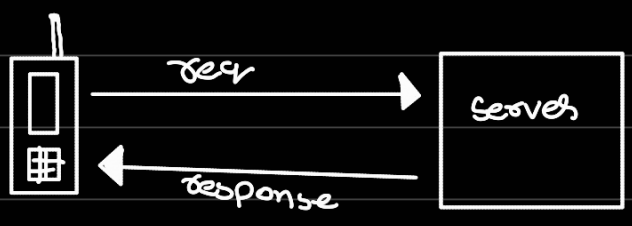
1. Vertical scaling vs horizontal scaling.

- API : Application programmable interface.
  helps us to recieve request & send response.



- as number of request increases our server is no longer capable to fullfill all the requests and thus experiences down-time.

- To prevent this we scale our server.
  Scaling → increasing / decreasing capacity of a server, handle more / lesser requests.

- Types of scaling.
  1. vertical scaling (Buy bigger Machine)
  2. horizontal scaling (Buy more Machines)

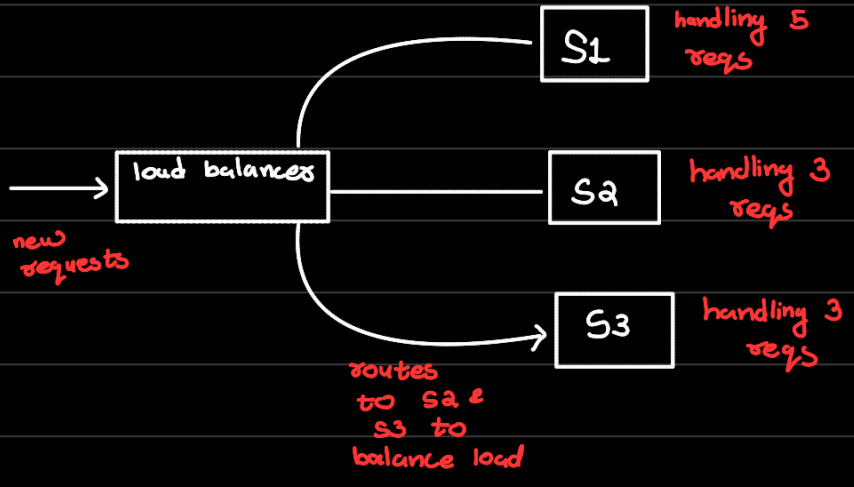
- Vertical scaling v/s horizontal scaling

Vertical	Horizontal.
	 multiple small box
loadbalancing not required	loadbalancing required

Single point of failure.	Resilient
Network calls (Remote Procedural call; which are slower)	Inter process communication (which is faster in nature)
Data inconsistency (need to develop syncing mechanism; i.e at best eventually consistent)	Consistent
scales well as users increases	Hardware limit; you cannot vertically scale unlimitedly.

2. load balancing

- Process of distributing incoming network traffic across multiple servers so that no single server is overwhelmed.



- types of load balancing algos:
  1. hash-based
  2. Consistent hashing

:

these are many other, but these 2 are important ones.

① hash based

- Use some parameter from incoming request; hash using our function and the output determines which server to hit.

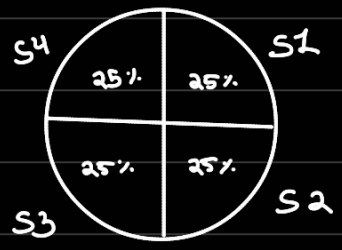
- Say we are using User ID & we have m servers then
  $h(U_i) \% m \Rightarrow$ server to hit

- advantage ? Request Stickiness !!
  i.e another request from same user will be routed to same server. This helps in caching frequently accessed info at servers.

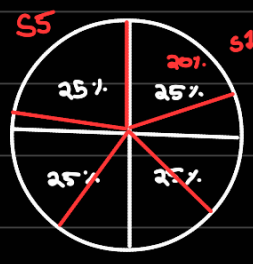
- request stickyness (also known as Session affinity) also implies a form of consistency. All requests regarding a particular user (just eg here) are routed to same server everytime. i.e no possibility of inconsistent update from other server.

- May lead to unbalanced load
- Very difficult to scale-out more !!

Eg 4 servers ⇒ 25 % load per server



now add 1 more servers ⇒ 20 % per servers



$\Delta = +20$

$\Delta = +0 - (-5) = 5$

$\Delta = +5 - (-10) = 15$

$\Delta = +10 - (-15) = 25$

$\Delta = +15 - (-20) = 35$

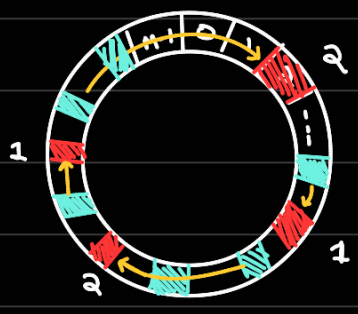
Total = 5 + 15 + 25 + 35 + 20
= 100

} Basically meaning all the Cache stored before become useless.
Need to flush out Old Cache.

- We want as less change as possible such that we minimize Changes in Cache.
- Solution : Consistent hashing.

② Consistent hashing :

- for Request ID ⟶ $h(r_i) \% m$
  imagine a ring :
  Blue squares represents requests.

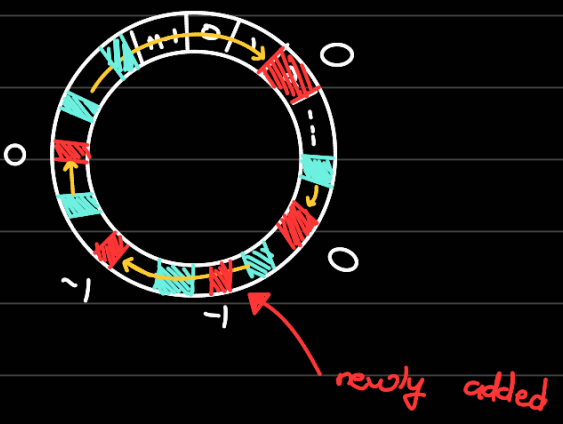


- We have servers iDs as well hash them with same function ⇒ $h(s_i) \% m$
  Red squares represent servers

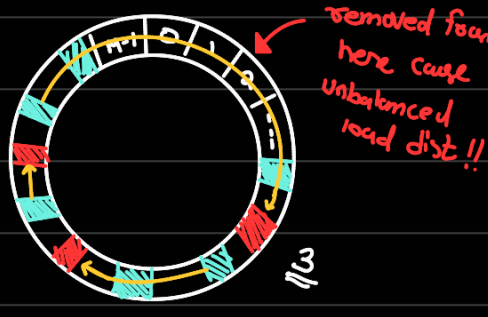
- Requests go clockwise and hit first servers it sees.

- Expected Average load per server = $\frac{1}{N}$

- Add a server :
  ⇒ we observe minimum change here. most of the servers remain unaffected due to adding of extra server



- Remove server
  ⇒ Theoritically load is 1/N. but practically Scenario where even half of the load is directed to same server is possible !!



- Solution ? Virtual servers.
  ⇒ have k different hash functions.
  ⇒ each serverid has points corresponding to output of all of these k.
  ⇒ total k*S points on ring for server
  ⇒ less chances of skew distribution of load !!

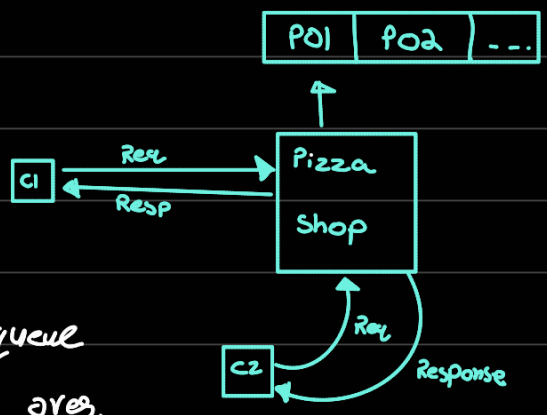
- Problem : Chances of results from these function clashing increases
  ⇒ multiple servers on same point.
  ⇒ solution : have only 1 hash function but create k diff replica ids for each server
  ⇒ reduces clash

* Message Queue

Example : a pizza shop

→ C1 orders pizza, gets a (promise) as response, order noted and production started

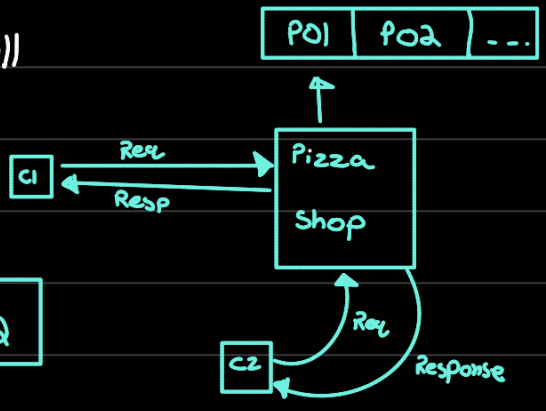
→ C2 orders, same process, order joins queue and will be made later when P01 is over.



- Say our chain scales and we open multiple branches :

→ Each Chain has some orders

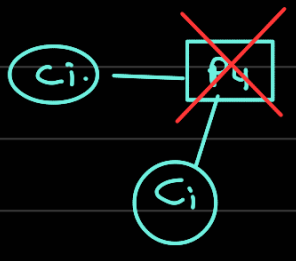
→ Say Chain P4 goes down. we still want to process all the Clients.



→ Needs :

① Orders persists

② Some other chain processes it.

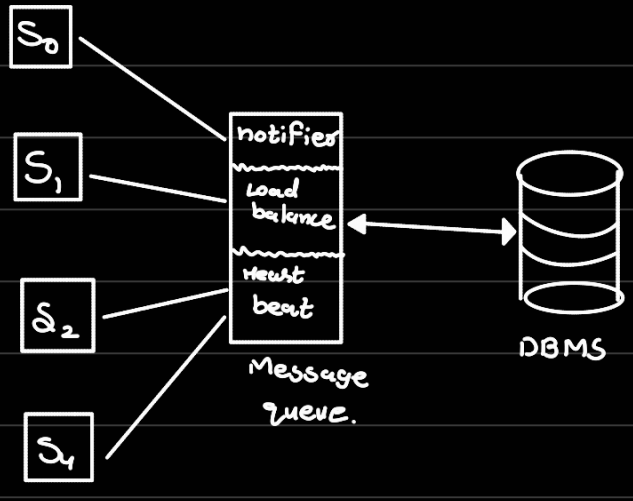


→ These needs are fullfilled by message queue.

* new and updated architecture :

→ Objective of message queue :

- heartbeat : check by polling if servers alive or not
- Load balance : while distributing alive & not executed requests check if other servers already working on it
- notifies : if server goes down notify others to share its load.



▶ all done by Message queue.

- advantages of Message queue :
  1. Decoupling : services work independantly without each other's interruption.
  2. Asynchronous processing : time consuming task done in background.
  3. Reliable & loadbalancing

* Reallife example : System design of "how hashisath hacked NEET database"
  Entire orchestration design developed there is almost 1:1 replica of this. Can discuss thin in interview.

* Monoliths V/S Microoservices

1. Monoliths :

⇒ application is build as a single unified Codebase.
⇒ all modules are tightly coupled together.

- Advantage
  - → Simplicity in implementation
  - → Easier E - E testcases.
  - → less DevOps

- Disadvantages.
  - → Very difficult for a new person to understand full codebase.
  - → not scalable : load on one part ⇒ scale full system
  - → Slower deployment : full build and deploy everytime

- Common misconception :

✗ full Code on a single machine

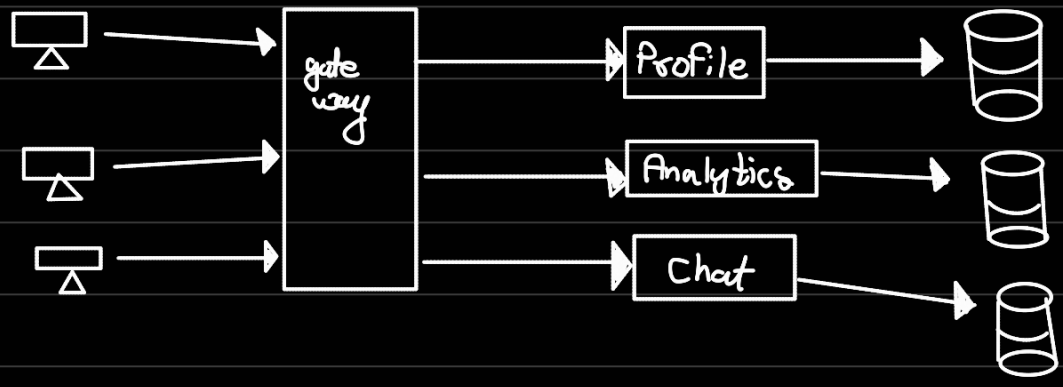
- Not necessarily ; multiple machines (called instance) can be behind load balancer and deploy.

- i.e monolithic architecture refers to how the application is structured (i.e all components built, deployed & scaled as a single unit).

- Basically :
  - + good for small team
  - + less complex / easier deployment
  - + less duplication
  - + faster due to inter-process communication & no RPC required.
  - − More codebase context required.
  - − longer deployment
  - − single point of failure.

② Micro Services

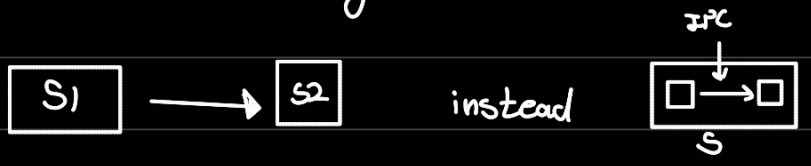
⇒ large application broken down into small, independent services each responsible for a single business capability that communicate over a network.



- Basically :
  - + Easier to scale
  - + Easier for new member to understand micro-service & work on it

+ Decoupled system ⇒ easier parallel development.
+ easier to figure which part is under load and just scale that.
− Needs smart Architect to develop a robust structure.

Common indicator of not needing micro-services but still using is:



- for largeScale systems, micro-service architecture is more preferable than monoliths !!

* Database Partitioning & sharding

- Partitioning : Dividing DB into smaller parts for easier access.

Types of Partitioning → Horizontal (Row based)
→ Vertical (Col based)

- Sharding is a technique to achieve Horizontal partitioning.
- Basically:
  Partitioning → dividing db to smaller parts.
  sharding → distributing these parts (shards) across servers

i.e usually Partitioning → Single db server
sharding → multiple db servers.

- With sharding comes a tradeoff b/w Consistency and availability.
  CAP theorem.

CP ⇒ Consistent DB

AP ⇒ eventually - Consistent DB.

Problems with sharding :

① JOINS b/w shards are extremely expensive

② Number of shards are fixed

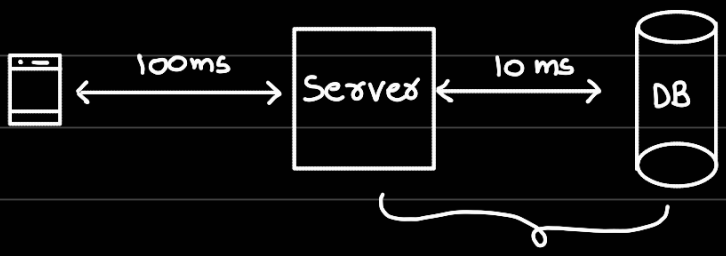
⇒ solution : install a managers at each shard.

⇒ perform more sharding on a shard and manager routes to required shard.

- A good practice is to index a shard. Makes read faster.
- For further scaling implement Master - slave architecture. Master : do writes
  Slaves : handle read operations.

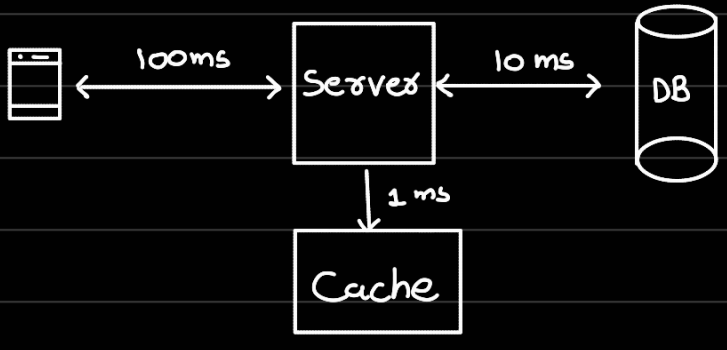
- Types of sharding :
  → Range based : based on ID's shard db
  Reduces search space.
  → hash-based : Apply hash then shard
  better distribution but hard to add/remove
  → Geo-based : based on geography
  Reduces latency of requests.
  → etc based on requirements.

Caching



Cache works here as a backend Optimization.

- basic idea : reducing repeatable work through Storage.
- Cache is much closer to server and hence much faster
- Cache are also smaller in size hence efficient :
  - Populating policies
  - eviction policies



- Drawbacks of an inefficient Cache :

- The cache never has usefull data. So in every backend call we query cache (extra work in this case)

⇒ without Cache : 100 + 10 + 10 + 100 = 220

with Cache : 100 + 10 + 1 + 1 + 10 + 100 = 222 (increase)

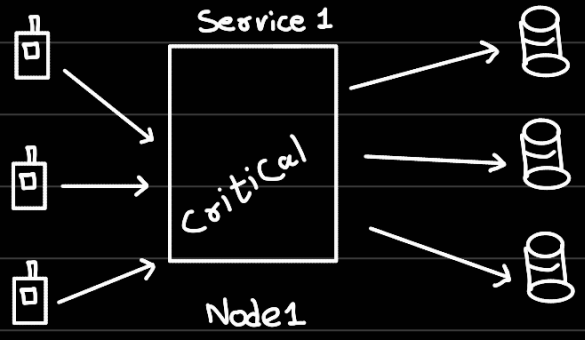
- This type of scenario is called thrashing.

- Eventually Consistent Caches
  - DB is the final true source of information.
  - Cache may be updated later (when is decided based on policy).
  - example: like count on youtube. it is ok to update the like count later and thus "evental Consistent" architecture might be suitable here.
  - Note: in some applications, like banking stale updates might cause panic thus "consistent" systems are required here.

- Cache placement
  - where to place a cache ? (Consistency across multiple servers)
    - in memory at Server
    - Distributed Cache near servers
    - Cache on DB
  - for large scale systems all are important but the most to focus on is Distributed cache (Redis)

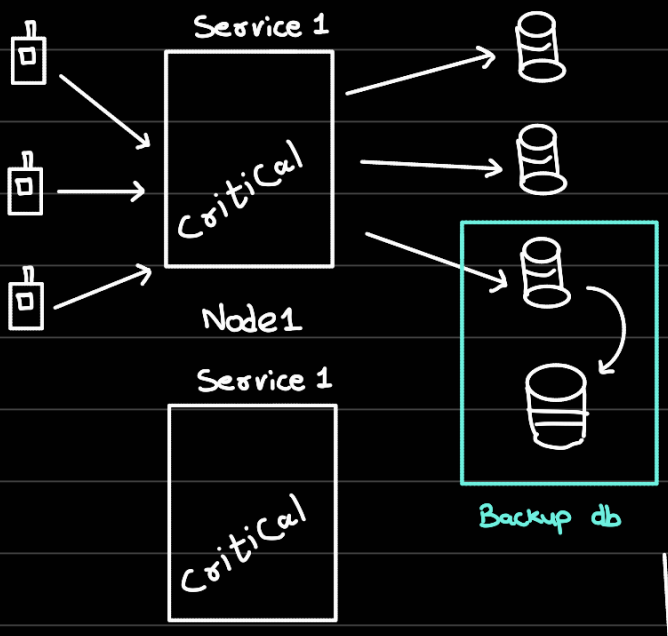
Single point of failure

- tests resilience of a system



if this node is down, full system fails.
i.e not a resilient system.

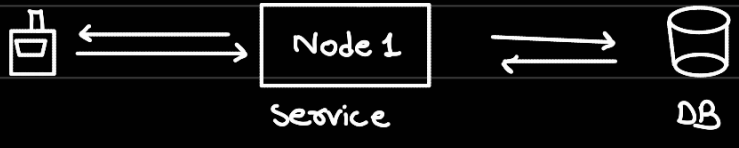
- Obvious solution : add another node. (called backup)



- More resilient DB
- All the changes in main DB mirrored to another
- This does make sense
- This process is called "making replicas".
- Master - Slave architecture can be used as well.
- Master handles writes which become "eventually consistent" Slave handles read operations.

- Backup services don't really makesense. if the service is not really used then what is the point of that service

Example :



- A non-Resilient architecture. how to make it resilient ?

① Add more nodes
- But then which node to hit ? add loadbalancer.
- Now Loadbalancer becomes SPF ! Add multiple LB.
- Which LB to hit then ? Add DNS that takes care of it

↓

Domain Name System
Converts human-readable domain names to IP address.

② Make DB replicas.
- Basically Distributed DB.
- Use a coordinator in midde.

③ Regional Dependancy.

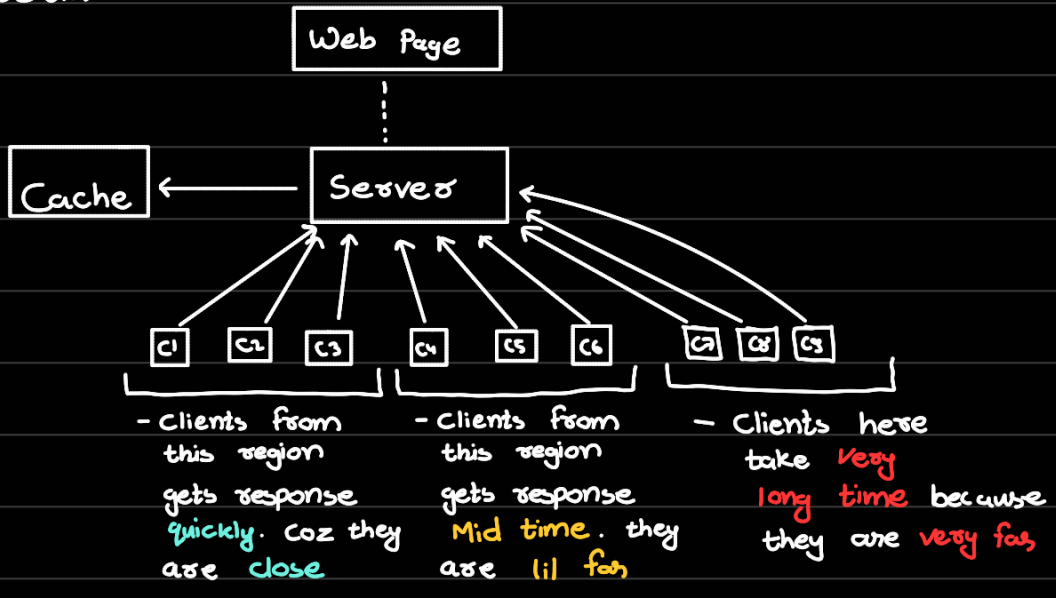
- if entire architecture in same location then prone to total failure by disasters. Hence distribute system to multiple region.
- also helps in regional latency reduction.

* Netflix has Chaos monkey that randomly spawns and makes random Nodes go down to test the resiliance of a system.

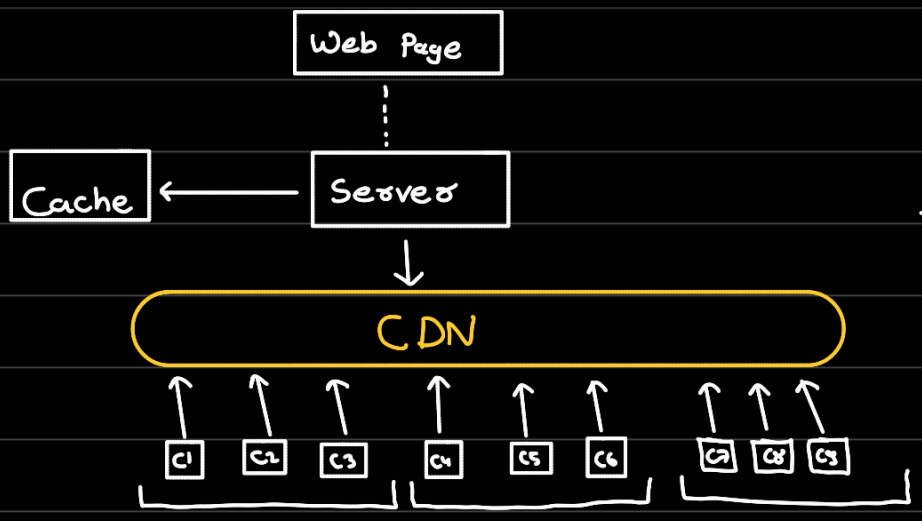
* TLDR : Throw money, Orchestrate and done !

CDN (Content Delivery Networks) :

- Makes system Cheaper & Faster.
- Problem :



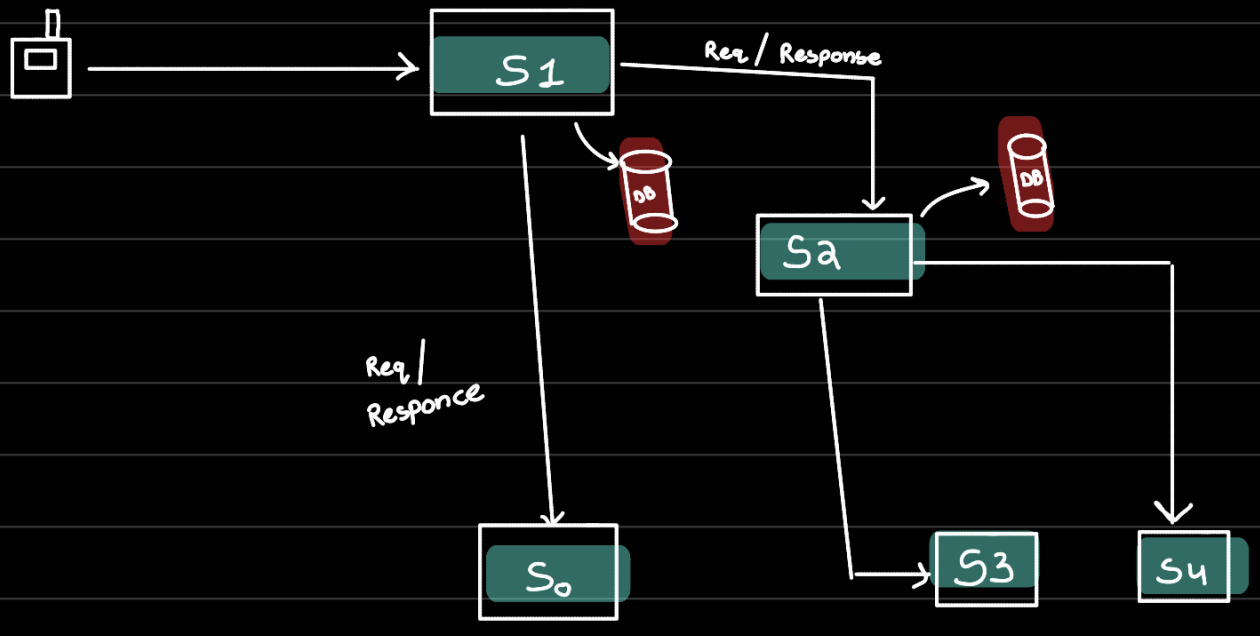
- Possible solution : move servers midway but then all will have mid latency.
- Problem 2 : Content Censorship based on region. Some of the shows and content may not be allowed in a region and has to be blocked.
- Relevance of Cache based on region.
- All these problem statement are solved by CDN.



* Basically a distributed server across all geography to solve the above problem statement

Event - Driven Services ; Need for a Pub-Sub , Adv & Dis-Adv of both!

① What is Event- Driven service ?



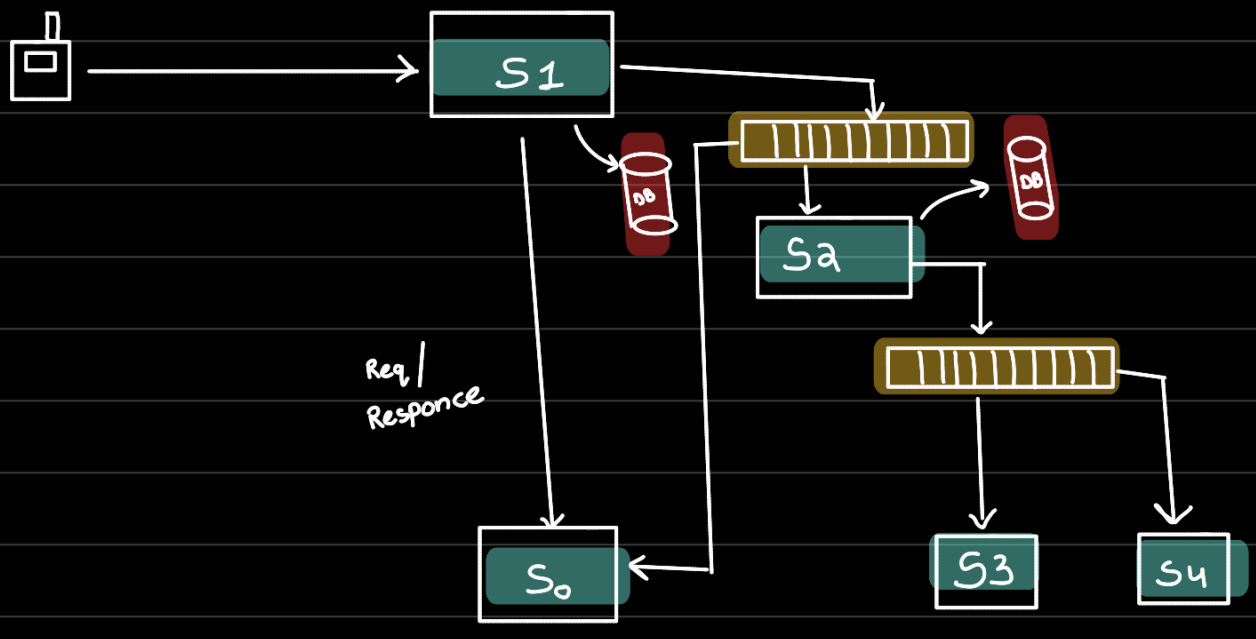
- S1 gets req does some processing then fires reqs to S0 & S2
- Order of these reqs don't matter.
- Similar scenario happens again at S2.
- Ideally the requests will be sent Asynchronously. then wait for response.

Drawbacks of Req / Response :

- Strong Coupling : S2 waits for S3 & S4 response then sends response to S1. S1 was waiting for S0 & S2. then response reaches to Client.
- failure latency : long time for client to know failure happened.
- if we have, get req → update DB → send req → wait for resp then there are chances of stale DB. i.e inconsistent DB.

This is where Pub/sub helps :

- Messages here are Fire & forget.
- send Message to Message brokers, which persists data.
- i.e once Message sent to brokes we can rest assure that it will reach. so no need to wait for response.



Advantage of Pub/sub :

- Decoupling : Removing dependancy of wait for response.
- Simplifies interfaces : Servers sends generic longer messages to Message brokes who intern send generic messages forward. no need to take formats into account.
- Transaction Gurantee : Once message sent to broker if will reach destination in some or other time. Coz brokers have persistant memories in them.

- Easier scaling process : messages are generic take what you want. new servers poll brokers so no/minimal change required for scaling.

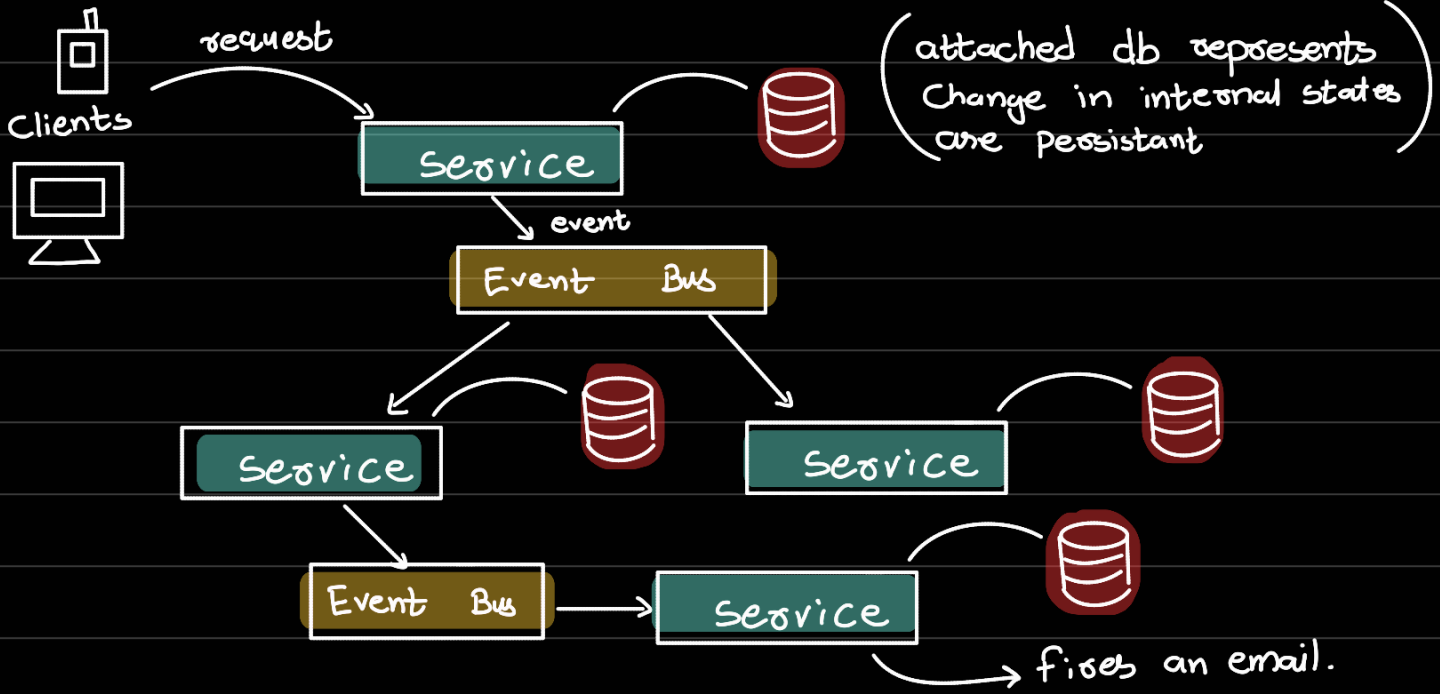
Disadvantages of Pub-sub :

- Poor Consistency :
  Order of event delivery or time of event delivery not guaranteed here.
- Thus, there is chance of state mismatch between systems.
- i.e not usefull in financial systems where we expect complete consistency all the time.

* Pub/sub majorly used in tweeters posts service !

Event Doiven Systems

- Systems do not directly communicate with each other
- A system publishes Events.
- All other systems consumes the Event and figures if it is relevant to them.
- These states may change these internal states and thus leads to more events for other services.



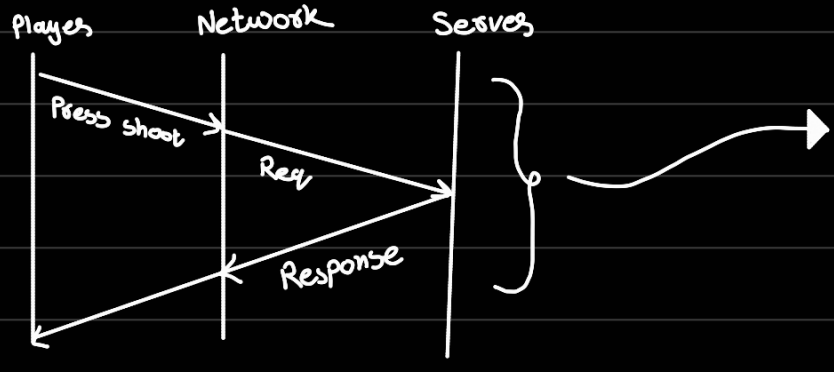
Note : Event Driven Architecture is an architectural pattern to focus entire system's implemention on Events. Pub / Sub is a way to implement EDA.

i.e All Pub/sub systems are part of EDA, but not all EDA must use Pub/Sub.

- EDA's most commonly used in
  - Git
  - React
  - FPS games etc etc.

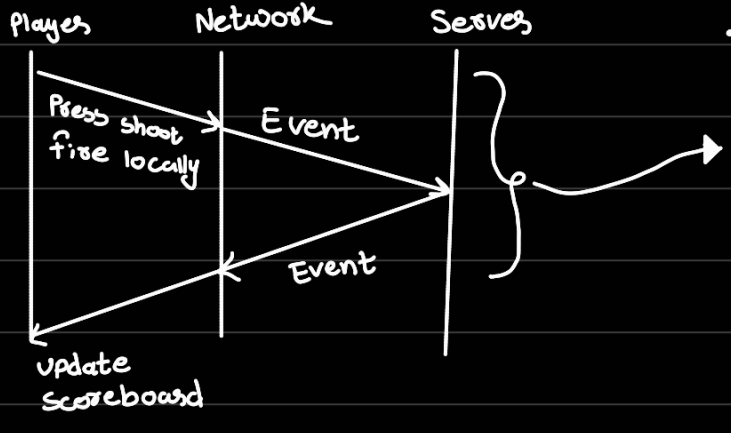
- Scenario : Player presses "shoot"

(a) Request / Response Architecture



Client experiences delay coz of strong Coupling with servers.

(b) Event Driven Architecture.



- gun fires instantly on client
- server later confirms
- i.e player does not feel the latency.

- This is why Desync happens in game, Client sends Event, server later processes and if incorrect shifts client on the correct decision thus client experience out of sync for some time.

* flow :

Service recievers request ⟶ sends event to Event Bus ⟶ other relevant services fetch the Event and stores it in there memory ⟶ Clear Event Bus.

- Unlike Requests, Events are full of generic information so storing them increases availability. Coz if the parent service is down, we still have locally info.
  Drawback : Consistency is an issue here, hence EDA not preferred in finance systems.

- Another advantage of EDA is Rollback, if we have event log we can move to any past previously easily. helpfull to debug

- Another disadvantage is lesser Control. What if some of the data in an event is not ment to be public. Possible to tackle this but difficult and add an extra layer of complexity.

- Ways to use event log (how to reach to a specific time - stamp) :
  ① Replay from start
  ② Diff based (delta change like i built in light weight VCS)
  ③ Undo (but not all events can be undone.)
  A simplification is to squash events, i.e end of the day Compress

- flow of program also very difficult to implement.

API Design

- Application Programmable Interface
- Way to expose your code's functionality to people !

eg: Google deepmind repo, model are close sourced but the API is opensource. API allows user to use model without leaking the model parametes.

- Factors to take care while designing API's :

① Naming API : if the call is supposed to get Admins name it getAdmin :)

② Parametes : Don't request unnecessary parametes.

③ Return Response : Only Return what is asked.

Scaling Writes in db

- DB uses B+ trees within them.

Insertion : O(log n)

Searching : O(log n)

* series of queries are fired on DB ; for each query we need to send response as well.

- We can do Request Condensing ; i.e group incomming requests and send later

Drawback : Additional memory required to store a group, temporarily.

- Use linkedlist ; O(1) writes.

Logs are similar to linkedlist and used here a lot !

Drawback : Reads are very slow.

— Sorted Array → O(logn) read
but b+ tree already O(logn) then why the shift ?
⇒ we can use Linked list + Sorted assay

	B+	Ours
Read	O(logn)	O(logn)
Write	O(logn)	O(1)

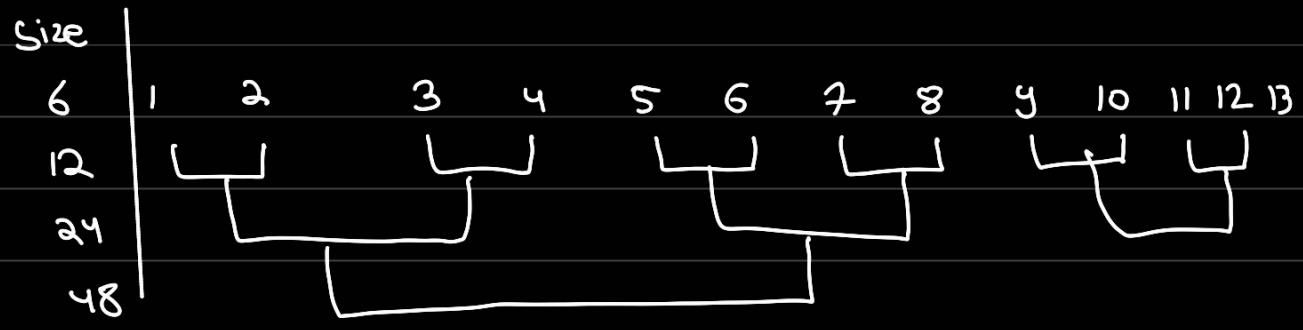
— Now the question is how to Sort Data !

- get all data, then sort & find (X) — really unefficient
- so we have to somehow store data sorted.

- If we have Sorted data → New data comes → sort again
  i.e 10K data, new 6 data points enter, we have to sort O(nlogn)

- maintain chunks of sorted data say size 6
  if we have say 2 chunks then
  log(6) + log(6) = 6 is worst case search time ; i.e nlogm

- But for large db numbers of chunks is also very large
  idea: merge smaller clusters
  i.e say 13 clusters of size 6 then



So we have 48, 24, 6 as our 3 clusters.