System Design

(Beginner's Version).

Design Requirement

Availability = $\frac{\text{uptime}}{\text{uptime + downtime}}$

* Measured in term of 9, like
  2 (9's) ⇒ 99% availability
  3 (9's) ⇒ 99.9% ,,

99.999 % is considered really good, because after it, improvement is of about .5 mins.

* generally defines Service Level Objectives (SLO)

SLA (expected Availability) ≠ SLO (Actual)

Reliability :- * P(system won't fail)

* Can't be calculated
* Like single server failing is more probable than double servers.

* Horizontal scaling is more benificial when compared to vertical scaling

fault tolerance :-

If one server stops to work, will our system continues to serve requests or not?

or

if a part of the system goes down, will our system continues to function or not?

Redundancy :-

Something that is not entirely necessary.

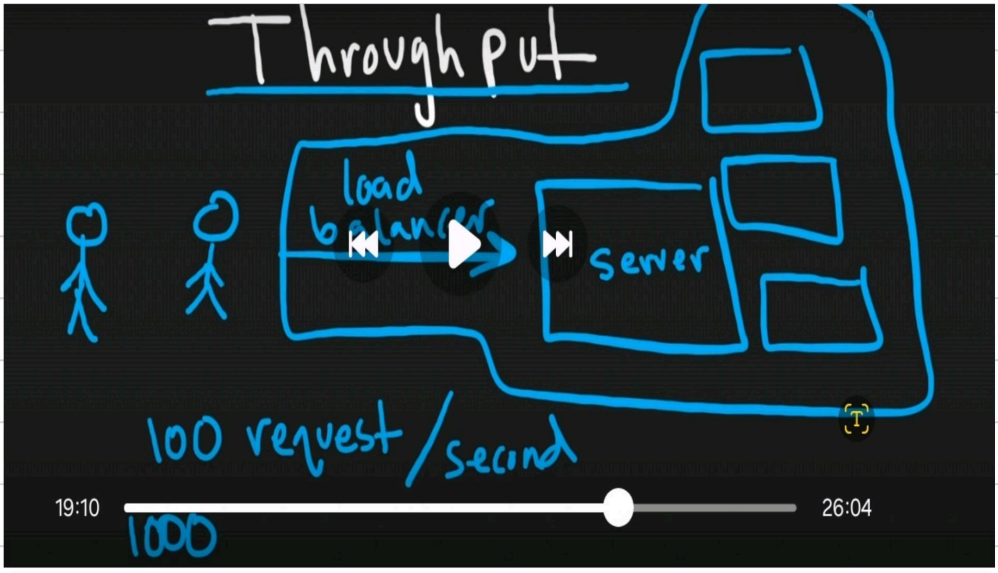
Suppose :- one server is enough to run our code but we have added another server so in that case the second server is redundant.

But it is generally good to have, because in case of a fault, we will have multiple copies.

Through put :-

Number of concurrent requests our system is able to handle per second.

Generally, we do horizontal scaling to increase the through put, but the downside is, we require to balance it using some load balancer which turns to be complicated



But horizontal scaling becomes even more challenging in case of a database because then some data will be at one place, other at another.

QPS (Queries/second) :- Number of queries our database can handle.

Bytes/ Second :- Used mainly in case of a data pipeline

Latency :- End to end :- Total time taken to complete from user to server and then may be back to the user

much better

To reduce latency, generally the servers are set at the different parts of the world

or

Using some CDN.

Network Basics

IP - Adress :- Address to uniquely identify machines.

IPv4 {
- * 32 bit
- * 12 integers, seprated by 3 dots. like : 012.345.678.910
- * Biggest 256.256.256.258

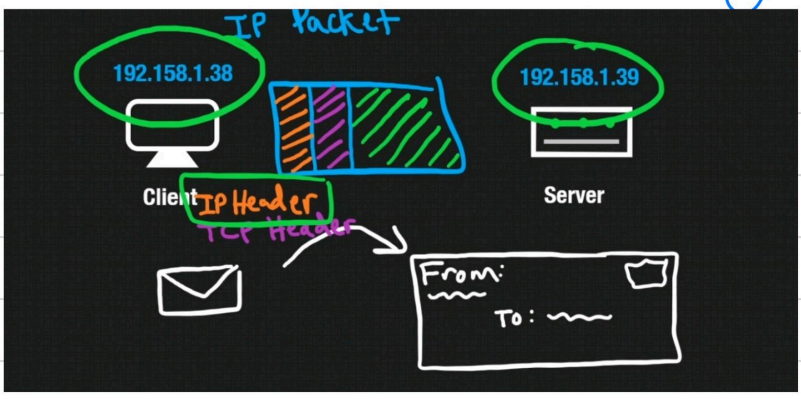
} At max 10^9 address possible.

IPv6 {
- * 128 bit

}

data is sent in form of packets where it contains :-

(i) data
(ii) sender's IP
(iii) Receiver's IP } Meta-data / Header



Public IP address :- for a server to be publicly accessible, it should have a public IP.

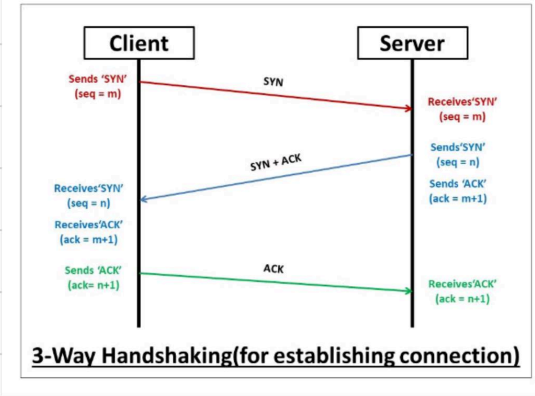
Private IP

Port :) single application on each port.

TCP & UDP

TCP

* ~~No~~ Order is maintained
* Reliable
* Retransmission of lost package. If package not arrived, then re-send it
* 2 way 3 way handshake
* Slower



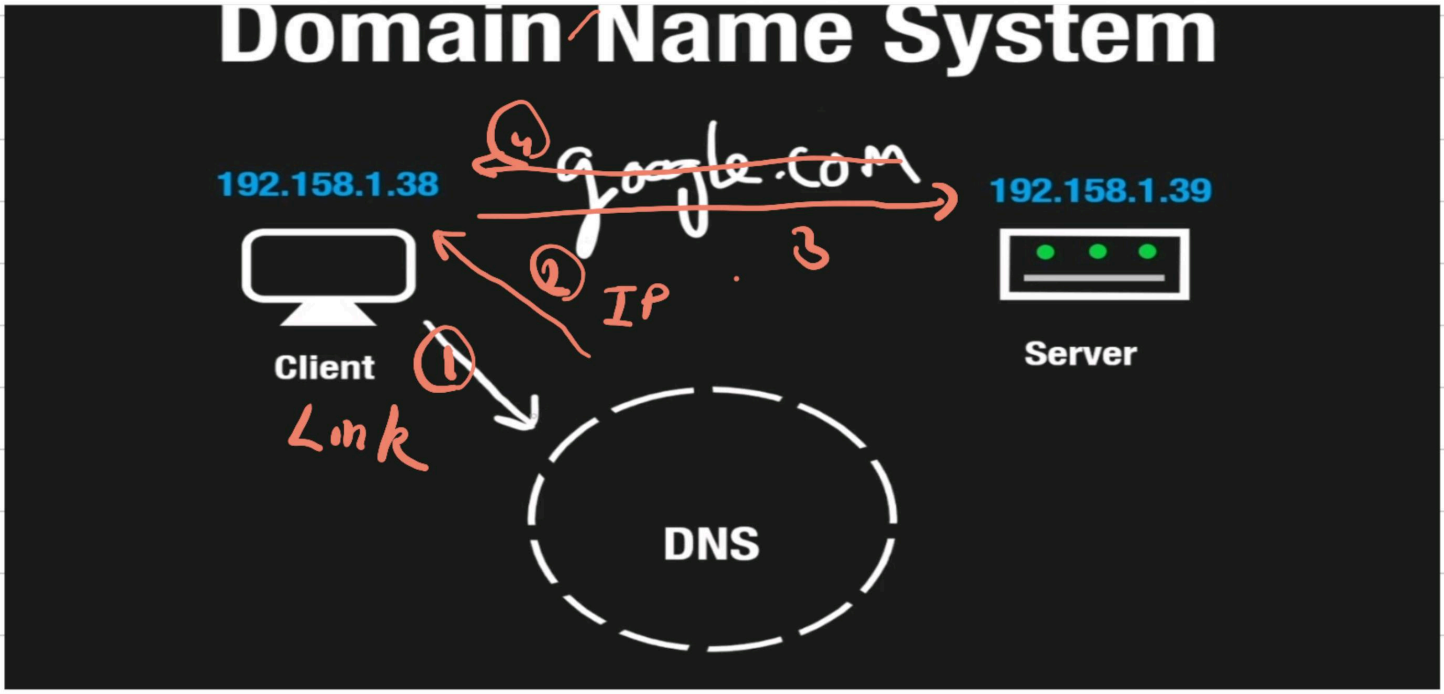
3-Way Handshaking(for establishing connection)

UDP

* Not maintained
* Non-reliable.
* Nothing as such
* One way
* faster
* Live streaming, we might loose one or two video frames.
* Also, used in DNS.

DNS

DNS



1 → Link enter
2 → Ask DNS for the IP
3 → Go to IP
4 → Get the response

ICANN ⇒ Internet Corp. for assigned names and numbers.

* Owner of all domains in the world.

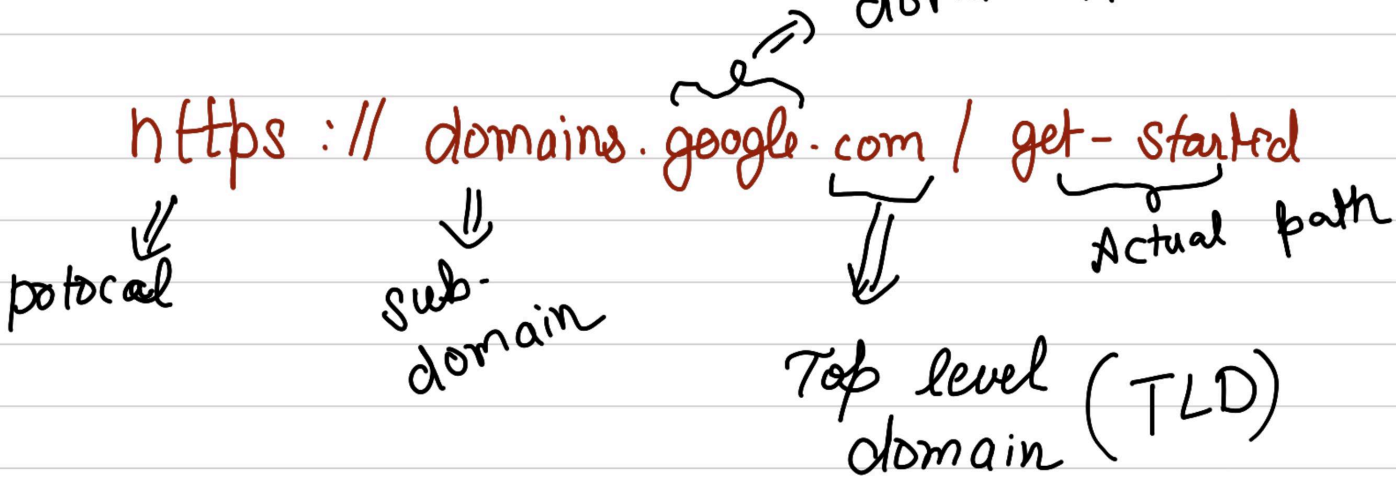
DNS records stores the info. about the domain names.

The IP address can be both static or dynamic.

But in general, IP addresses do not change very often, therefore they are cached. in the users computer.

Facts

firewall on the public server allows the req. to reach and ask.
but do not allow in case of private servers.



www is also a sub-domain

Application Protocols

Client may be a server.

RPC :- Remote procedure call

* When we req. to move data b/w machines. which data we are req. to move. So, this is done via an RPC.
* Like a fn call.
* Think, the code is on the server, we are req. to execute it on the server. So, we call it through an RPC.
* Since, loosely def^{n}, most of the network calls are considered as RPC.

HTTP

* Request response protocol
* Built on the TCP.

TCP
UDP

} discussed Earlier.

SMTP, FTP, web-socket, SSH.

Various Methods :-

Get

=> Return some data
=> Endpoint
=> Retrieve Info.
=> Do not have body. So include info in the header or the url.
=> More cache prone.

Post

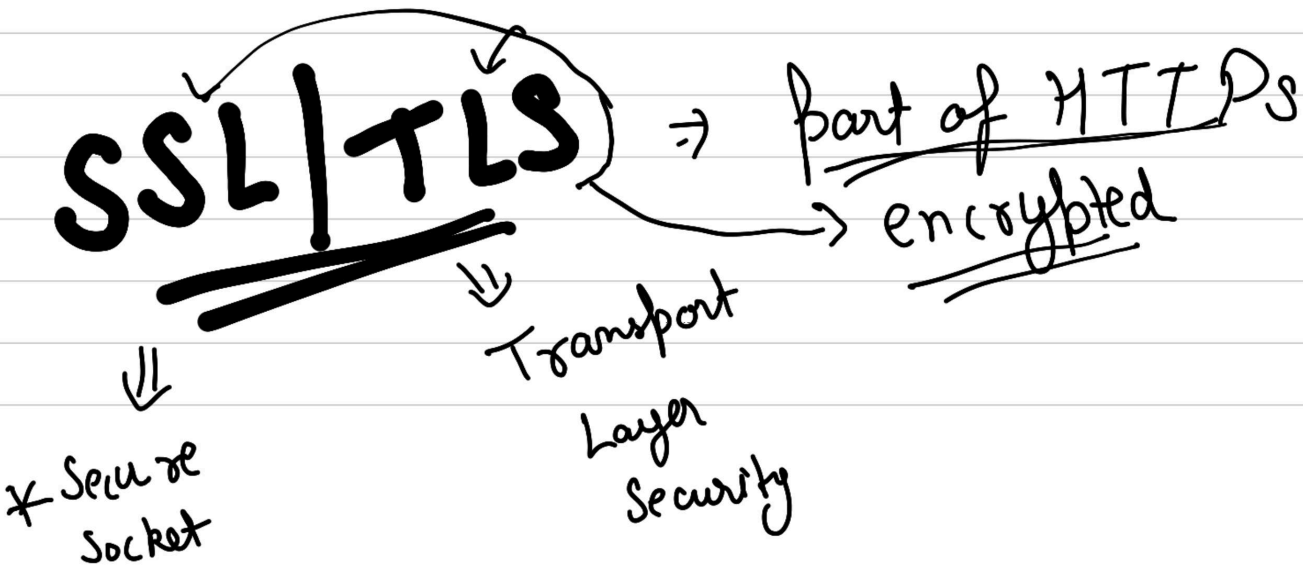
=> Do something like creating a user account.
=> End point
=> Creating an info.

Put :) update

Delete :) delete something
:) Idempotent.

Status Codes:

Informational responses	(100 – 199)
Successful responses	(200 – 299)
Redirection messages	(300 – 399)
Client error responses	(400 – 499)
Server error responses	(500 – 599)



Layer

* Old

* New

HTTP is not really secure in case of an MITM (Man in the middle) attack.

Limitations of HTTP :-

* Can't be used for live chat. Only way to make it possible through HTTP is polling where we continously send a req. after a fixed interval.

*

Web-socket

* for chatting.

* Steps :-
  (i) Set up HTTP
  (ii) Upgrade to web socket using

the 101 Ack. from server.

* Both dirn

* Made obselete after the HTTP v.2 which uses streaming method. but are still used :)

API :- Application Programming Interface

API Paradigms

Most Commonly used.

Rest

GraphQL

gRPC

ReST

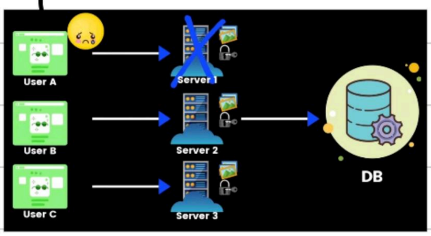
* Representational State Transfer
* Loose restriction which are applied

to HTTP.

* Scalable
* JSON (data format)
* Commonly used
* Stateless
* Closely aligns to HTTP

Stateful (data)

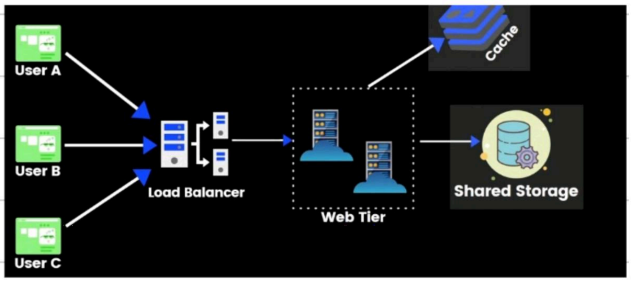
* State of user is maintained in the local server, but not in the D.B. so now even if one server goes down, the user will loose all its session info.



* Availability issues if one server crashes.
* Scalability. (neither horizontally)

Stateless

* State maintained in the shared storage or cache / DB.



Graph QL

* Built on HTTP
* Uses only 'post' request in whose body, the query is included.
* Query tells which fields do we want.
* Solves problems like overfetching, underfetching.
* Single Endpoint → Query
  → Mutation.
* Can't cache Graph QL

```
{
  launchesPast(limit: 10) {
    mission_name
    launch_date_local
    launch_site {
      site_name_long
    }
    links {
      article_link
      video_link
    }
    ships {
      name
      home_port
      image
    }
    rocket {
      rocket_name
      first_stage {
        cores {
          flight
          core {
            reuse_count
            status
          }
        }
      }
      second_stage {
        payloads {
          payload_type
          payload_mass_kg
          payload_mass_lbs
        }
      }
```

← Sample Graph QL Query

* Schema is req. to be followed

gRPC

* Built on HTTP 2.

* → Can't be directly used by the browser
  → proxy is needed.
  like gRPC-web
  → because gRPC require HTTP 2 which can't be directly accessed by browser

* Info is sent in form of protocol buffers.
  Schema's.

* Data is sent in binary formal and typically lesser packet size is req. to send same data, when compared to ReST.

* Powerful
  → Do things like streaming (both one way and 2-ways).

* Newer
  but
  less tools available
  Harder
  but takes less time.

* No status code but have error messages.

Defining A Message Type

First let's look at a very simple example. Let's say you want to define a search request message format, where each search request has a query string, the particular page of results you are interested in, and a number of results per page. Here's the `.proto` file you use to define the message type.

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

API - Design

Operation	Rest	RPC
Create	Post	Create
Read	Get	Read
Update	Put	update
Delete	delete	delete.

Example :-

https://api.twitter.com/v1.0/tweet- POST

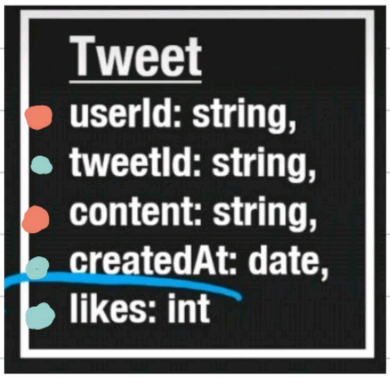
⇒ version

So, our API will take, these three as a parameter.

createTweet(userID, content, ParentID=null)

Optional parameters are used in general to make the newer versions of an API, backward compatible

But sometimes, it is not possible to make the API's backward compatible, in that case versioning comes into the picture.



- Required
- Can be crated on the server directly.

To read the tweets, we may use :-

https://api.twitter.com/v1.0/tweet/:Id - GET

This id helps us to get tweets one by one, else user's mobile won't be able to handle likes 100,000's of tweets simultaneously.

Moreover, many times pagination, limit, offset is done in order to solve the above issue

https://api.twitter.com/v1.0/users/:id/tweets?limit=10&offset=0

user ID

we want tweets.

Query parameter.

Get requests are not used to update something on the server, because they are cached by default. Therefore, every time the value will change.

Caching

CPU can read fastest from cache.

Objective
- :- Increase throughput
- :- Lower the Latency
- :- reduce the network cost.

Done by moving some data from primary storage to cache.

like suppose, I open a site very freq. and it is static in nature. So, instead of making network calls every time, the browser will cache the results of the older calls.

Header has
cache-control :- max-control-3600

⇓

The content can be cached for at max 3600 sec.

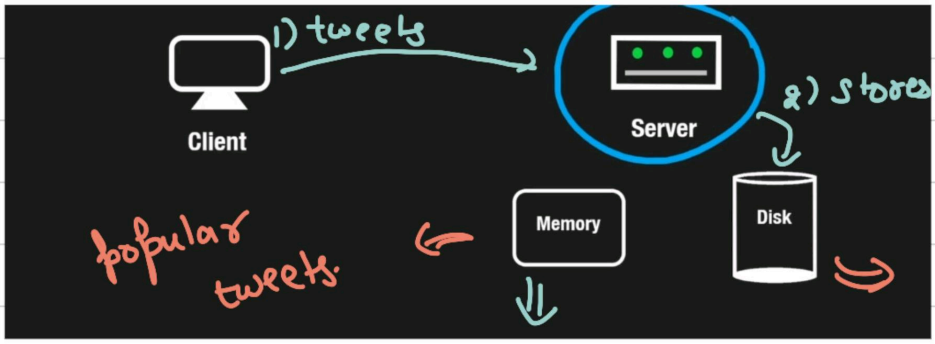
Data Disk >>> Data network.

Cache hit ⇒ if (cache[data] != null)

Cache miss ⇒ if (cache [data] == null or cache (data) is expired)

Cache ratio :- $\frac{hit}{hit + miss}$

Caching from the server's prespective :-



Key-value pairs
(Redis) { database inside the memory }

Suppose, user wants to search for a tweet inside the D.B, now this get Tweet can be cached.

Example :- No one is going to read my tweets (So store in disk, because no. of req. ≈ 0)
But, for Narendra Modi, no. of reader will be high, so high no. of get requests. So, store in the memory as well as disk.

write-around cache :- skip the cache entirely in the starting.

Disk — when we read, if cache

memory.

miss, then find in the disk and write it in the cache and return to the user.

write - through cache :-

Write in memory / cache first and then inside the disk.

write-back cache :-

might be used for the case of liking and disliking tweets.

* Lazy way
* faster when possible
* But less reliable.

* Store only into the memory not inside the disk.

Problem * fails in case of power supply out rage because ram are volatile

Soln * So, periodic cycles, the data is also written to the disk.

* Used, where we can afford to loose some amount of data.

Eviction Policy

* Cache limited in size
* So, what to do when the cache is full ?

* Can do randomly or
* Objetive

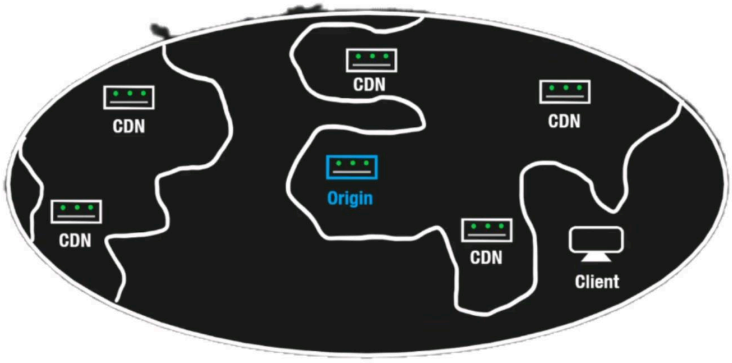
* Maximize cache hit ratio.

* FIFO :- first in first out.

More preferred * LRU :- Least Recently used

* LFU :- least frequently used.

CDN's



Content delievery Network.

Used to reduce network latency. by making copies of content (image/video) / static code and placing them at various locations in the world.

Can't put everything on the CDN servers
→ only static content can be stored here.
→ But these days Edge servers are getting introduced, which can run code on them.

Types of CDN :-

Push	Pull.
* Origin updates the data in the CDN.	* In case of a miss, CDN will call origin for data,

get it and cache it.

* E.g
  * User changes D.P.
  * update DP at origin
  * origin pushes the same change to all the CDN's.

* E.g
  Match streaming (India vs Pak)
  Now, only CDN near India will get the most req. So it will pull the same from origin and store it with him.

* Cache control : public
  ↳ The file is static in nature and can be cachable.

Proxies & Load Balancer

Proxy (forward)

⇒ A server which acts as a middle layer.

⇒ Hide client from the main server.

⇒ Proxy hides user's info.

⇒ Used to bypass network restriction.
like suppose a website is blocked,

we will access proxy. Proxy will access the website and return the content to us.

⇒ Proxies are also used to block some website like in the case of a school/corporate network.

Reverse Proxy :-

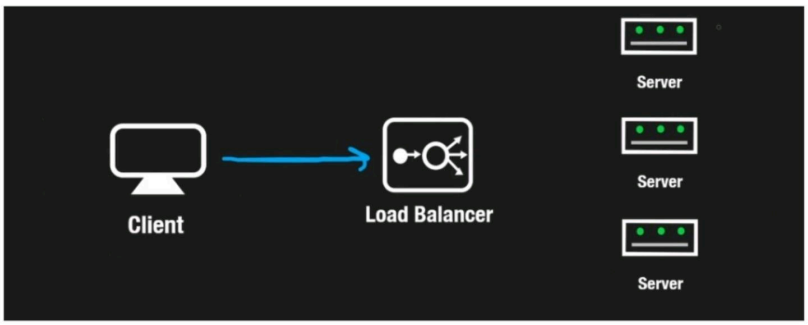
* Hides server from the client.
* But server will know about client.
* Eg. CDN and Load Balancers.

Load Balancers ⇓

Selects which server will serve our request.

⇓

So, that every server will have eq. amount of load.



⇓⇓

Implement with REST

⇓⇓

weighted ← .

Ways to implement the load Balancer.

- Built using wts Round Robin. method
  Or
- Server with least no. of connections
  Or
- Server nearest to the client.
  Or
- Using the Hashing.

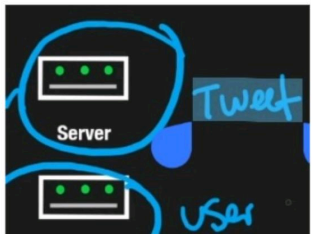
Types of load Balancer

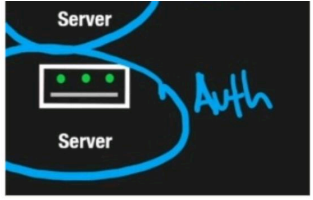
Layer 4 (TCP)

→ Location based
or
Round Robin.

Layer 7
(Application Layer)

→ Based on type of request.





→ More powerful but also expensive.

What if our one and only load Balancer goes down?

So have multiple / backup load balancer. But generally do not do not do much and hence won't go down.

Open Source Load Balancer :-

Nginx

Hashing.

Same as implementation of maps.

Suppose we have 'n' servers numbered from 0 to n-1. Now, take the MOD of the IP with N, and route the request accordingly.

So, a user always approaches a same server.

→ Benifit

↳ when the servers are stateful and then they won't req. to share their cache with one another.

→ loose

↳ It might be that user won't be going to the server with least amount of load.

↳ or., one server crashed.

Now. 3 Requests ⇒ {9, 10, 11}

Initially (we had 3 servers)	Now, we have 2. (Server 2 crashed)
9 %. 3 ⇒ Gone to 0	9 %. 2 ⇒ Goes to 1
10 %. 3 ⇒ Gone to 1	10 %. 2 ⇒ Goes to 0
11 %. 3 ⇒ Gone to 2.	11 %. 2 ⇒ Goes to 1

Servers changed.

So, they lost their data.

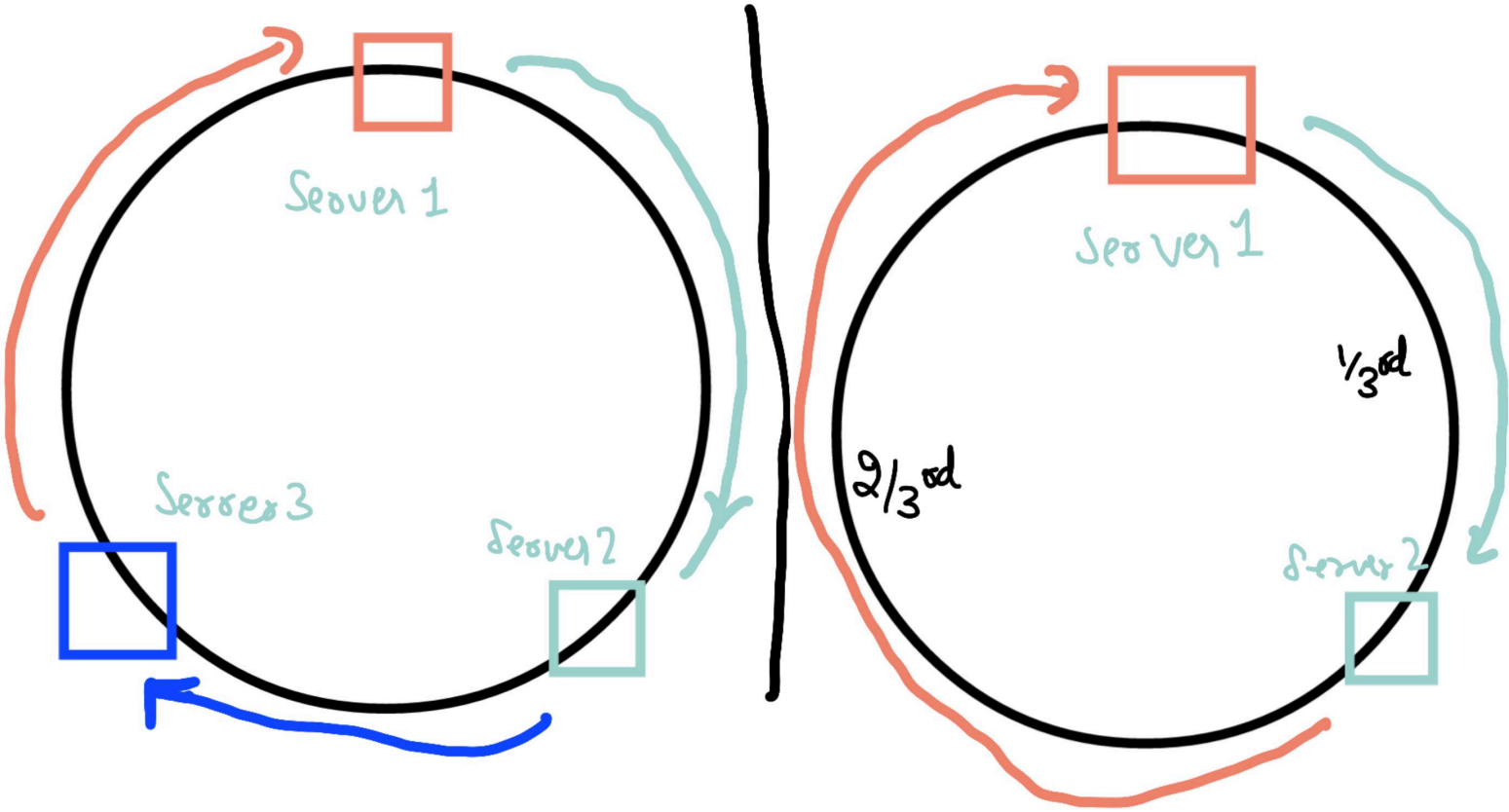
To Solve above cache problem, consistent Hashing is used.

Method of Consistent Hashing.

↳ Mark the servers on the circle at equal angles.

→ Now, hash the incoming IP address.

→ If the angle == angle of server then good, else move clockwise till you encounter a server.



Benifit :- Cache is preserved.

Downside :- Load Balancing do not occur in a good manner.

SQL

To improve consistent hashing, virtual servers are made. as explained by Gaurav Sen.

↳ Standard Query Language.

↳ RDBMS

↳ Stored on Disk.

↳ want to read / write very efficiently

↳ Generally B+ Trees are used for the same.

↳ m-1 keys

↳ Stored in leaf node

↳ Sorted.

↳ Schema

↳ what is the field name

↳ what is the data type.

↳ Tables are related to each other with help of foreign key.

↳ Every row should have a primary key.

↳ Joins can be performed.

→ These are ACID compatible.

ACID

A ⇒ Atomicity

C ⇒ Consistency

I ⇒ Isolation

D ⇒ Durability

for durability

* Try to store things in disks in place of storage
  * Speed of read/write reduces.
  * Our data won't be lost in case of power outrage.

Transaction

↳ A collection of queries.

↳ Start with the keyword 'Begin'. and end with 'Commit'.

Atomicity

↳ A database transaction can't be splitted.

↳ A transaction should either be completed or should do nothing.

↳ suppose, server crashes while transfers ing money, now the money is debited from A, but never able to reach B.

Thread-safe with becomes Imp.

Integrity atomicity

Isolation

↳ Isolation of one transaction from another.

elsewise

- ↳ Dirty Reads
  - ↳ Reading inconsistent value (value which is not finalised yet).
- ↳ Phantom Reads

or will happen race cond^{n}.

↳ Therefore, all the transactions should be serialised to avoid such problems.

↳ Think of B and C sendig A (10 intially), 100 ⇒.
now final balance of A should be 210. But it might be 110 if both happened parallely.

transaction B ⇒ Balance A (10) + 100 = 110

transaction C ⇒ Balance A (10) + 100 = 110

Consistency

↳ Like name. colm should not be null.

↳ should have a primary key.

No SQL

↳ Not only SQL

↳ open ended.

↳ Non-relational databases.

↳ So, can't use SQL to query them.

↳ Solves problem of scaling up a d.B.

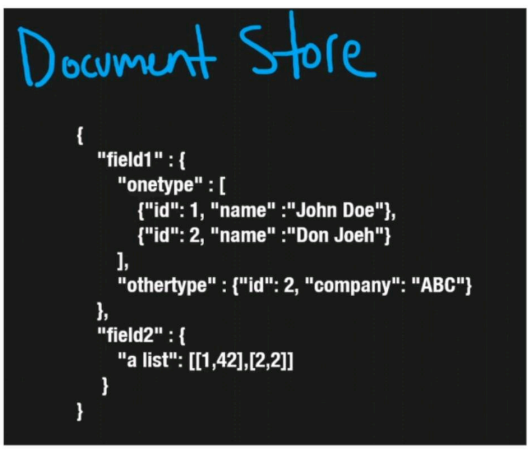
Eg of No SQL D.B.

↳ Key-Value :- Redis, MemCached, etcd.

Works like a hashmaps.

Adv :- very fast because uses RAM.

↳ Document Store.



↳ A collection of Document

⇓

A typical JSON object.

↳ Example :- Mongo DB.

↳ No Schema

→ Wide Column DB

↳ Handle massive massive scale

↳ majorly used, where a high no. of writes are going on.

↳ like time-series D.B.

↳ And low no. of read and updates.

↳ Example

↳ Cassandra

↳ Big Table

→ Graph D.B

↳ Used in general in social media sites.

↳ Like A follow B, who follows C, who in term follows D.

↳ So, to do this in SQL, we have to join 4 tables (A, B, C, D)

↳ But graphQL can do it easily.

↳ Uses directed graphs.

Scalability in NoSQL

Challenge 1.

↳ It is hard to scale a DB horizontally, because in that we won't know, which part of D.B. is our Data located. Is it present in server 1 or is it present in the server 2.?

↳ Or, suppose we are trying to join data from server 1 to data in server 2.

↳ So, it is why people try to prefer to vertically scale a dataBase.

Challenge 2.

We have to maintain ACID properties in case of a SQL database. But thats not the case with NoSQL d.B's. They sacrifice the acid property.

Acronym of NoSQL data base.

Base

Ba

Basically Available
Users can access the database concurrently at all times, and one user doesn't need to wait for others to finish a transaction before updating a record.

S

Soft state
Without a guarantee of consistency, the state of the system can only be predicted with a certain probability after some time.

E

Eventually consistent
If the system is functional, the data will eventually get to a consistent state after

enough time has passed.

Eventual Consistency

Suppose we have server 1 as the main server and server 2, which stores the copy of server 1.

Now, what if some write happens in the server 1? Because no update in Server 2 will make the D.B inconsistent.

So, it becomes the responsibility of the Leader (Main D.B.) to update the follower (server 2)

Benifit

↳ Generally, the no. of reads are more than no. of writes. Therefore the above scenario comes into the play / picture.

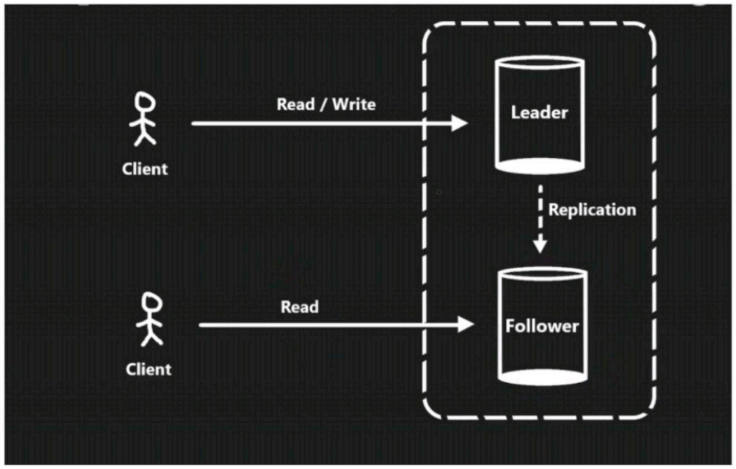
Replication (Leader - Subscriber model)

↳ Creating copy of the D/B.

↳ Used when

handle the no. of requests it is getting. (Read/write)

→ In this case, we are allowed to only read from the Replica (Follower).



→ Reason, because as we have read in eventual consistency, it is the responsibility of the leader to write in the follower.

→ Can be used to scale up the read requests.

→ **Challenge**

Async Data update to the follower.

Like data Leader updated at 12:00 am.

data update scheduled for 12:05

follower

Someone tries to read data from follower at 12:02 am?

Sync Data

- Replicate the write req. immediately.
- But this increases latency.
- Steps
  - i) Client initiates a transaction.
  - ii) Update the leader.
  - iii) Leader update the followers.
  - iv) Get as ACK of the work done successfully.
  - v) Leader tells the client that the write req. succesful.

These extra steps increases the latency.

⇒ Benifit.

- Increased availability and reliability.

Replication

Leader follower

Leader - Leader.

Downside

↳ Becomes out of sync easily.

↳ Do not remain consistent.

Pros

↳ Useful when we can divide the user base into two - or more sub groups and later assign D/B to them. Now if the D/B's get out of sync; that won't be an issue too.

Sharding

↳ Parting into

smaller pieces.

Use:

An approach of horizontal scaling. Used when we have data on the scale of Peta Byte.

Queries are taking multiple seconds to run.

What we do

↳ we not only keeps the data on the multiple machines but also split the data into multiple smaller DB's. and putting these smaller DB's into their each seprate machine.

How to Shard ?

* Range Based

↳ Like if first name has first character in the range of A to L, then put it into Server A, else wise put into server B.

Cons

↳ Running joins will become both slow as well as complicated.

* Hash Based / Consistent Hashing.

* Popular SQL databases such as MySQL or Postgres Do not support sharding by default. because we loose the consistency

* Most NoSQL DB's have sharding by default.

CAP - Theorm.

C ⇒ Consistency

A ⇒ Availability

P ⇒ Partition Tolerance.

} Relates to Databases.

*

* Applies only to replicated Database. Therefore it won't be applicable to SQL d.B.

* CAP states that the 'P' is always applied and it remains our choice to choose from 'C' and 'A'.

* Partition :- It refers to the situation where a part of the D/B becomes disconnected from the rest of the D/B network.

Like due to some reason the connection b/w the leader and follower broke. The users are still able to connect to the leader and follower but, the leader is not able to connect to follower and vice versa.

* Partition Tolerance :- Our system will continue to work even if we have got a partition. Because, we won't want that due to a small partition, our whole system goes down.

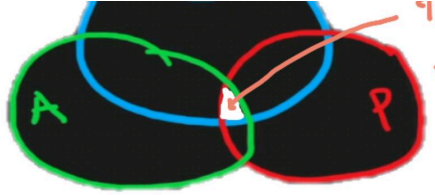
* **Consistency** :- Data - Consistency b/w the D/B's. Every single read will get the most up to date data.

∴ In case of a Partion, the updates b/w the leader and follower stops. Now, either we can take down the replica (Availability ends) and make the DB consistent or let it be in consistent but continue to serve.

* **Availability** ⇒ Availability of a Every node. Na ki 99.9 % availability.

∴ In case of a Partion, the updates b/w the leader and follower stops. Now, either we can take down the replica (Availability ends) and make the DB consistent or let it be in consistent but continue to serve.

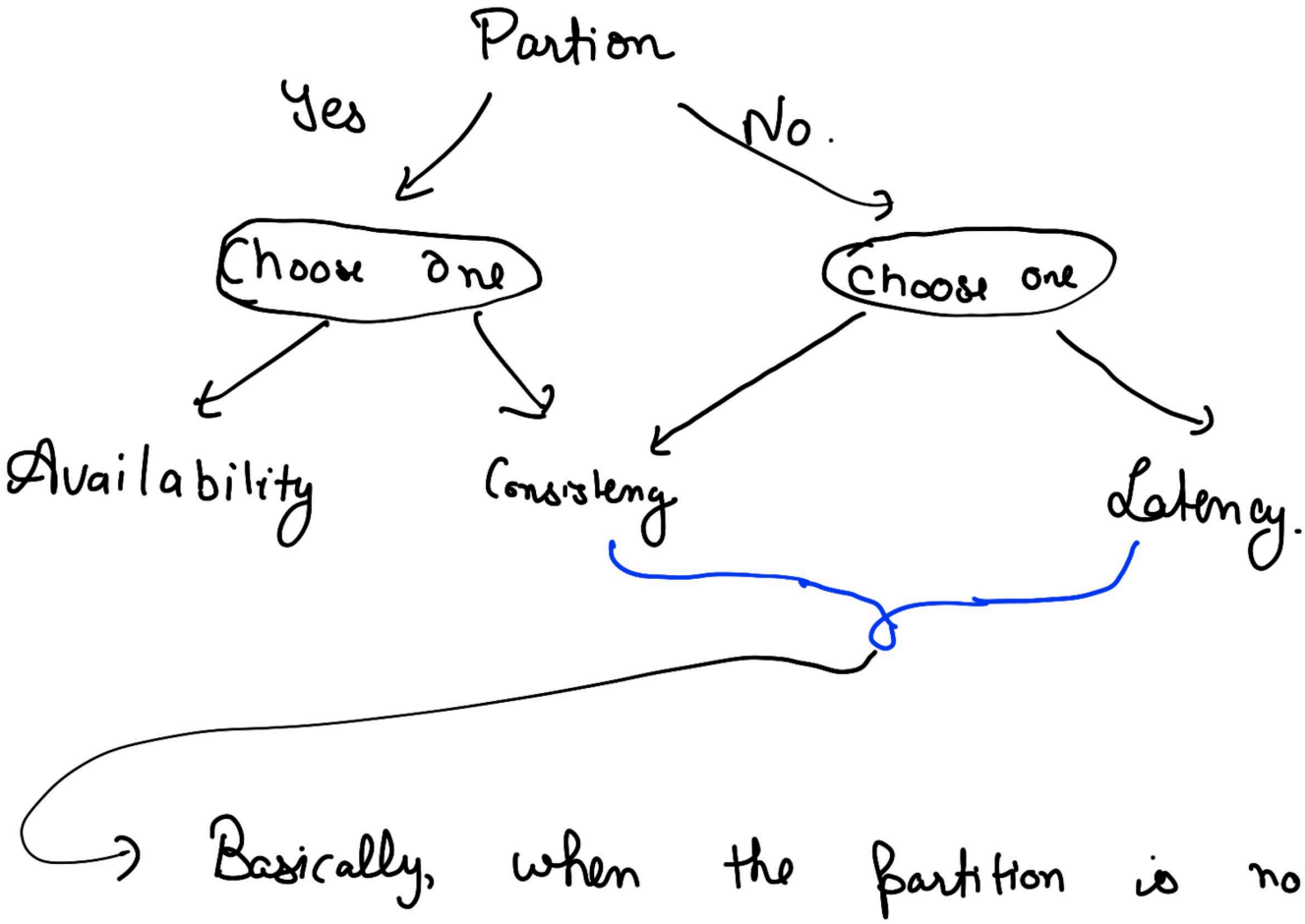
There are a very few D.B's which fits the 'CA' defination.



PACELC

Given P, choose A or C.

Else favour the Latency or Consistency.



→ Basically, when the partition is not there we might choose Consistency (by every write req. updating the replica and increasing the latency) or reduce the latency by choosing not to make the D.B.'s Consistent.

OBJECT STORAGE.

→ No hierarchy. → ∴ We can immediately find the objects.

→ Data is stored in form of a flat way.

→ Example :-

* AWS S3.
* Google Cloud Storage
* Azure.

→ Can't update thing we add.

→ Optimized to store large amount of data.

→ Same as a Hashmap

* Unique key for every object.
* Therefore, can't store duplicate objects

↳ Should have a globally unique names.

* Videos and Images are stored in the Object storage as they are never gony to be queried and will make the D.B. slow if stored.

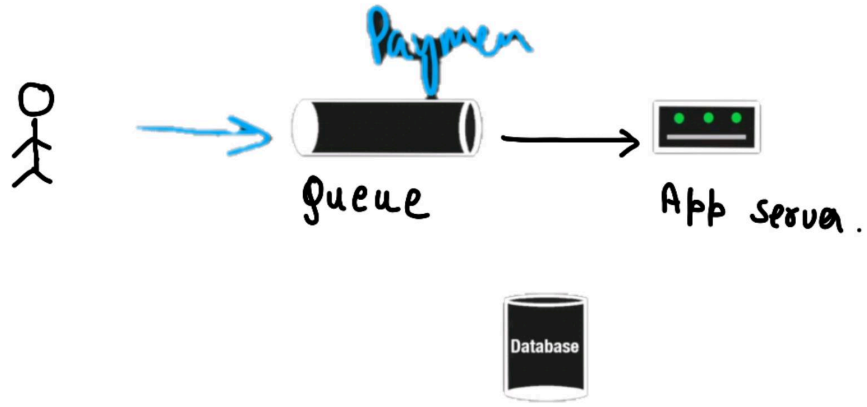
* Reading and writting is done using the HTTP. (but have own security problems.).

MESSAGE - QUEUES.

↳ If we have a large amount of application events and, they are going faster than what our app server can handle. And we do not want to scale up the server.

In this case, message queues comes into the picture.

Message Queues



* The queue is required to be durable. That is, if the system crashes, the message should still be there in the queue.

A Message Queue (MQ) allows applications to communicate and process operations asynchronously. This means that the sending application can send a message to the queue and then continue its own processing without waiting for the receiving application to process the message. The receiving application can retrieve and process the message at a later time, thus decoupling the sender from the receiver and enabling asynchronous communication. This approach is particularly useful for improving system scalability, reliability, and responsiveness.

* The queue will store the incoming requests into it, and when the server gets free it releases the requests. The order remains FIFO.

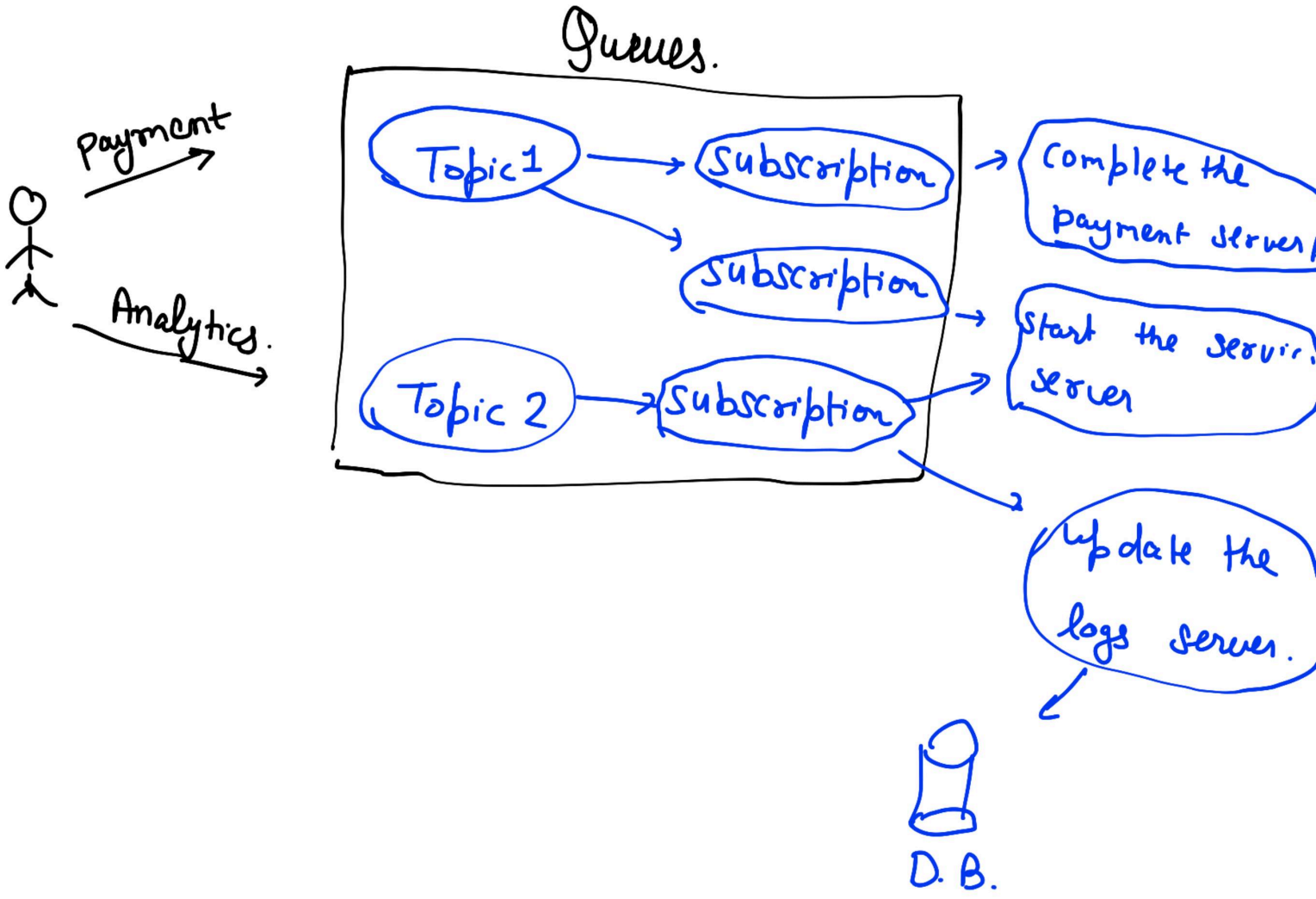
* Queues uses 'Push.' to send the req. to the server. In case, server is not able to handle it, it will send a 'Pull' req. to the

queue to take it back.

In case of successful completion of a request, server sends a 'ACK' for the same. If no ACK, queue will try to resend the same req. to the server untill server acknoledges or the max. trials for one req. runs out.

In case of many applications, sending req.'s to the queue, we get the Pub/Sub mechanism.

- Publisher (users).
- Subscriber (Servers)



Sending same topic to many subscription is called. fanning out.

Same req. can go to multiple applications.
Same Application can recieve multiple type of requests.

Example.

- Rabbit MQ
- Kafka
- Pub/Sub.