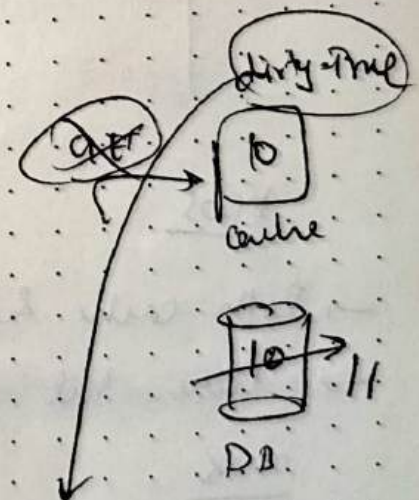
Eg:- say table has 20 rows then cache should also have the same.

3) Write Around Cache

→ Directly writes into the DB.

→ ~~If not~~ It does not update the cache (cache invalidate) (dirty)





then it'll read from the DB and put latest data.

PROS

→ Resolves inconsistency problem b/w Cache & DB.

CONS :-

→ For new data, cache-miss happens.

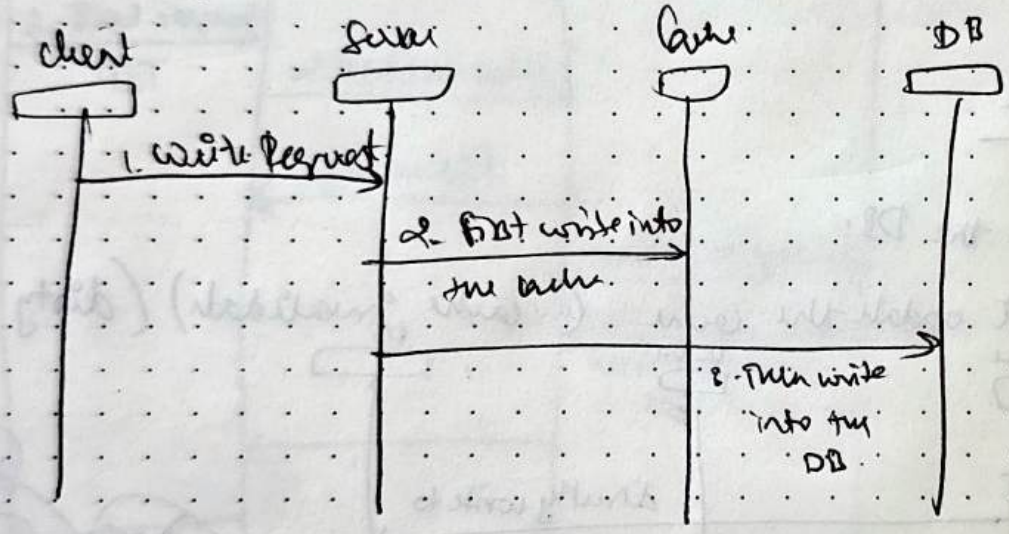
→ If DB is down, write operation will fail.

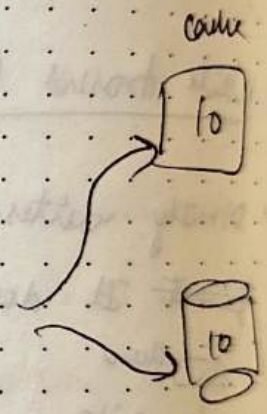
(So, write is not fault tolerant).

4) WRITE THROUGH CACHE

→ First write data into cache

→ Then in synchronous manner write to DB.





~~If one~~

Even if one of the operation fails Then we need to rollback the transaction

PROS

→ Both cache & DB are consistent.

→ Cache hit probability increases.

CONS

→ Slow as not useful. (Have to use it with some read strategy)

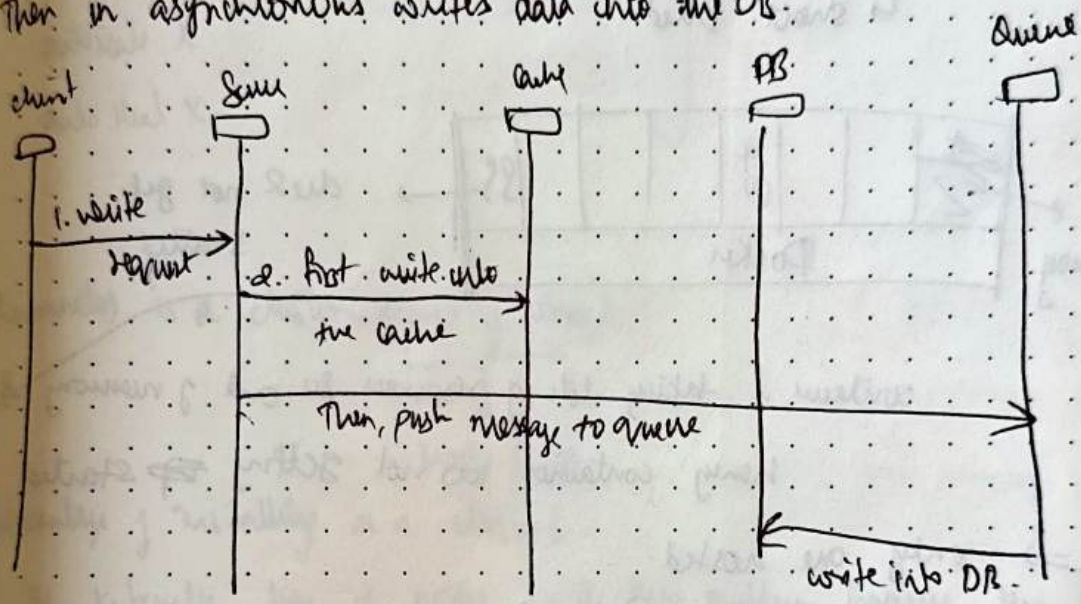
→ 2 phase commit, need to be implemented with this (Both succeed or both fail)

→ Not fully fault tolerant.

3) WRITE BACK CACHE

→ First write data into cache

→ Then in asynchronous writes data into the DB.



* Now, even if DB goes down we dont mind. As we have the stuff into cache/queue.

So fault tolerance is present.

PROS :-

→ Good for write heavy application.

→ Write API latency comes down.

→ Cache hit chance increases.

→ Much better performance when used with ~~write~~ cache aside / read through.

~~Cons~~

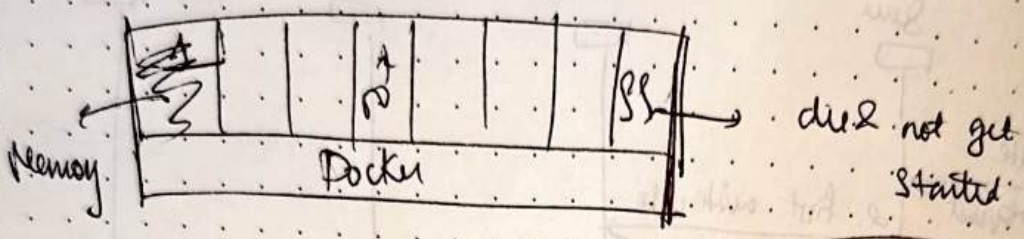
→ Even if DB ~~crash~~ fails, write operation still works.

* Cache has TTL → Cache Eviction Policy → least recently used (LRU)
→ FIFO - first in first out.

INTRODUCTION TO KUBERNETES

(1) Kubernetes is container orchestration platform.

Containers are ephemeral in nature
↳ short lived



container 1 taking lots of resources so out of memory & heavy container too not getting ~~up~~ started

Problem 1 → only one hosted.

→ Killed container
↓
App
↓
not accessible

Missing Problem 2
→ Auto healing

without manual intervention it should start automatically. But Docker doesn't give that

Problem 3 → Auto Scaling

→ load balancer feature missing in Docker.

→ Docker is minimalistic (or) simple platform

→ ↳ It does not provide enterprise level support

LB ✗

Firewall ✗

autoscale ✗

Auto Heal ✗

API Gateway ✗

→ Present in Kubernetes

Kubernetes is a cluster (group of nodes).

↓

Master / node

Advantage of installing as a cluster?

If Kubernetes has 2 nodes, and this problem happens, the Kubernetes puts them into a different host / node.

→ Kubernetes has replica set / replication controller, → auto-scaling / Load Balancer

K8's Architecture

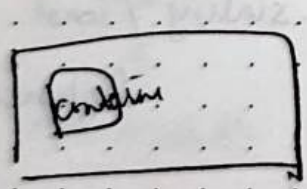
Why Kubernetes is called as K8?

Kubernetes has control Plane & Data Plane.

Control Plane	Data Plan
→ API Server	→ Kubelet
→ etcd	→ proxy
→ scheduler	→ container run-time
→ control manager	
→ cloud controller manager	

Docker → simplest thing is container

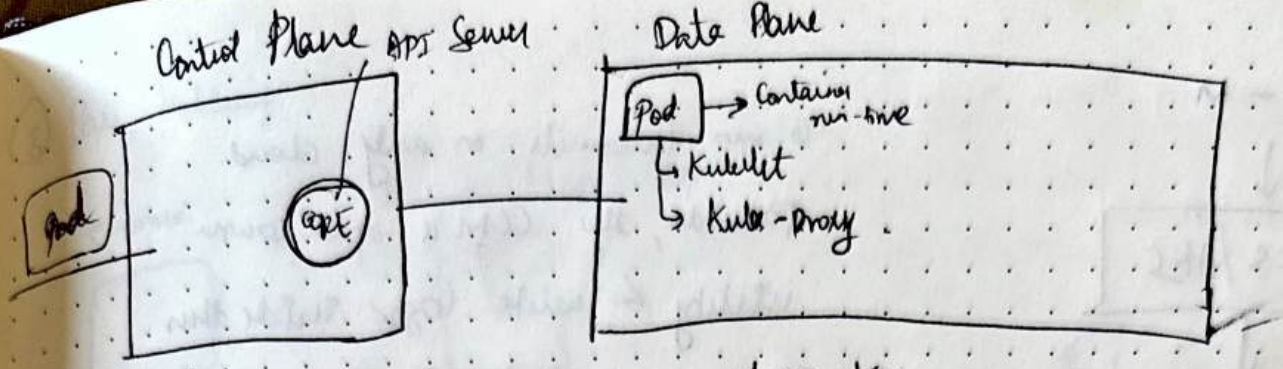
In Kubernetes simplest thing is pods.



Docker

without container run-time, container won't run. Just like Java won't run if Java runtime is missing.

container runtime → Docker runtime



1 master

1 worker

Below components are present in worker node

→ Kubelet responsible for running the pod; if pod not running then this makes sure it runs.

→ Kube-proxy → provides n/w, IP addresses & LB capabilities.

Why do we need a control plane?

→ Who will decide on which node the pod has to be deployed etc?

It takes all the requests.

→ API Server → Exposes our kubernetes (heart) → (decides)

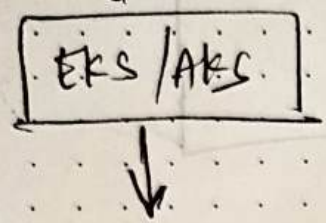
→ Scheduler → responsible for scheduling pods/resource → (takes action).

→ etcd → (key value store) → entire cluster info stored as K-V / objects here.

→ controller manager → To support auto-scaling, Kubernetes has some controllers like replica-set.

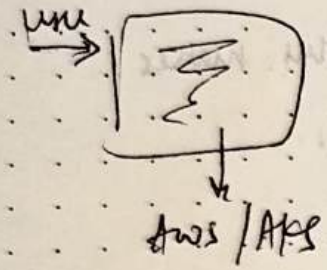
CM → controller
↳ replica set

C-C-M



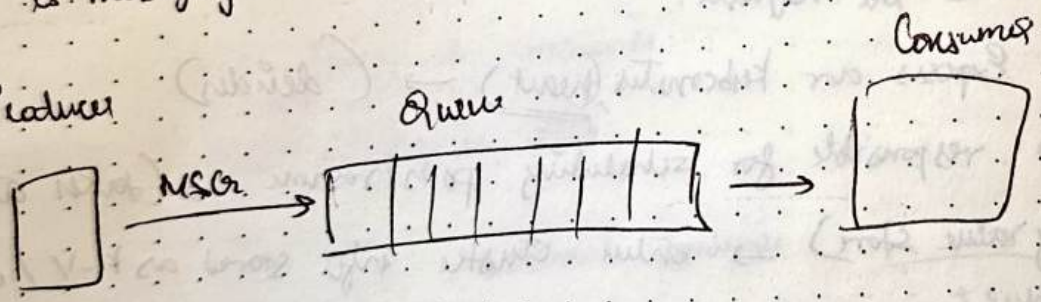
To run kubernetes on any cloud provider, the CCM is open source utility to write logic inside this to support kubernetes in that cloud-provider.

Load storage

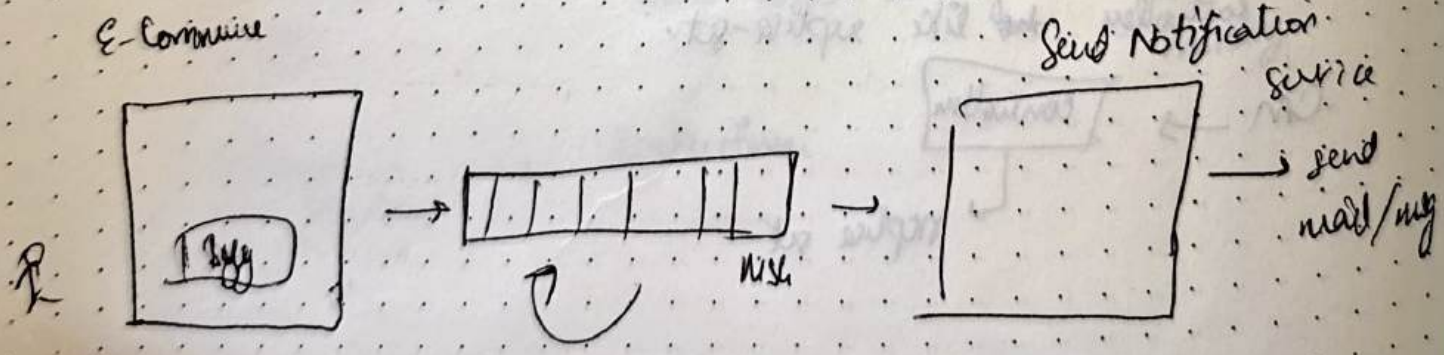


System Design - Distributed Messaging Queue (like Kafka, RabbitMQ)

What is messaging queue?

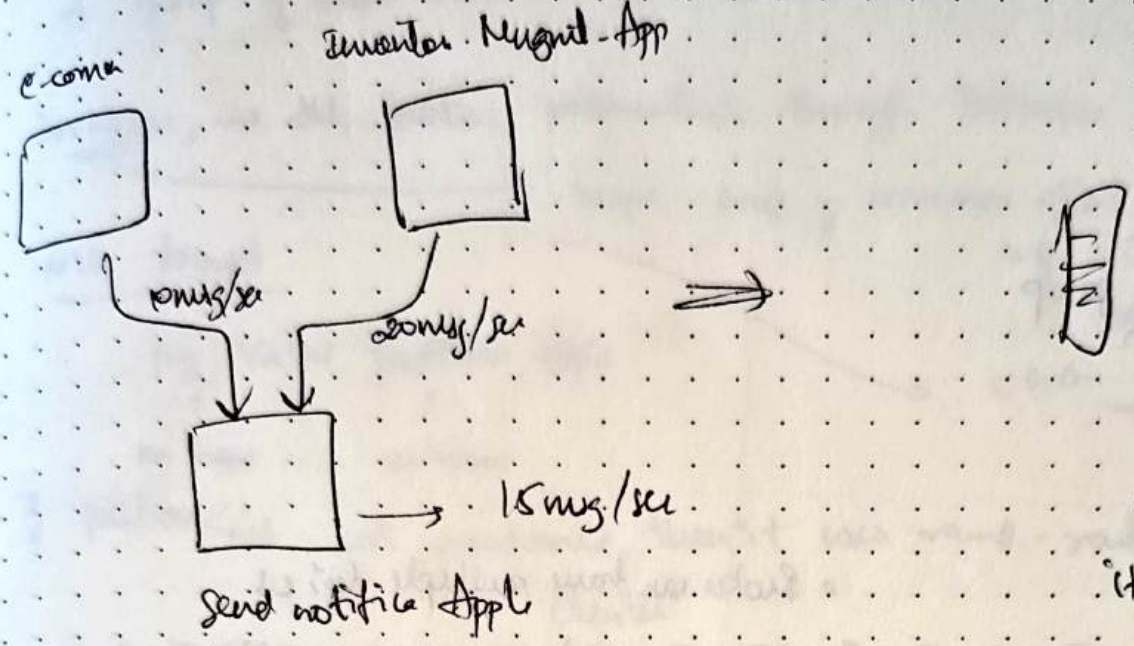


(1) Use case :- Asynchronous nature

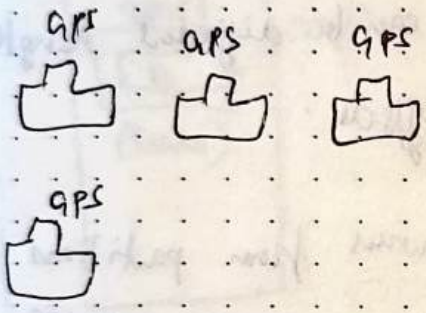


(2) Queue gives us retry capability

③ Pace Matching

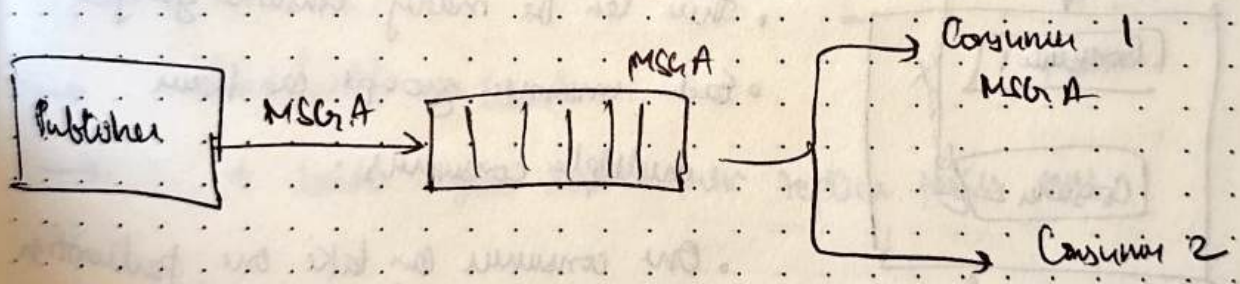


so instead we put in a queue & consumer will consume it at its own pace.

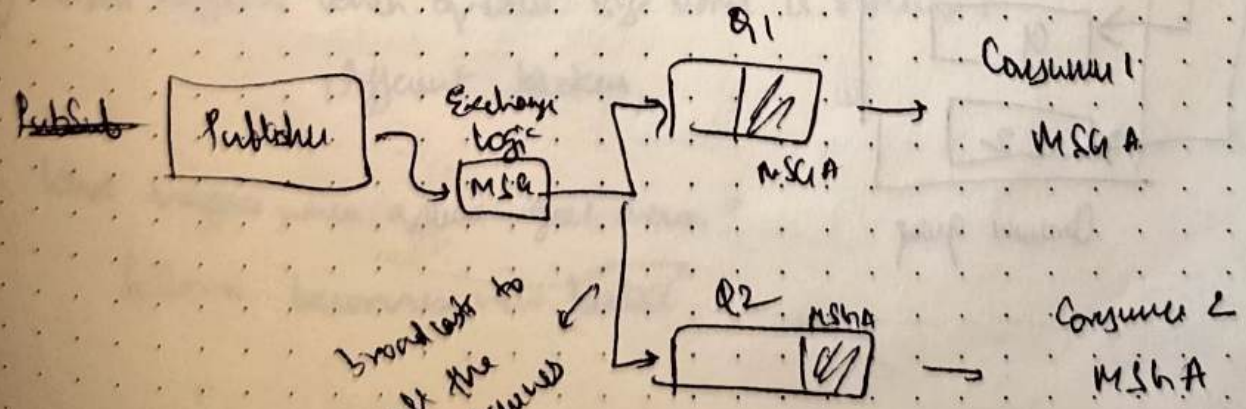
④ 

10/sec every car sends its carId & location.

Consumer cannot consume in such pace hence a ~~consumer~~ queue is required.



P2P service, message will be processed only once. But in PubSub



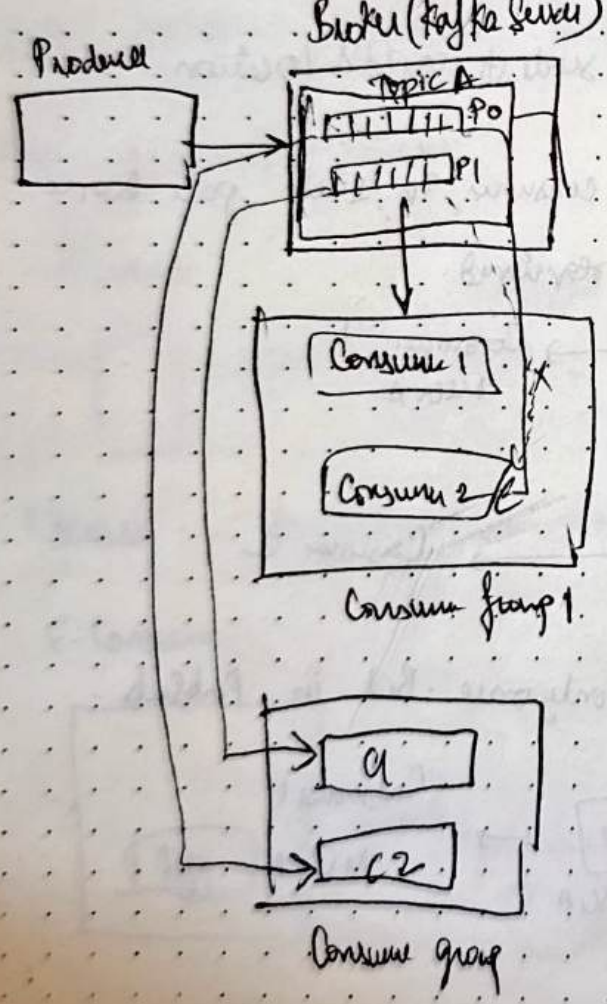
How messaging Queue works?

① KAFKA

- Producer
- Consumer
- Consumer group
- partition
- Offset
- Broker
- Cluster
- Zookeper
- Topic

- Broker can have multiple topics.
- Topics can have many partitions

different partition can be different length and offsets.



- Consumer consumes from partitions.
- There can be many consumer groups.
- Each consumer groups can have multiple consumers.
- One consumer can take one partition inside a consumer group.

Different broker means different nodes (other machines).

A group of these machines is called as cluster.

Zookeeper, → All brokers communicate through Zookeeper

→ keeps track of consumer offset.

usy = 3, meaning consumer has read till 3.

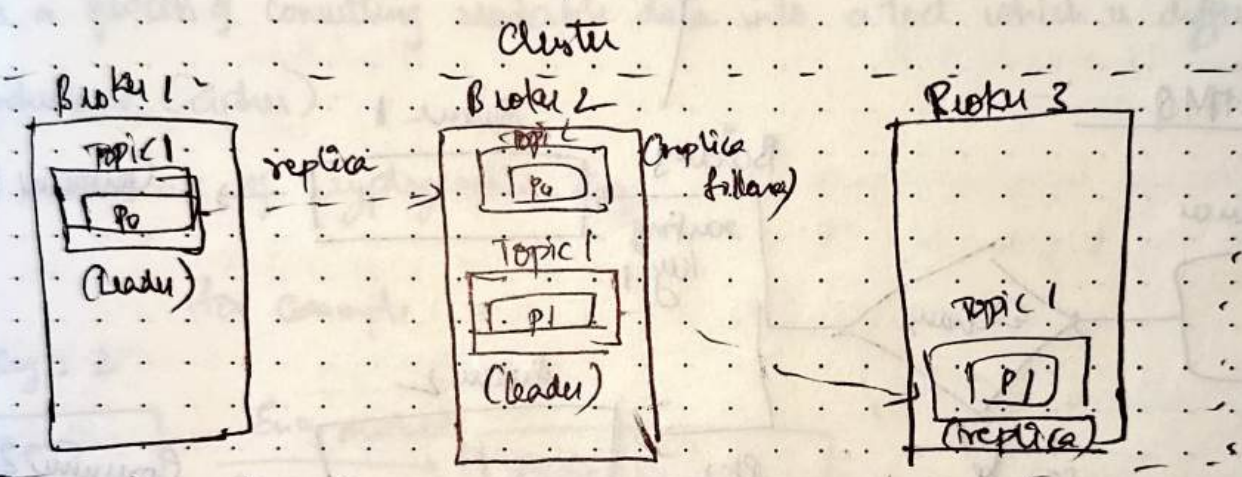
MSG format

key, value, partition, topic.

?? not mand ?? not mand

→ CGA → which consumer group to pick.

If partition is not mentioned, then it uses round-robin fashion.



→ Read/write → Leader

→ If leader goes down then replica takes over

→ And leader & followers are synced

(4) What happens when queue size limit is reached?

Different brokers,

5. What happens when queue goes down?

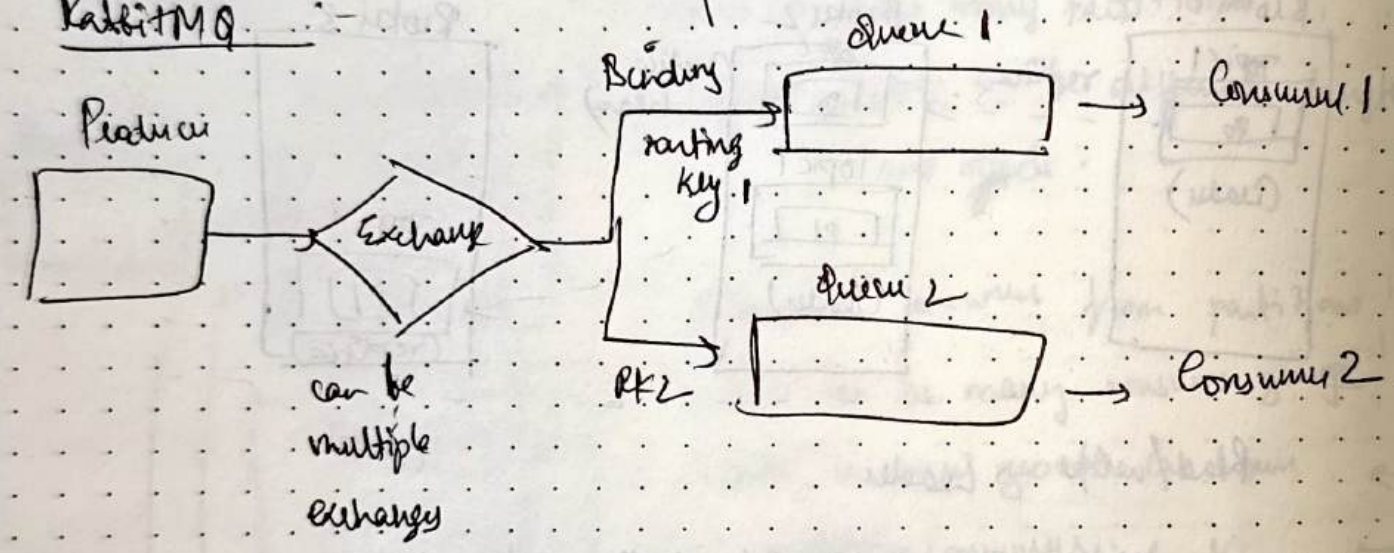
follower becomes new leader

→ What happens when consumer goes down?

Another consumer takes over.

RabbitMQ	Kafka
Push-based approach	Pull-based approach

RabbitMQ :-



Exchange

① FanOut exchange (push the message to all queues associated with the exchange)

② Direct Exchange (Exactly matching)
(Message key & routing key)

③ Topic Exchange [wild card]

X There is no concept of offset

→ If a consumer is not able to process them, it will put back to the queue to reprocess.

(or) can put back to the dead queue.

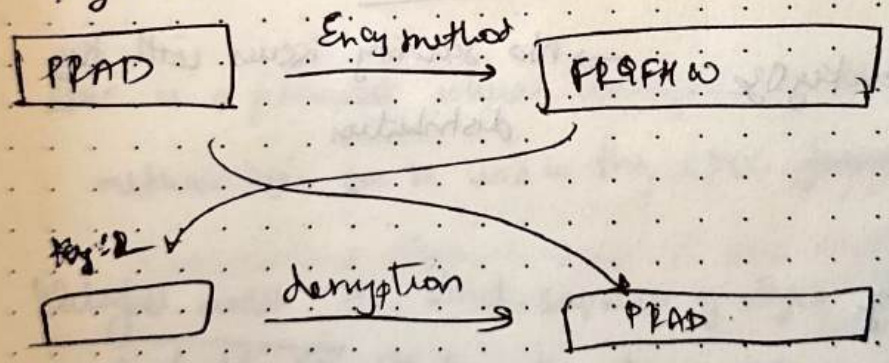
Cryptography

Encryption

- It's a process of converting readable data into a text which is difficult to understand (cipher).
- It makes use of cryptographic key.

for example

Key : 2



Encryption

Symmetric Encryption

→ Same key used for Enc & dec.

→ Enc & Dec. algorithms are :-

- DES (not recommended)
  - → fixed key length : 56 bits
  - • Process data in 64 bits
  - • Perform 16 rounds in the encryption process.
- AES (Advanced Encryption standard)
  - → Support key length : 128, 192 (or) 256 bits
  - → Process data in 128 bits
  - → Encryption round depends on key size
    - 10 rounds → 128 bits key
    - 12 rounds → 192 bits key
    - 14 rounds → 256 bits key

Advantages

Fast & less computation.

Good for encrypting bulk data.

Disadvantages

Key distribution i.e how to securely dist key b/w client & server.

Asymmetric encryption

→ Uses 2 diff. keys private & public

- public key is used by sender to encrypt.
- private key is used by receiver to decrypt.

Algorithms

→ RSA 2048 bits key length

→ DSA

→ Diffie Hellman

Advantages

→ No security issues with key distribution

SDLC (Software development lifecycle)

It is a structured process used by Software development teams. The objective is to ensure the high-quality software to develop in a systematic & cost-effective manner.

7 Stages of SDLC are

(1.) Planning
(2) Requirement gathering & analysis
(3) System Design
(4) Implementation (or coding)
(5) Testing
(6) Deployment
(7) Maintainence

SDLC is a framework, whereas Waterfall is a methodology. Different methodologies can be used in the SDLC framework.

Waterfall model → strict sequence of stages, linear approach to development. Inflexible, difficult to make change after a stage is completed. Feedback is typically at the end of the project.

Agile → focuses on delivering small incremental improvements & responding quickly to changes in requirements.

Key Principles of Agile

Agile Manifesto

1. Individuals & Interactions
2. Delivering software freq., in sprints
3. Continuous feedback from stakeholders / customers
4. Collaboration b/w cross-functional teams
5.

Agile process

Project planning	Development	Sprint Review	release & deployment
Sprint planning	Testing	Sprint retrospective	repeat

Code Reviews

1. Logical errors / incorrect handling of edge cases
2. Code Quality → Ensure that code is clean, readable and adheres to the team's coding standard.
3. Performance / Security → Check if there are any bottlenecks / inefficient implementation → (a) Potential security vulnerability
4. Maintainability → Ensure code is well-structured and easy to understand for future developers
5. Also check for code coverage (test coverage), to make sure new code is adequately tested with UT's, integration tests etc.