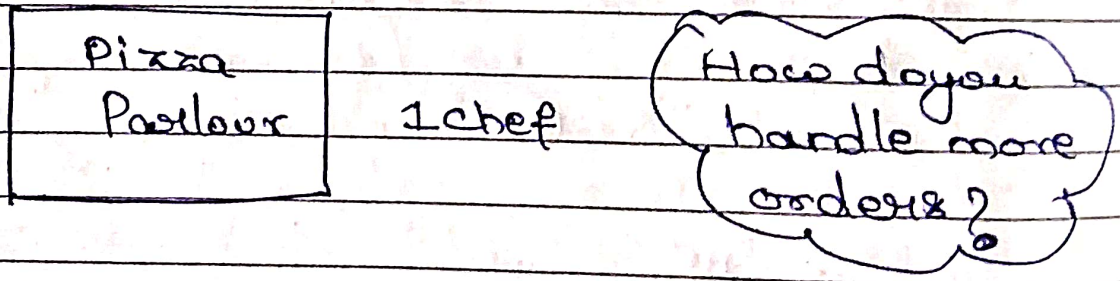
* System Design:- Usually when we are building an engineering system, there is some sort of background behind it. Example (Real world) — Opening a restaurant. Let's see how it happens.

You Have a Shop. Can you scale.



① Optimise processes and increase through-put with the same resource. This is called Vertical Scaling.

② Prepairing beforehand at non-peak hours. (Pre-processing and Cron Job).

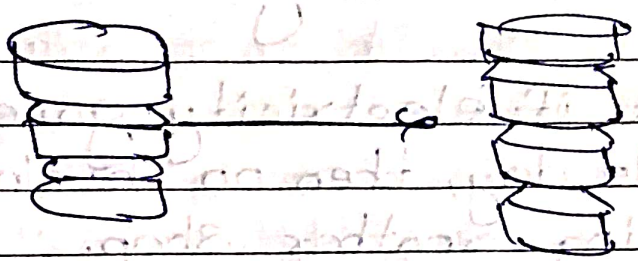
RESILIENCE:-

PAGE NO.:
DATE: / /

(Recovering from failures quickly).

→ Having backup chefs.

③ Keep backups and avoid single point of failure.



Master Slave

→ Masterslave architecture.

④ Hire More resources } Horizontal Scaling.

10 chefs + 3 backup chefs.

The Shop Cooks time for... {Ex-Pan-Sion}.

Chef①

Chef②

Chef③

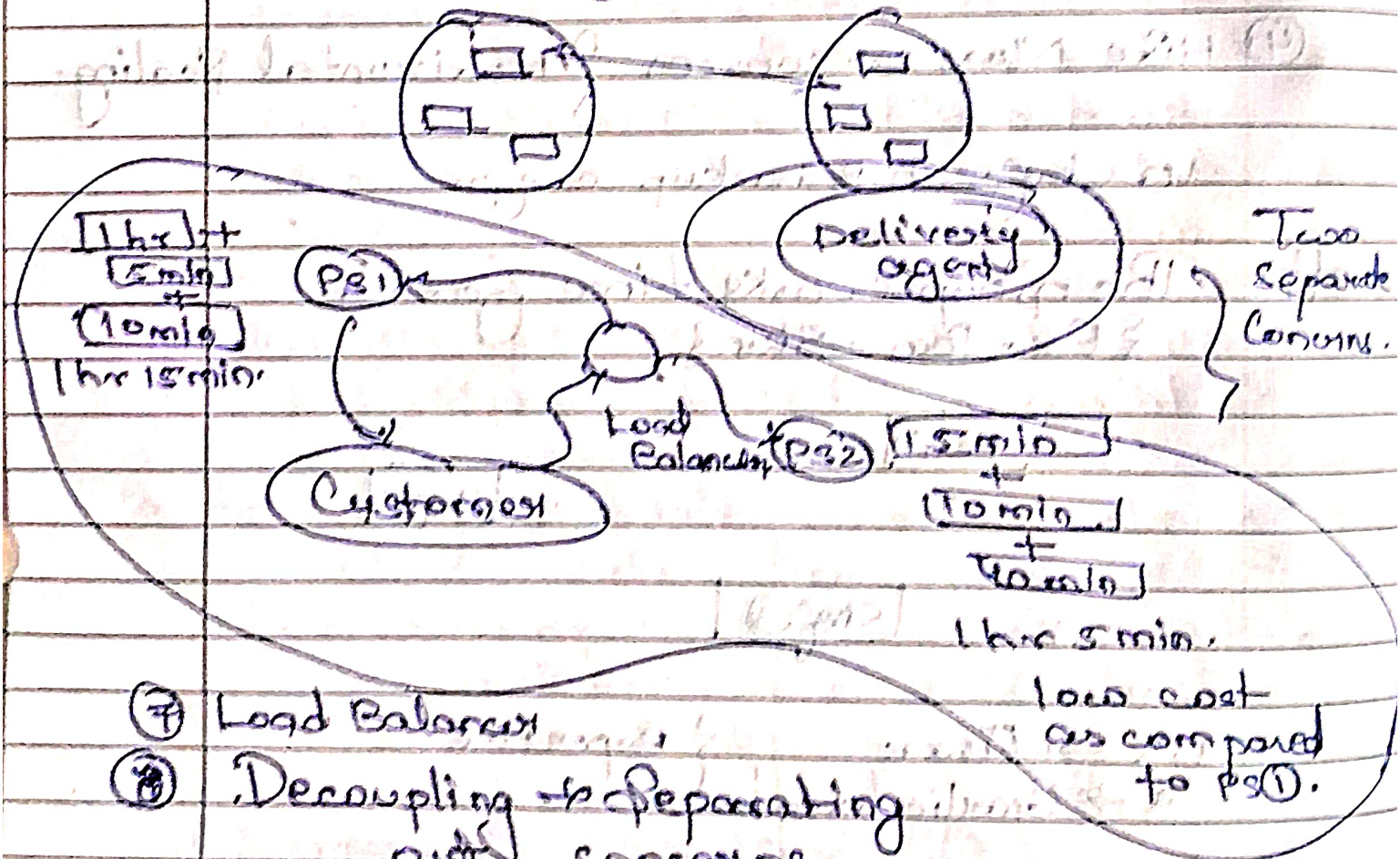
1, 3 → Pizzas } Expertise.
2 → Garlic bread }

On receiving an order, which chef should see. assign it to?

⑤ Micro Service Architecture - (Scaling a team at a different rate as compared to the other two teams of pizza and dividing responsibilities. Basically nothing outside the business usecase that you handle.

What if electricity gone for the shop whole day, then no business that day...
Opening another shop.

⑥ Distributed System (Partitions)



⑦ Load Balancer

⑧ Decoupling → Separating out concerns.
(delivery agent only cares about delivering food).

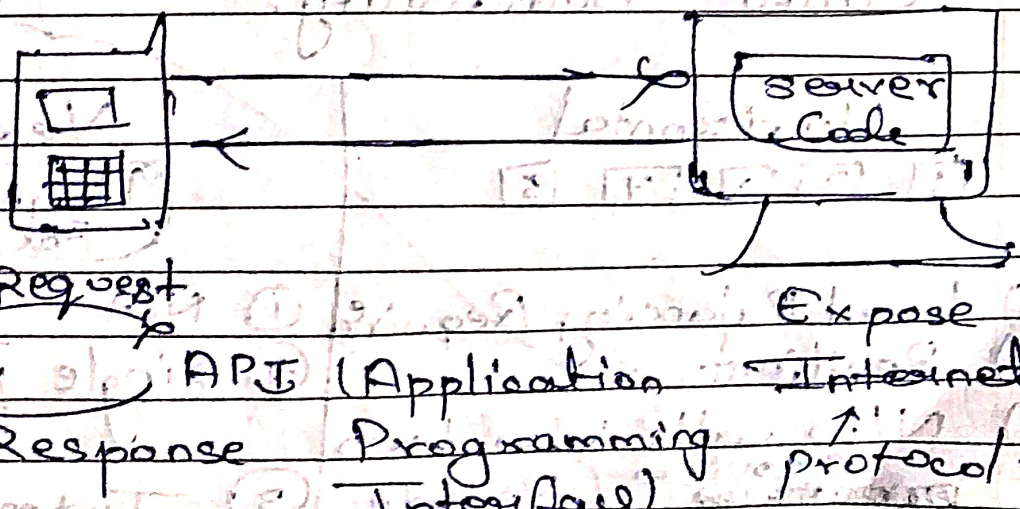
PAGE NO.:
DATE: / /

⑨ Logging and Metrics Calculation. (where delay happening)
⑩ Extensible/Not rewriting code again and again (reusability).

Ques:- Can You Scale ? — Yes.
At high level, what problems we face and how to solve them. (High Level Design)
i) Order Overload → Recruitment
ii) Complexity → Separation of concerns
iii) Mishaps → Fault tolerance

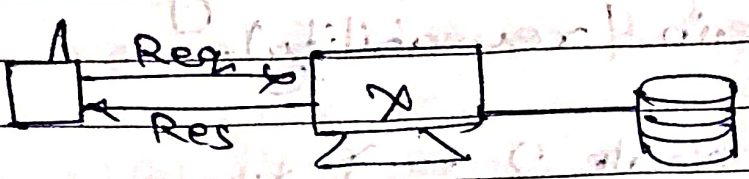
Now → HLD vs LLD → how to actually code stuff, making classes, objects, functions, etc.
→ Deploying on servers, figuring out how two systems will be interacting with each other.

→ System Design Basics : Horizontal vs Vertical Scaling :-



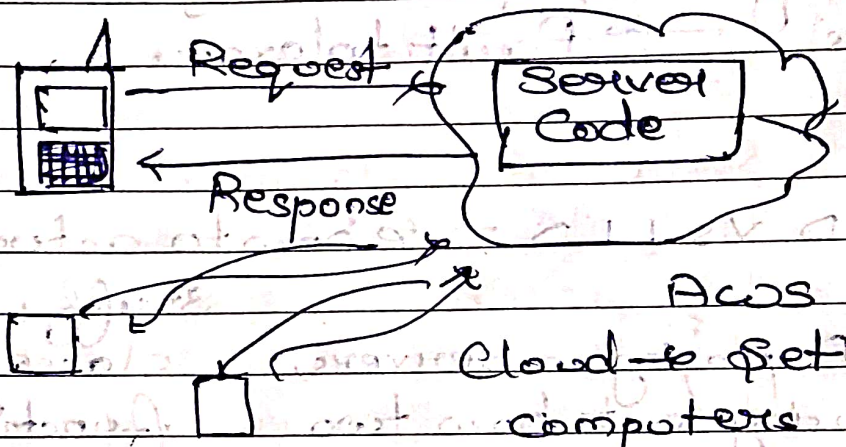
PAGE NO.:
DATE: / /

Now db also connected.



What if server goes off..

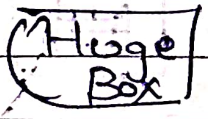
→ We should host services on the cloud.



AWS
Cloud → set of computers that you can use to run your service.

i) Buy Bigger Machines (Vertical Scaling)
ii) Buy more Machines (Horizontal Scaling).

→ The ability to handle more requests is called Scalability.

Horizontal	Vertical
	
i) Load Balancing Require	① N/A
ii) Resilient	② Single point of failure
iii) N/w calls blw two services (Remote procedure calls) slow	③ Inter process communication (Fast)

One crashes other comes up.

(iv) Data Inconsistency (Transaction ③→④→⑤...)

Suppose (ii) where operation has to be atomic, we have to lock all the servers which is impractical. So; loose transactional gaurantee.

⑧ Scales well as users increase (linear)...

(1) Consistent...

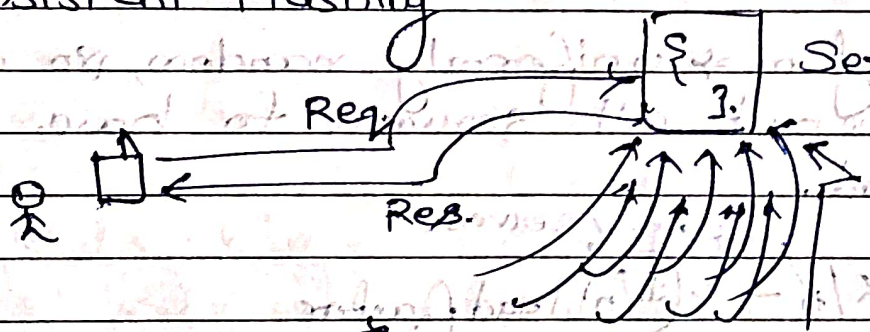
PAGE NO.:
DATE: / /

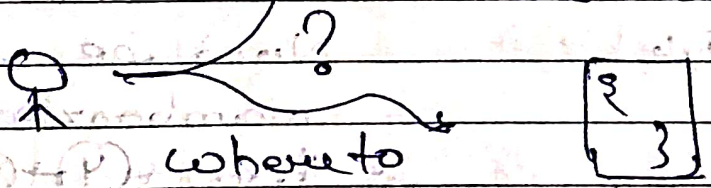
⑤ Hard coaxie limit. (Can't make bigger-bigger...and solve issue)

Hybrid Solution → (ii), (v), ③, ④ and ⑤.

→ WHAT IS LOAD BALANCING?

Consistent Hashing





Where to go?

How to balance load on both the servers?

→ Taking N-servers and trying to balance load evenly on them is called as load balancing

Request ID } every request (randomly generated no.)
↳ 0 to M-1

h(r1) ⟶ m1 // hashed no.
↳ can be mapped to a particular server.

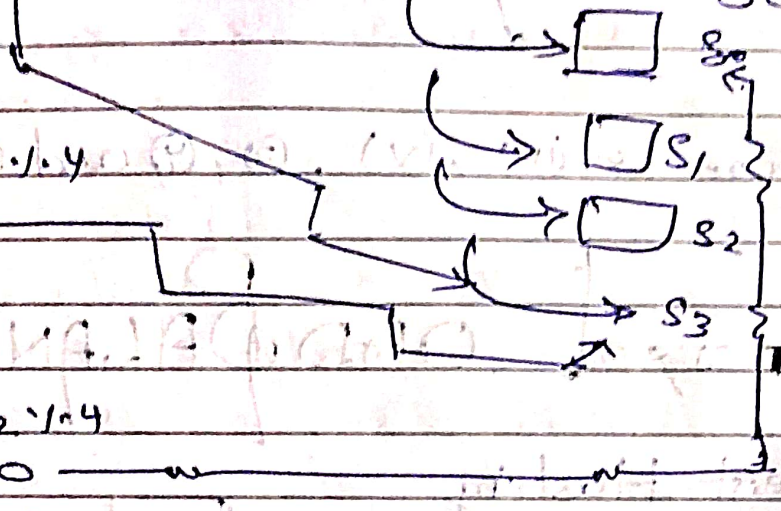
h(10) ⟶ 8%4
= 3

h(20) ⟶ 15%4
= 3

h(35) ⟶ 12%4
= 0

m1 % n

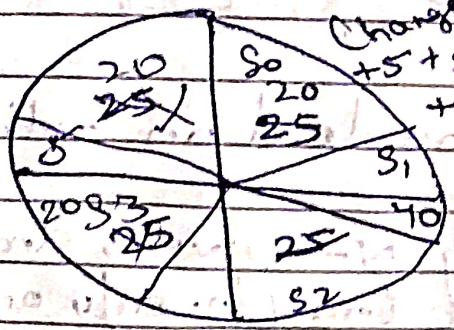
Send rem. index to respective server.



hash func ⟶ uniformly random, so can expect all servers to have uniform load.

X/n ⟶ (1/n) load factor
↳ no. of servers
↳ no. of requests

Now we add S4 then mamboozled, change hash ⟶ (4) ⟶ (5)
n ⟶ n



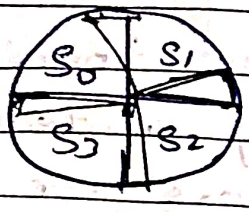
Change
+5 +5
+10 +10
+15 +15 +20 +20

Cost of change ⟶ 100 = M

In practice not random r_1, r_2 but a user id (cached) → storing relevant info for those servers.

PAGE NO.:
DATE: / /

Avoid → Range of numbers that you are serving.



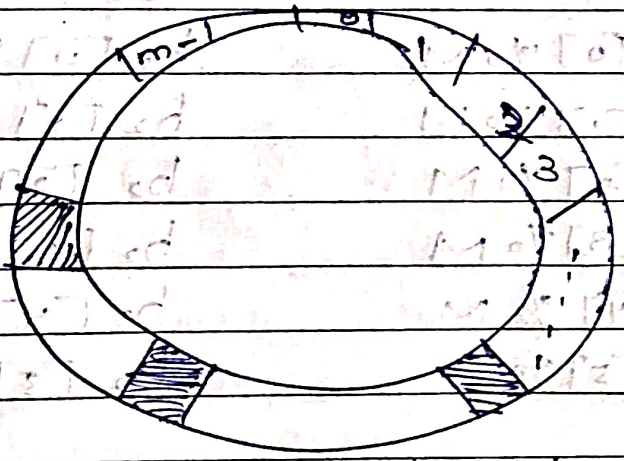
Taking a bit from every server such that sum of these areas is 20% but overall change should be minimum, that's the general idea.

So, we need advanced approaches and here comes the consistent hashing.

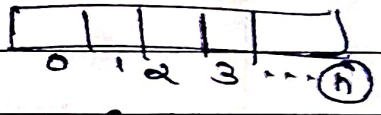
→ Consistent Hashing :—

(Problems is not hashing but adding or removing servers). (It changes completely the local data that we have in each servers).

Still Hashing req^n id's bit ...



ring of hash



array mapped.

h[0] % M
h[1] % M
h[2] % M
h[3] % M
:
:

Taking remainder with search space M } hashing them as well } Server ids

h(0) % M = 19

PAGE NO.:
DATE: / /

When a request comes to this ring, what we do is to go clockwise and find the nearest server.

Expected load factor — [1/N]

Suppose I loose a server, then any request suppose first, added, goes to S4. Now load factors per server changes, now suppose S1 goes down. Now S4 will be in burden. Practically there happen skew distribution. Half of the load on a single server is terrible. Theoretically, its going to be the minimum change, but how practically it will be solved ??

Solution — Start Making Virtual Servers (Not Virtualboxes), but using multiple hash functions.

$h_1[0]$ % M	$h_2[0]$ % M
$h_1[1]$ % M	$h_2[1]$ % M
$h_1[2]$ % M	$h_2[2]$ % M
$h_1[3]$ % M	$h_2[3]$ % M
$h_1[4]$ % M	$h_2[4]$ % M
$h_1[5]$ % M	$h_2[5]$ % M

K — hash functions, then each server will have K-points.

Likelyhood of one-server getting much much load is very lesser.

So, Expected minimum change in no. of servers.

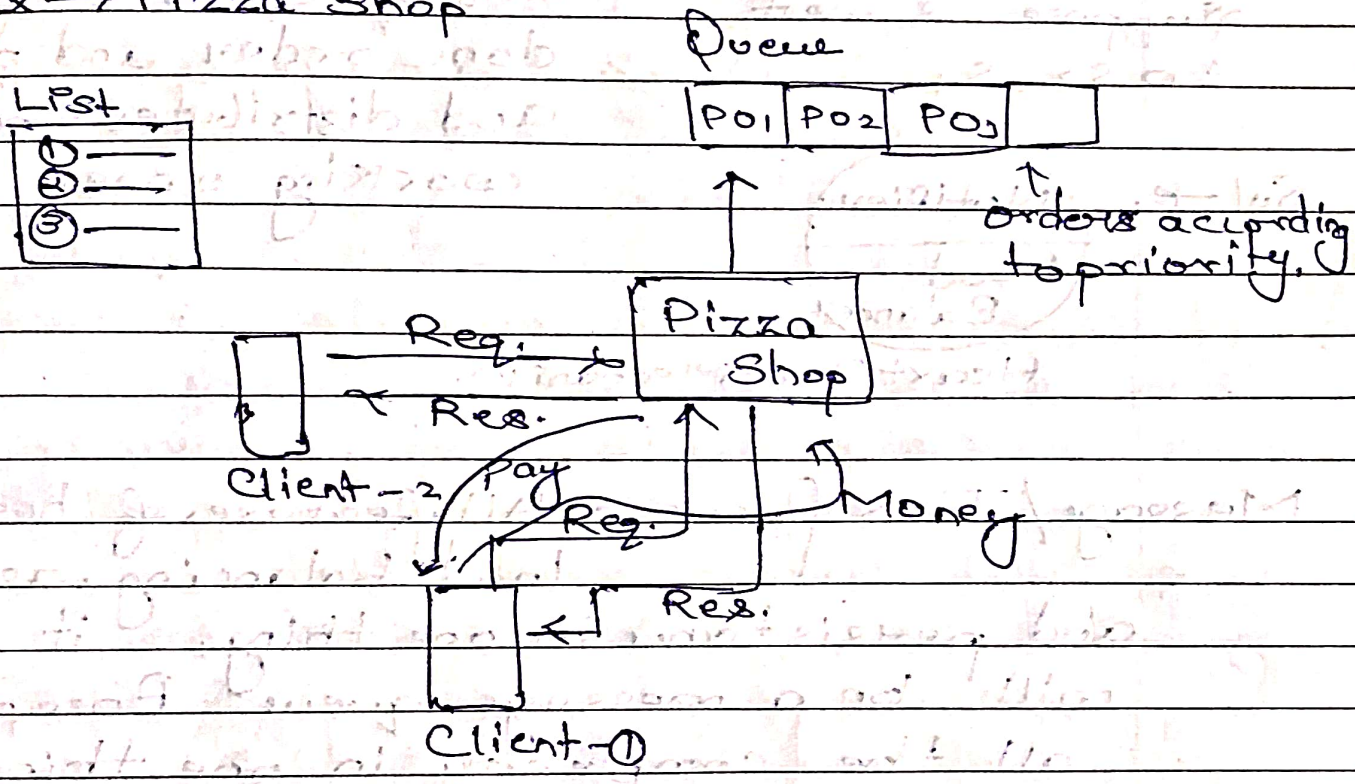
Load Balancing used in many places ~
a) Distributed systems
b) Web Caches
c) ~~Databases~~

PAGE NO.:
DATE: / /

Consistent hashing → Flexibility
→ Load balancing in very very clear and efficient way.

MESSAGE QUEUE (What is it and where it is used??).

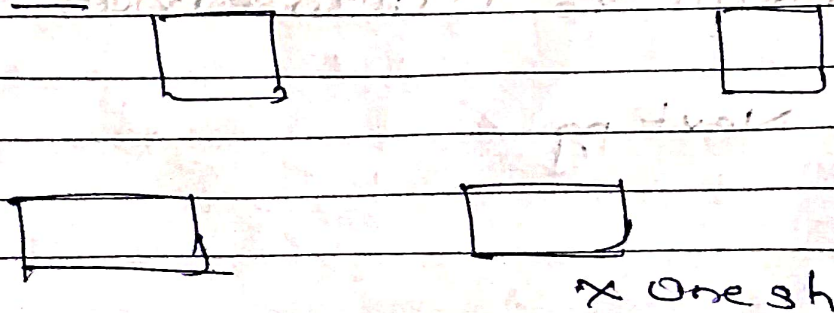
Ex → Pizza Shop



The whole thing works asynchronously.

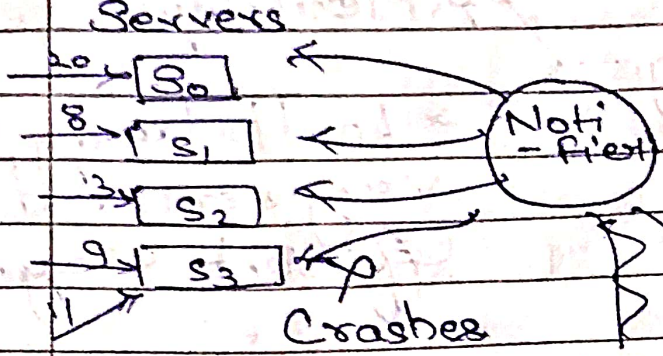
ASYNCHRONOUS PROCESSING

Now: —



x One shop goes down

Then list will not work, so we need persistence of data; so this list will be stored in the database.

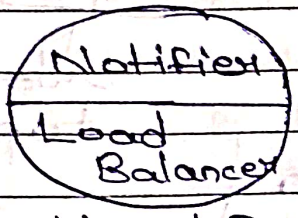


ID	Content	Done
1	Peperonni	Y
2	Hut chese	N
3		
4		

Every 5 second check server is live or not and queries db for not done order and picks and distributes them to working ones.

Issue → Duplication
Suppose 3 went to S_3, S_1.

Sol →



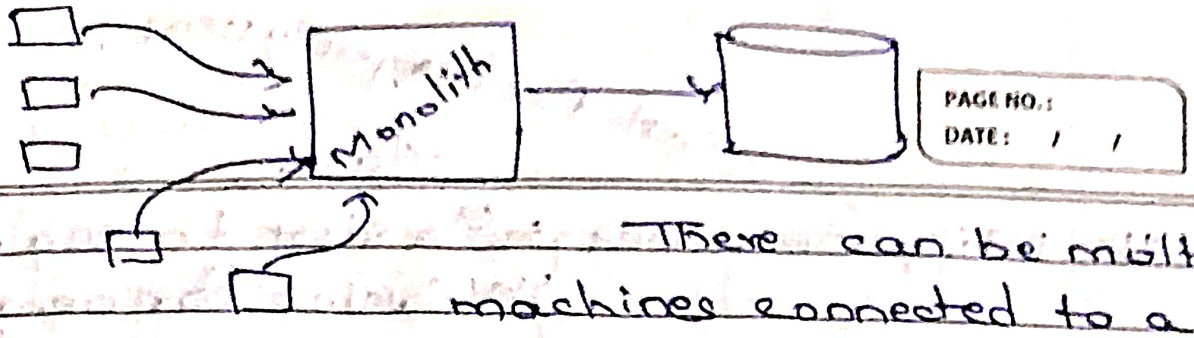
Heart Beat Mechanism.

Message / Task queue - All features of heart beat, load balancing, assignment and persistance in one thing, it will be a message queue. Encapsulates all the complexity in one thing.

Libraries →
i) RabbitMQ
ii) ZeroMQ
iii) JMS

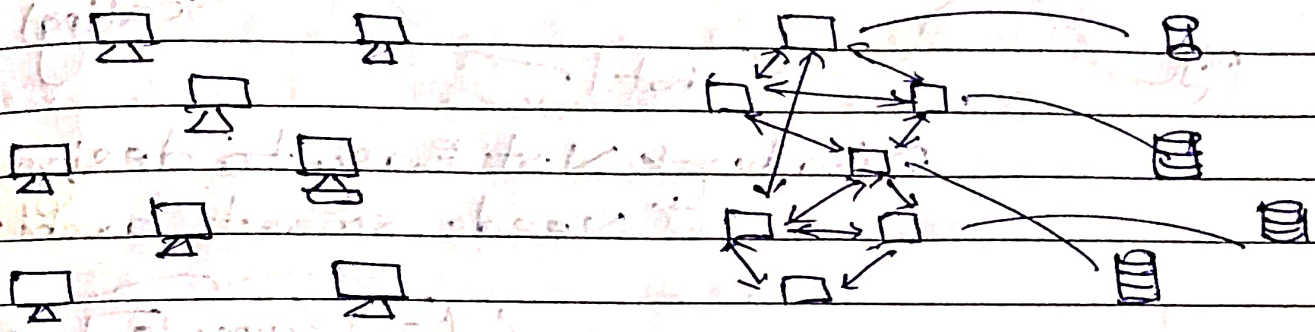
Monolith vs Microservices :-

Next pg...



PAGE NO.:
DATE: / /

There can be multiple machines connected to a same monolith, so we can horizontally scale out with monolith. Concept of monolith, a big machine running everything is not true.



Microservices — A single business unit, all data and functions relevant to a unit are putted in a single unit. And Every db usually talk to their own databases. Client talks to gateway and gateways to databases.

Monolith
- Good for small teams.
- Less Complex.
- Less Duplication.
- faster.

Advantages

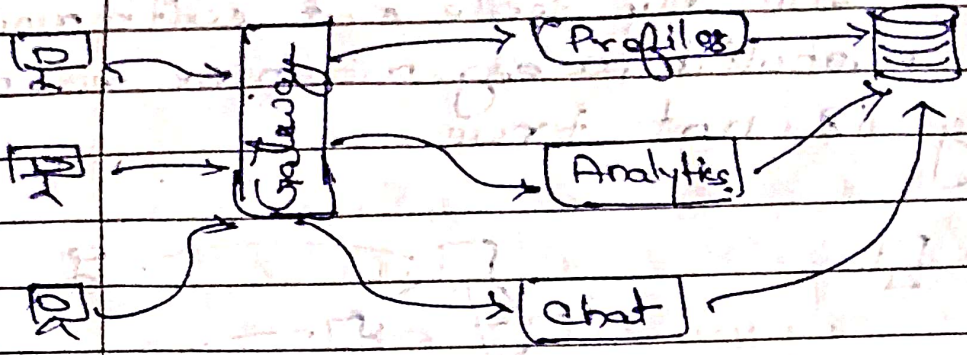
- More context needed (on more addition)
- Deployments Complex (Every change triggers need deployment and requires monitoring)
- Too much responsibility on each server / Issue ∝ whole

Dis. Adv.

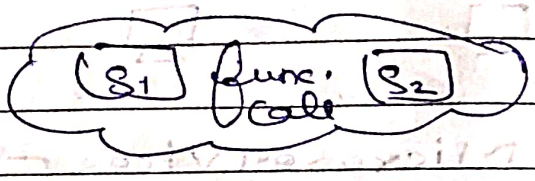
→ Best for large systems.

Adv..

MicroServices → Easier to scale
→ Easier for new team members.
→ Working in parallel (easy).



Disadv → Not Easy to design.
→ Needs smart architect.



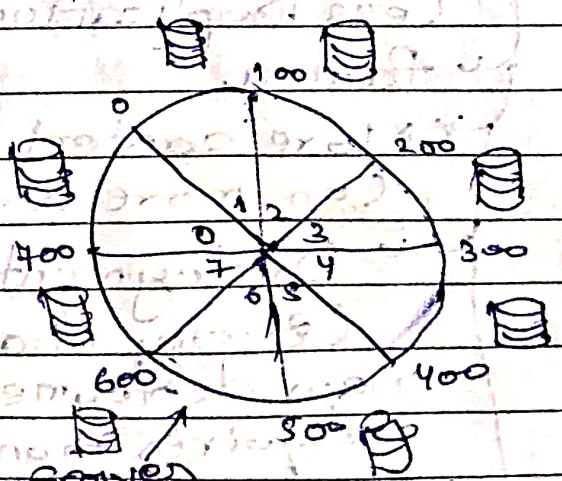
StackOverflow :- Monolith
Google, fb, etc :- Microservices.

* Database Sharding :-

Optimizing
Indexing
NoSQL
→ RDBMS only.
Should not use

→ large data

So, we will use sharding.



Horizontal Partitioning (Data into pieces)

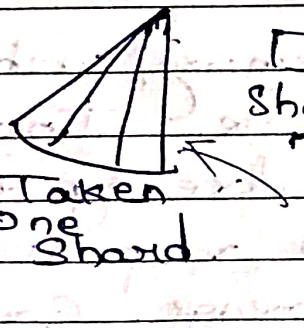
Vertical Pa..
→ Break into Columns.

Servers here are database servers.
→ Consistency is imp.
→ Availability.

PAGE NO.:
DATE: / /

Shard db on the basis of what?
→ maybe shard on locations
x ⟶ x, shard city.

Problem → ① Consideration for joins across shard.
② Mem Cached (db that uses consistent shading/joins)
③ Fixed no. of shards.



{ Hierarchial Sharding }

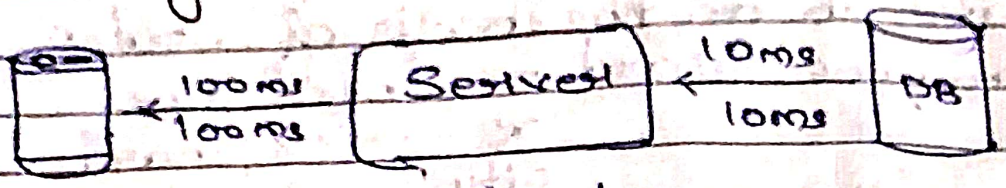
Indexing on shards. (faster)

Now, if electricity goes off....
↳ Solution → Master Slave Architecture.
Slaves → Continuously pulls the master and read from it.
(Helps to avoid single point of failure)

No SQL databases → internally use concepts like sharding only.

PAGE NO.:
DATE: / /

*Caching in Distributed Systems:—



Instagram

Total time = 100 + 10 + 10 + 100
= 220 ms.

Similar user, asking for similar feeds.
Suppose (Two young indian guys who loves cricket)

⇒ Querying the local memory/cache for the new user. Instead recalling, show the already stored result. Reducing the repeatable work through storage. Caches are much faster as compared to querying a db.

$\frac{\text{Cache query time}}{\text{DB query time}} = \frac{100*2+1*2}{100*2+10*2} \approx 10\%$ Saving

Also can be done on the client side via local cache if user goes and comes again and go... then instead making a promo call, use local cache.

Summary: Caching → Avoiding repeated work through storage.

Why not to take whole db and put into cache?

Ans:- Makes sense only for smaller db's.

Engineers optimise cache storage by trying to keep data that will be accessed next.

Cache Policy
① How do I manage writes?
② What do I evict on overflow?
↳ LRU, LFU, etc...
(for limited policies).

Thrashing → Useless work (Poor cache policy and can hurt performance)

addLike() ← dont need live addon to the user but after some intervals, the right numbers. Issue → (Data not true)
↳ Financial System
→ false promise.

This is Eventual consistency → Eventually the cache will be consistent and eventually cache will be consistent, but when??
↳ Based on cache policy.

Cache Placement
Either to place in server or Database or Global system (external cache) Independent one.

Which to use??

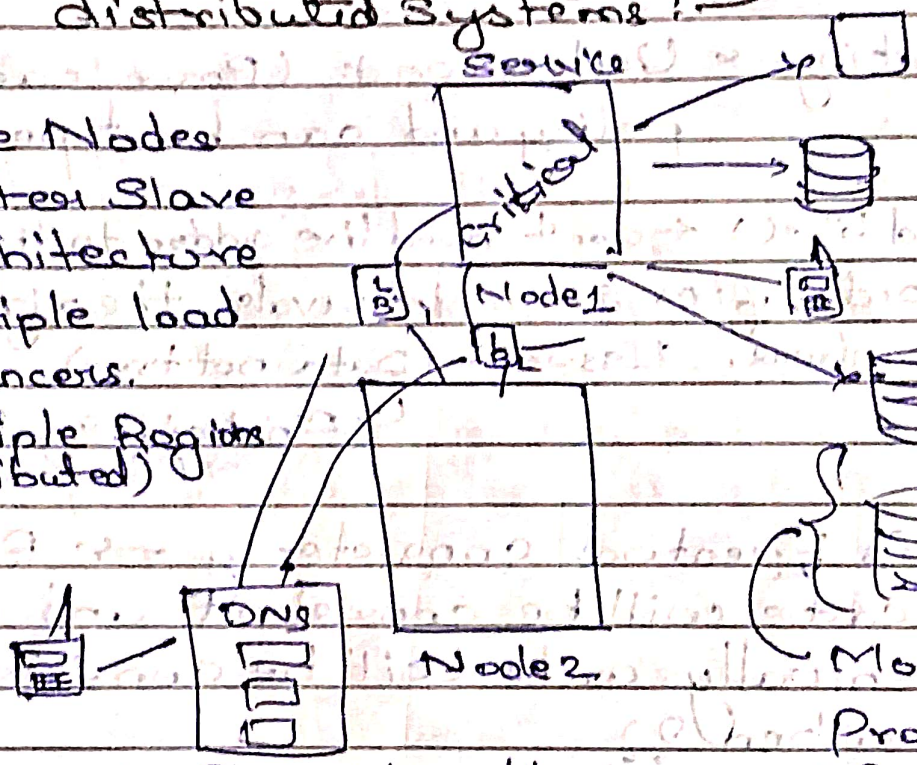
In Large Scale Systems, all 3 are applied.

As a SDE → Distributed Cache (Redis)

Depending on the system, coe are meant to make right choices.

How to avoid single point of failure in distributed systems :→

1) More Nodes
2) Master Slave Architecture
3) Multiple load balancers.
4) Multiple Regions (Distributed)



S.O.P Example
↳ Humanity on earth

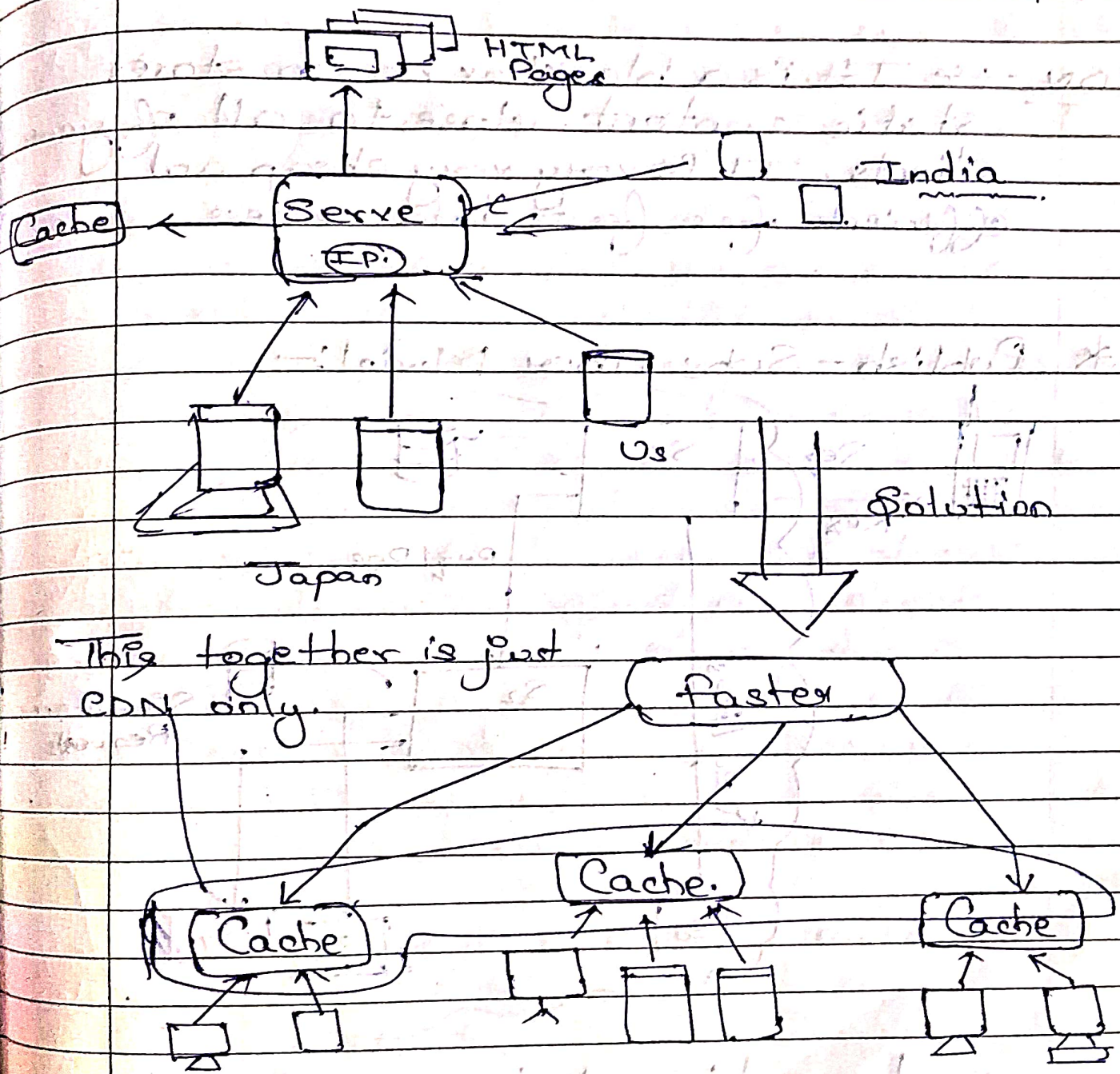
Netflix uses chaos monkey which randomly goes on production and takes down one node.

PAGE NO.:
DATE: / /

*Content Deliverey Network:-

(Makes the system cheaper and faster)

Pre-requisites — i) Caching.
ii) Distributed Systems.
(what is a client/server)



CDN → i) Boxes close to users.
ii) Follows regulations.
iii) Content updated from server.

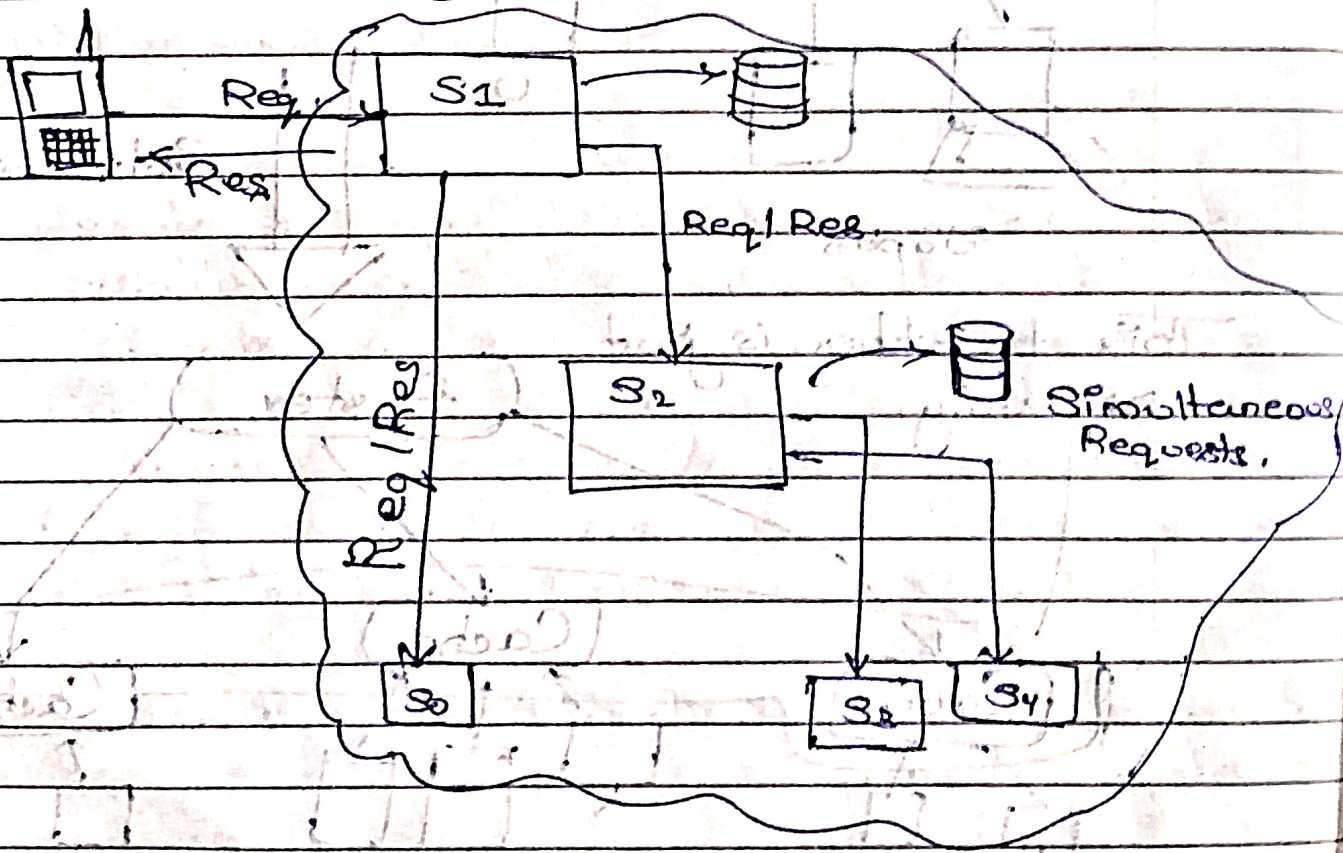
PAGE NO.:
DATE: / /

Example → Amazon Cloudfront
- Cheap
- Reliable
- Easy to use

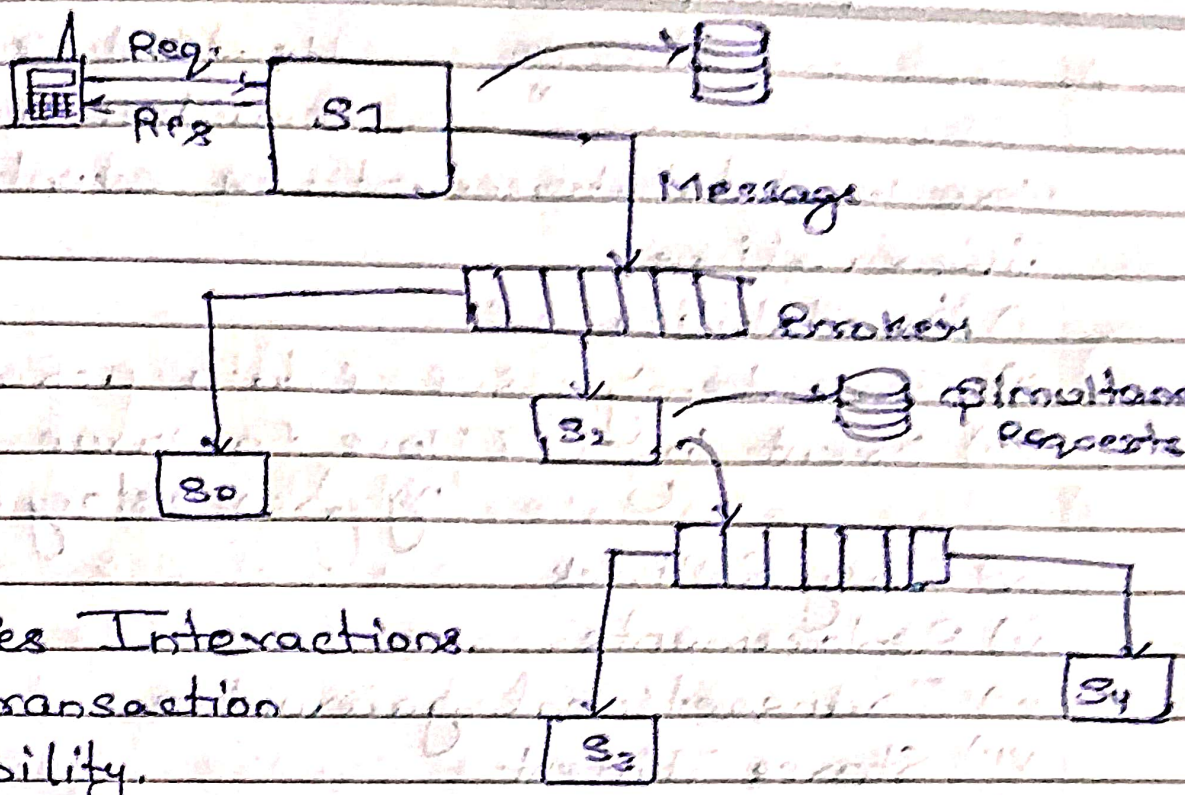
→ It has integration with amazon S3.

CDN → It is a blackbox which stores static content close to all of your clients. It is very very cheap and efficient for fast data access.

* Publish – Subscriber Model :-



Long-timeout issues, Pol. is → next page.

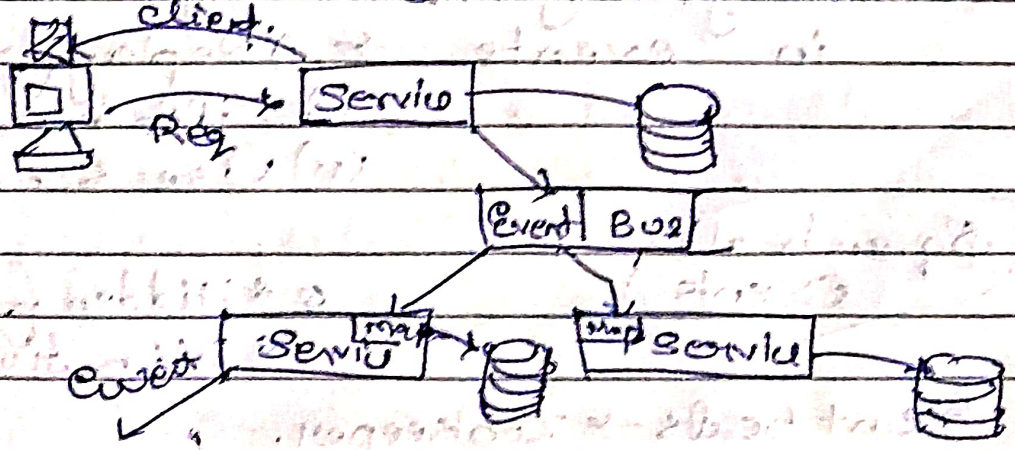


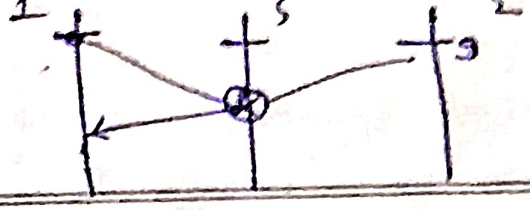
⇒ Simplifies Interactions.
⇒ Some transaction capability.
⇒ Easily Scalable

Issues → i) Poor Consistency
ii) Cannot use for event critical systems such that there is just a success or failure
iii) It doesn't gaurantee idempotency.
ex → 50 rs. removed multiple times.

Twitter Uses this architecture.

* Event Driven Architecture →
Git (example, react, node.js, game servers uses even driven architectures.





PAGE NO.:
DATE: / /

9 10 Headset Example
x (counter strike)

Here local databases stores event informations.

i) Availability ✓
ii) Consistency issue ✗ (data across same needed)
iii) Event log (Move to error log from history in future)
iv) Easy Rollback
v) Replacements
vi) Transactional gaurantee.
vii) Stores Intent
~~viii) N/A to gatecoays~~
ix) Lesser Control

~~Event~~

Event Bus architecture is just what people want to consume and what people don't want to consume. Problem is that maybe I want some services to consume your events and want to stop some of the services from touching events.

3 ways to get to a particular point in events → i) Replay from start
ii) Diff based
iii) Use something like undo.

Squash all events.

* *Hidden flaws.
* Migration tough.

Heart beats + Zookeeper.

PAGE NO.:
DATE: / /

* Tips for System Design Interviews :-

1. Don't get into details prematurely.
2. Avoid fitting requirements to a set architecture in mind.
3. KISS Principle - Keep it simple stupid! Remember to look at the big picture and avoid too many hacks while solving.
4. Have justifications for points you make. Don't use buzzwords or half-hearted thoughts in your design.
5. Be aware of the current solutions and tech practices. A lot of solutions can be purchased off the shelf which simplifies implementation. You should be able to argue for a custom implementation with its pros and cons.

Points evaluated during interviews :-

1) Clarity of Thought
   - a-> Express your thoughts in a clean manner.
   - b-> Justify your decisions. Critical reasoning and argument are key to a successful software design.
   - c-> When faced with a problem, use standard approaches to migrate it. For Example, say you are faced with an availability problem. State that replication and partitioning help increase availability in general, and move to offer a solution.

d). Don't make points without thinking them through. Half-hearted attempts at solving problem are frowned upon heavily.

(2) Know About Existing Solutions:

a) Stay up to date with the current solution in the market. This includes: products and design ~~purposes~~ practices. If NoSql, if no Sql is been adopted left right and centre, you need to be aware of it.

b) Know when to pick a solution vs. building something custom. If you name a product, you should be (generally) aware of the feature it provides.

c) Design practices enable you to meet custom requirements. Examples are decoupling systems, load balancing, sticky sessions, etc.

(3) Flexibility

a) Switch your targets as requirement shift. If the interviewer wants to know about the particular part of the system, do it first.

b) Never have a set architecture in mind. We all try to fit requirements to the system, but only after it has been shaped by the initial ones. A rigid attitude creates a brittle architecture. It will break before you do.

PAGE NO.:
DATE: / /

c) Take a step back at times to make adjustments to the general architecture. Being focused on one part can narrow our vision and bloat those areas. There will be components which can be extracted out and extended to the rest of the system.