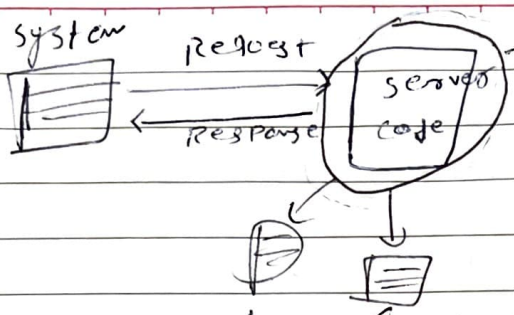
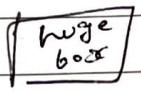
System Design

Day 1



Buy More Machines — vertical scaling (Request distributer)

① Buy Bigger Machine — Horizontal scaling

Horizontal	vertical
	
① load balancing required	① NIA
② Resilient.	② single point failure.
③ Network call (Now require to call one system to other)	③ Inner process communication
④ Data consistency issue	④ consistent
⑤ scales well (as users increase)	⑤ Hardware limit

we can't stop every Machine when call fails so, faster May be system runs in same Machines.

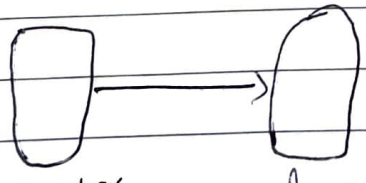
Day 2 Distributed system (High level Design)

- Recruitment
- seperation of concerns
- Fault tolerance

vertical scaling
① Optimise process & increase throughput using same resources.

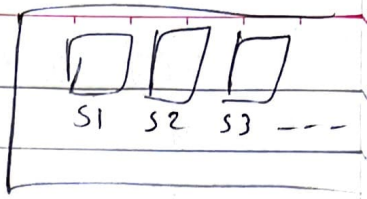
Resilience (Recovering from failure quickly)
② Preparing beforehand at non-peak hours.

backup
③ Keep backups and avoid single point of failure



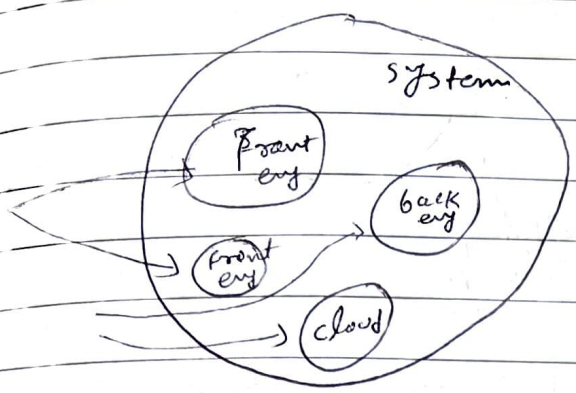
Page No.
Date / /

④ hire more resources } horizontal scaling.

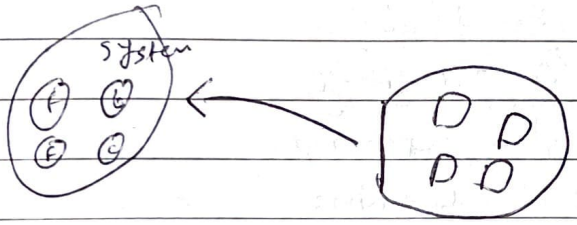


⑤ Microservices

Ex:

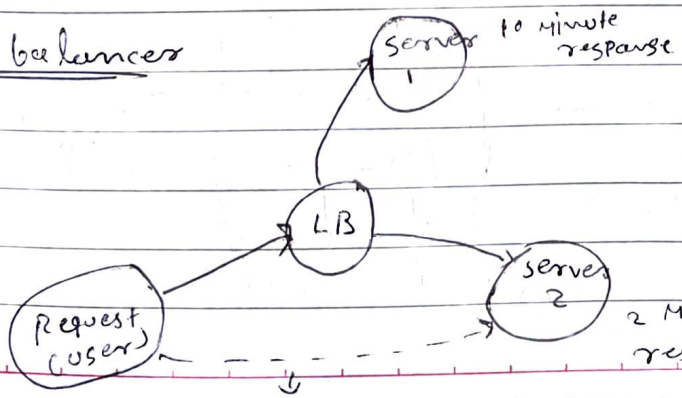


⑥ Distributed system :- when there is an issue in main system, the other part of the system are ready to serve.
- its a fault-tolerance.



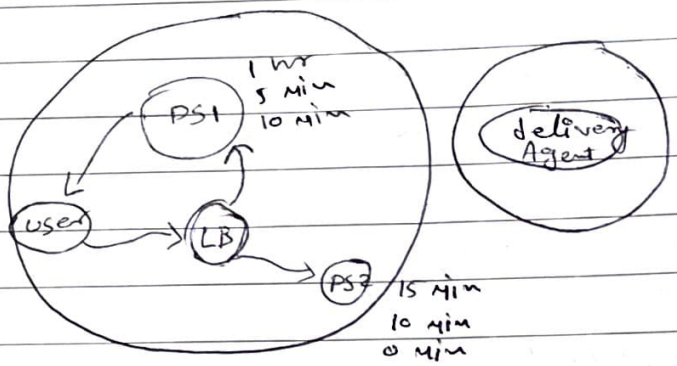
for ex. facebook has the local server everywhere, so peopl from india, get the connect with the india server, and respond quickly, instead of taking time to response in connect with USA server.

⑦ Load balancer



⑧ Decoupling

Ex: pizza shop



→ we separate delivery & order, so we can scruting, Analyze, Audite, report, etc...

⑨ logging & Metrics

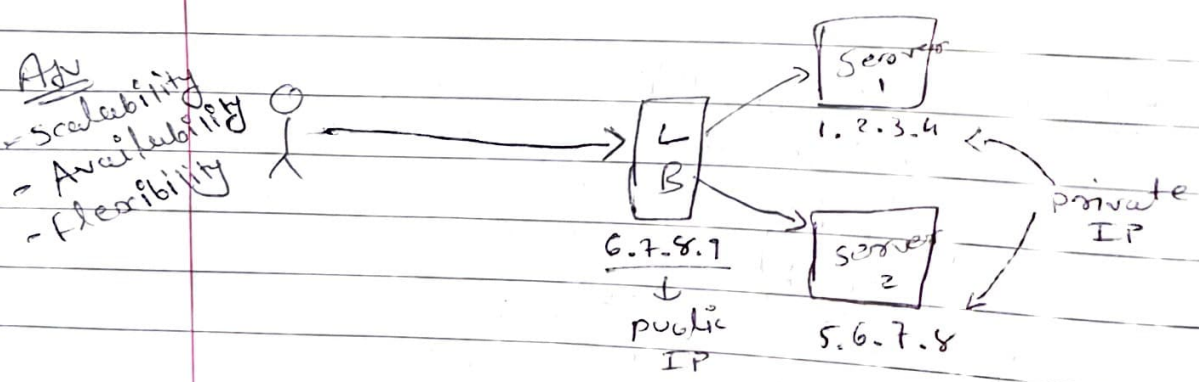
⇒ to check every system we decoupled.

1. Analyze
2. Auditing
3. reporting
4. Machine Learning

⑩ Extensibility :-

---

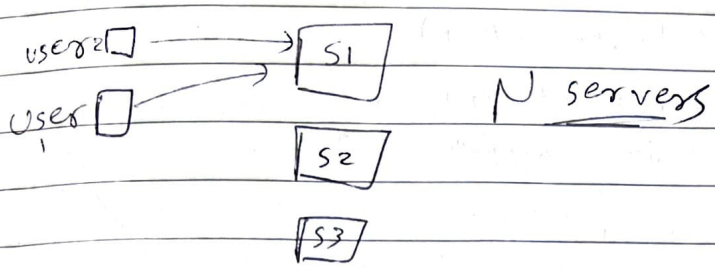
* Load balancer :-



DOMS | Page No.
Date / /

Day 3 Load Balancing :-

* Consistent Hashing



Load balancing.

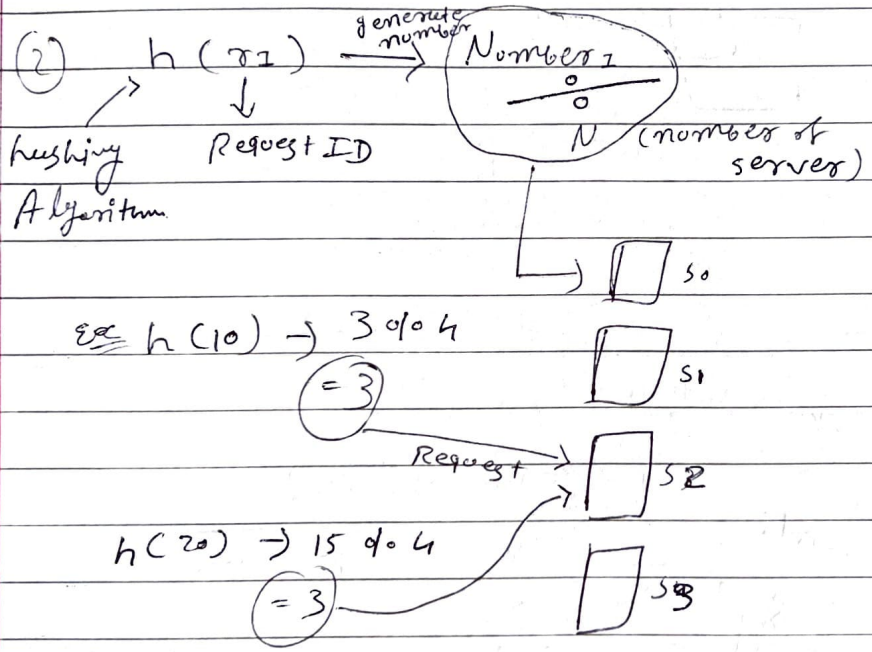
(1) Request ID ⟶ 0 to M-1

↳ when user send a request it randomly generates number from 0 to M-1

(2) h(r1) → generate number → Number 1 (0 to N (number of server))

Hashing Algorithm

Request ID



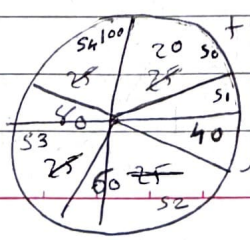
eg h(10) → 3 % 4 = 3

h(20) → 15 % 4 = 3

Now, if we want to expand the system, but it is more biggest issue why?

4 - server ⇒ 25% each

5 - server ⇒ 20% each



loss add | add lost

5 + 5 + 10 + 10 + 15 + 15 + 20 + 20 = 100

25 - 15 = 10

5 + 20 = 25

Change every entire workspace

It's a problem because

for ex

we hash h(userID) by user ID.

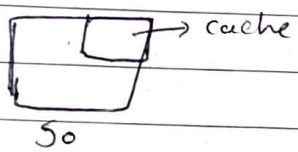
Same user request same server again & again, so server store his data from database to local cache, so time & long saves.

but, we add new server the cache of server changes. it's an issue. (useful cache is dump due to system change)

Ex h(darshan2754)
h(darshan2754)
h(darshan2754)
||
||
So on

⇒ 20 % 4 = 0

So, the same userId always points the same server, any we set the cache in the server.

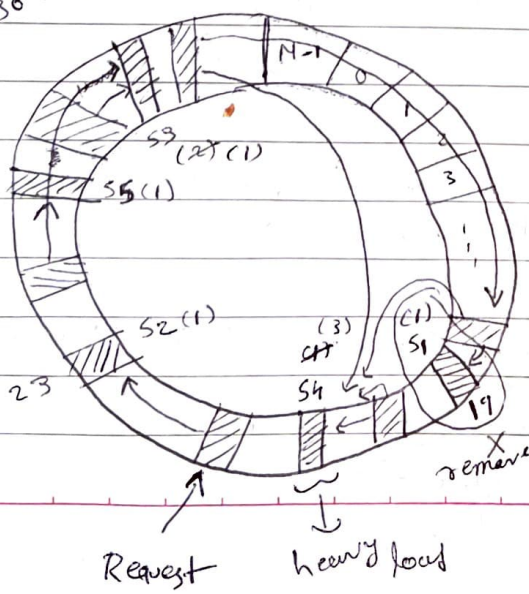


Solution

① Traditional Methods:

0 Request ID → h(ri)

h(0) % 30
= 49 % 30 = 19



② server ID, total M

h(0) % M
h(1) % M
h(2) % M
h(3) % M
h(4) % M
— server

h(5) % M
— New server

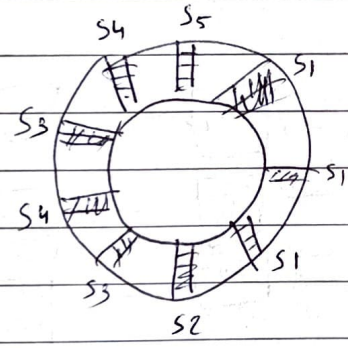
DOMS | Page No.
Date / /

We can do much better way.

Instead of hashing only 4 server, we hashesh each servers no. of server times.

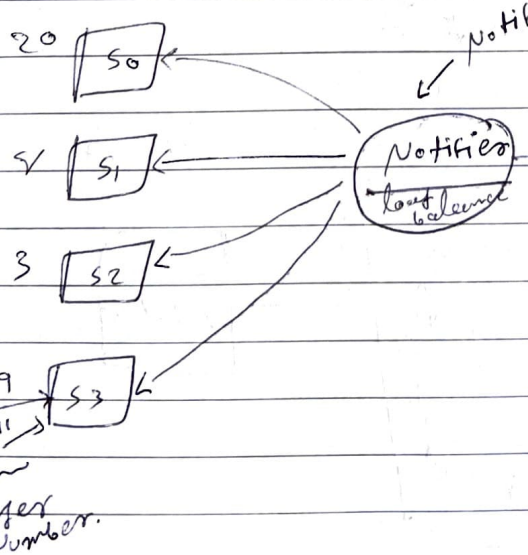
$h_1(0)$ %M	$h_2(0)$ %M
$h_1(1)$ %M	$h_2(0)$ %M
$h_1(2)$ %M	$h_2(0)$ %M
$h_1(3)$ %M	$h_2(0)$ %M
$h_1(4)$ %M	$h_2(0)$ %M
S1	S2

So, on.



server distributed Multiple times any request divides.

* Message Queue



order Number.

Ex. If Monolithics is we deploy our system in Aws any compiler database any live our system.

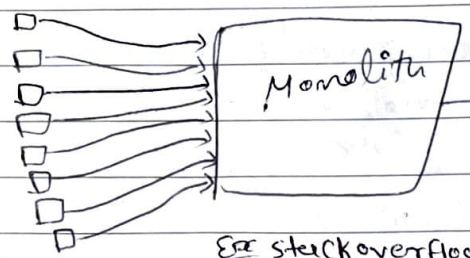
DOMS | Page No.
Date / /

Day 4

Monolith vs Microservices

Adv.
- less team
- less complex
- less duplicate
- faster

dis
- More content requires
- complicated deployment
- Single point of failure.

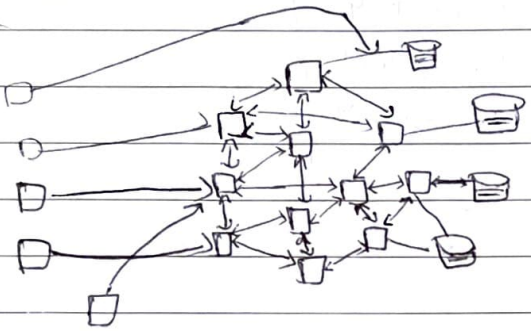


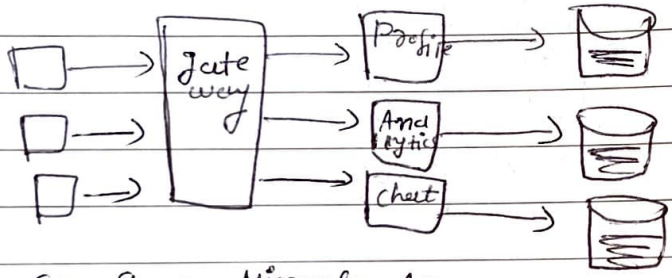
Ex stackoverflow

tight - coupling in development.

Adv.
- easy to scale
- easy for new member
- working in parallel
- easy to reason about

dis
- Need skill Architechture

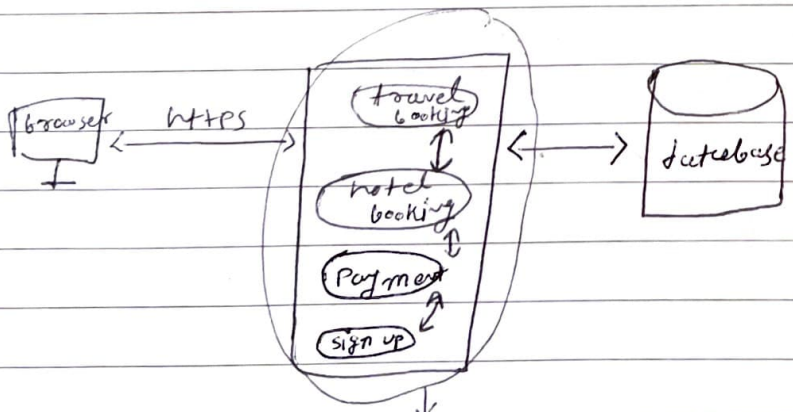




Ex. Google, Microsoft, Amazon, --- etc.

~~Database strategy~~

* Monolith :-



All this features is tightly coupled.

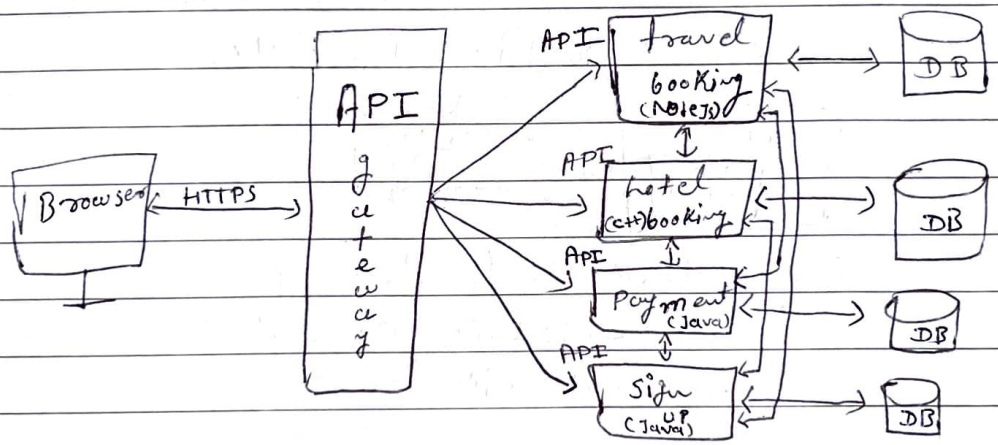
Microservice Implementation ⇒ Kubernete

DOMS | Page No.
Date / /

⇒ The problem is that for ex we want to change the features of hotel booking any change the database, then we can't do it, we need to change everything, every features. So, it's scalability issue.

Adv	dis
- low complexity	- Rigid
- easy to deploy	- hard to scale
	- slow performance
	- singl point of failure
	- slow continuous deployment

* Microservices

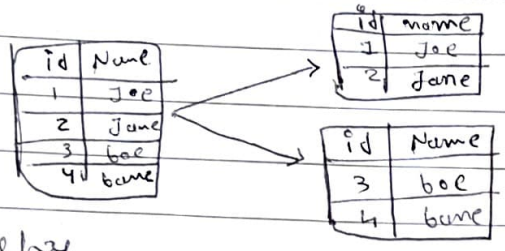


Adv.	dis
- loosely coupled	- high complexity
- Agile & flexible	- Consistency
- Independent Development	- Automation
- Independent Deployment	- debugging
- Fault Isolation	
- Mixed technology stack	
- Granular scaling	

Day - 5

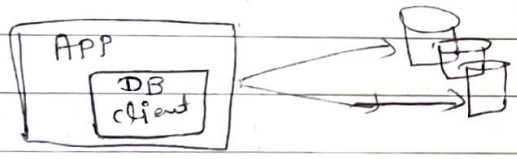
Database Sharding :-

- Splitting data into smaller chunks or shards, where each shard / chunk can be on different machine.

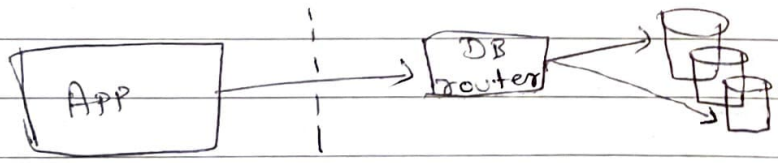


- why (complex solution. so, do if it's needed.)
  - data may be too large to fit in single machine. Provides horizontal scaling of database.
  - Distributed processing to speed up computation.

- How is it achieved?
  1. Application level sharing
     - Database client picks relevant shared for reading / writing data.



  2. Database level sharding
     - Router / config present in database and not with client



- Sharding strategies - Hashing
  - computes a hash of the shared key field's value. Each chunk is assigned a range based on the hashed shared key values.
  - while range of shard key may be "close" their hashed values are unlikely to be on the same chunk.

- Sharding strategies - Ranges

- divide data into ranges based on the shared key values. each chunk is assigned a range based on the shared key values.
- A range of shared keys whose values are "close" are more likely to reside on the same chunk.

Ex Order Management system

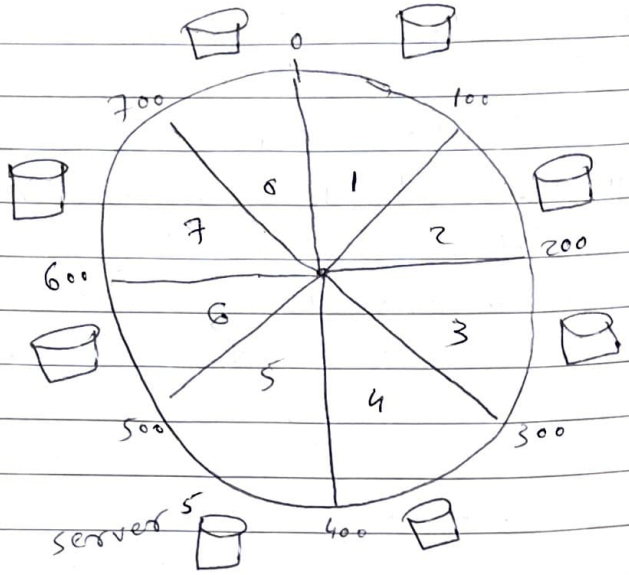
- order data of users in e-commerce site

```
Order {
    long orderId;
    long userId;
    Date date;
    Status status;
} OrderDetails details;
```

- shard on basis of orderId or UserId.
-

⇒ Database with sharding support
- Cassandra
- HBase
- MongoDB, etc...

* Issues
① Hotspotting → uneven distribution of data
- one key have large amount of data
② Redistribution of data
- New machine in cluster
③ Network Partition
- Network partition between DB machines when using DB based sharding.



(1) Horizantal partitioning with one key like user ID

(2) Vertical partitioning with column.

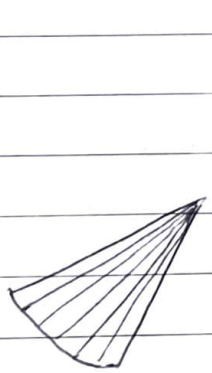
Problem:-

⇒ if we want a data from two different shards/server, then we need to pull out data any to "Joining"

⇒ MeMcached

⇒ Fixed number of shards.

!! Solution.

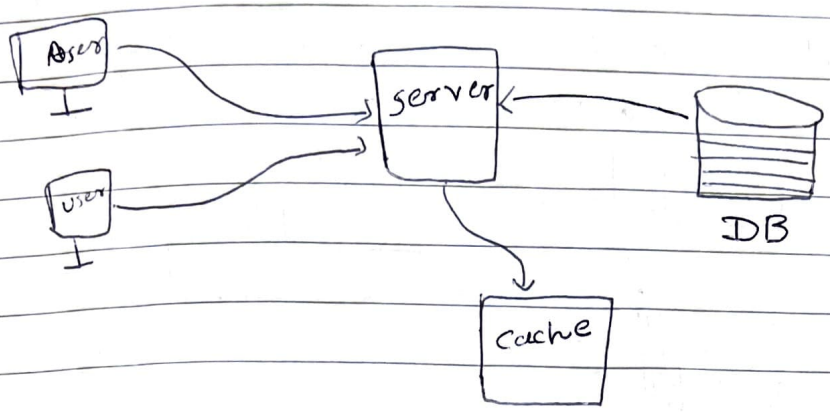


one Manager from this server Manage the different parts. It's called "Hierarchical Sharding"

⇒ If the any one server fails, we use Master-Slave, so, slave pulling data from Master, so we have slave to get the data.

Day-7

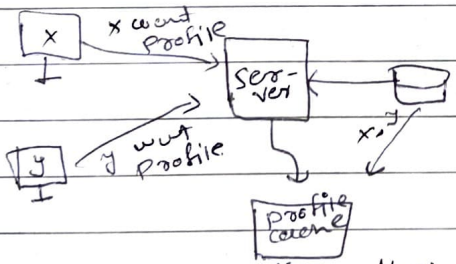
What is Distributed caching?



(1) Reduce Nw calls
(2) Avoid recomputation
(3) Reduce DB load

* Cache Policy (Maintain cache properly, any remove data, any update)
  (1) LRU :- remove old entry from the cache.
  (2) LFU :-
  (3) sliding window policy

* (1) Disadvantage.
  (1) Extra call
  (2) cache thrashing (when we have less cache)

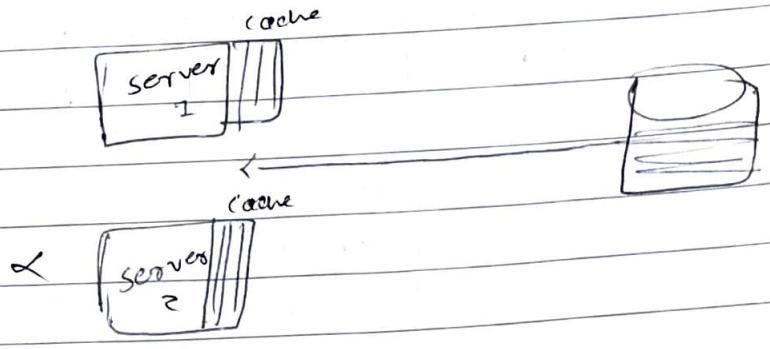


x, y, Now again when x get the data, it's finding in cache, its thrashing.

(3) Data consistency.
=> when user update profile in database, any when they tried again to get updated profile, any it it's not changing in ~~database~~ cache, then it gives old profile data to users.

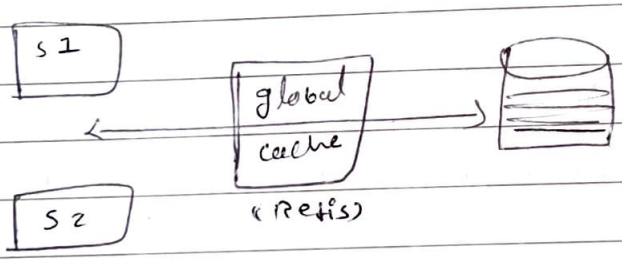
Date / /

* where we can place cache?
  ① it's close to the database
  ② it's close to the server



⇒ if we put the cache in memory of the server, then when server fails, cache also fails, cuz the cache of server is also not sync with server 2, so we can't get the all information that user wants.

⇓ solution (global cache)

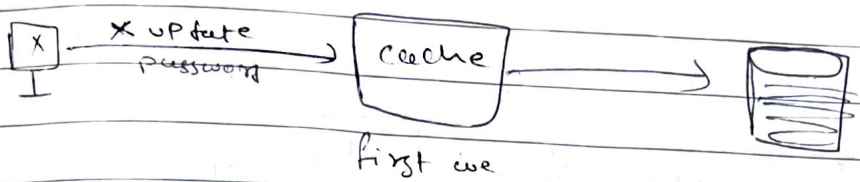


* global cache
  ⇒ faster
  ⇒ Accuracy

* How consistent data in cache?
  ① write-through
  ② write-back

DOMS | Page No.
Date / /

① write through



first we change user x's password in cache befor updating to database.

② write back

⇒ we change update in database, avg them at the same time we change/update in cache

⇓ Problem

if we keep updating cache avg database every time, it's a performance & timing issues.

⇓ solution

we keep changes in cache for certain entries like 50 to 100, avg them after some time in one network call we update the database.

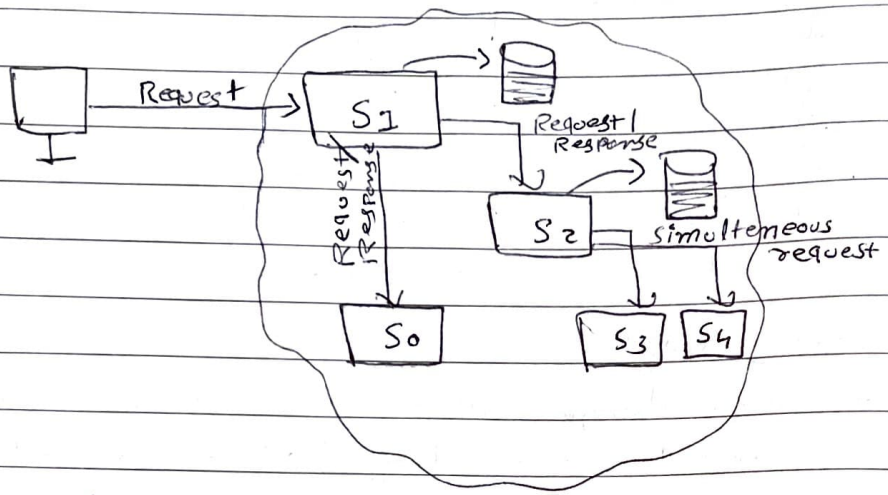
API Design

① Naming
② Parameter defining
③ Response object defining

www.xyz.com/chef-Mess/getAmin/v1

* Publisher - subscriber Model

⇒ what's the issue in microservices ?



⇒ S0 to S4 is a services

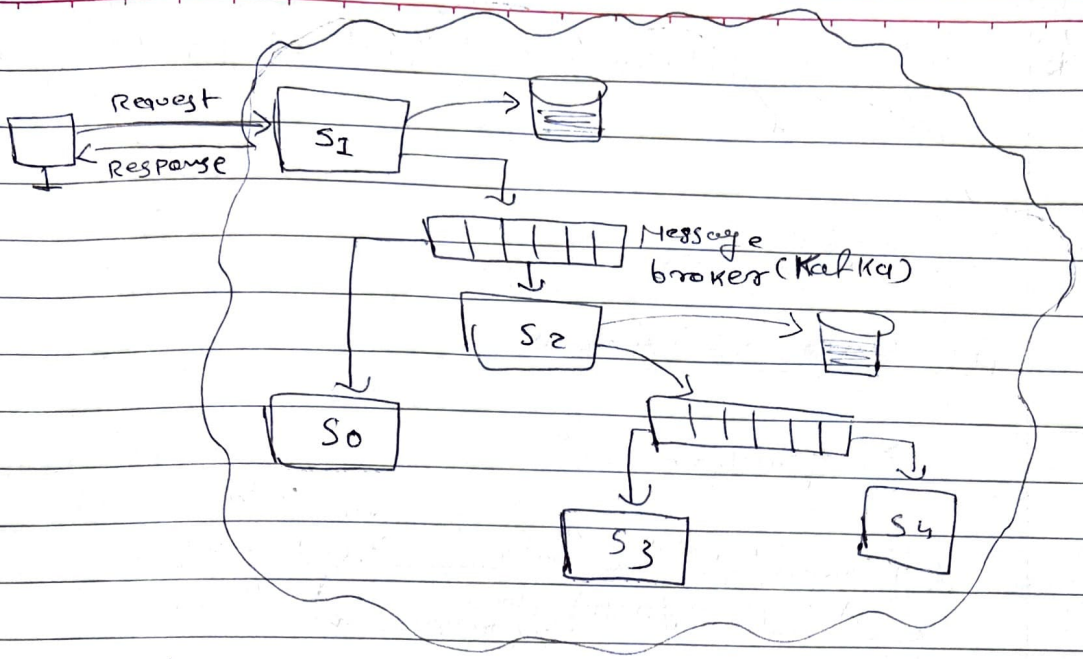
⇒ client send request to S1, after S1 has done it's changes, S1 needs to send two Msges to S2 & S0. any after S2 process, it's also needs to send messages to S3, S4. So, it's a "strong coupling."

⇒ here S1, need to wait befor serving response, S1 waits for S2, S2 waits for S4 response. So, it tooks long time.

⇒ Now, for ex. if S4 fails, it send update to S2, S2 send to S1, any S1 send fail response to client. Now, client again send to S1, S1 previously made database changes, but now it's also do for 2nd time. So, it's "inconsistency of data."

* Solution :-
Publisher / subscriber Model

event driven services



⇒ S_1 ⇒ Message broker (it keeps request)

MB ⇒ S_2 (if S_2 fails ~~ready to serv response~~, ~~then MB serv it~~ S_0, MB keep request, serv when S_2 is ready, then serv them a request.

S_1 ⇒ client (success).

Adv :-

- simplify interaction
- single point of failure
- easile scalable (we can add other services easily serv attached to MB.)

Dis :-

- Poor consistency (we can't use for the secure services like payment serv all)
- Idempotency still required.

Ex Twitter usage Public-subscribe Model.

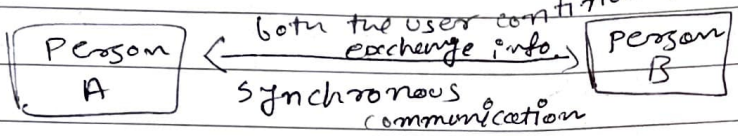
Day-9

* Why do database fail ? Anti patterns to avoid

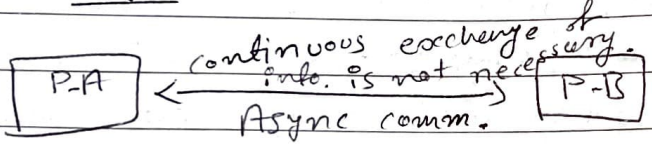
Message queue

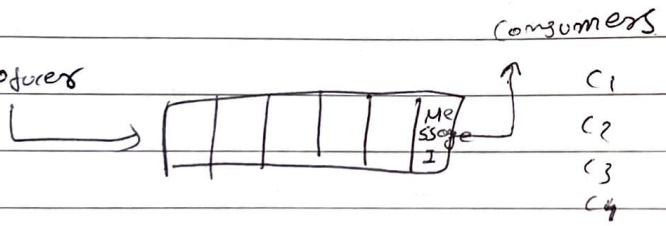
① Sync vs Async

→ synchronise



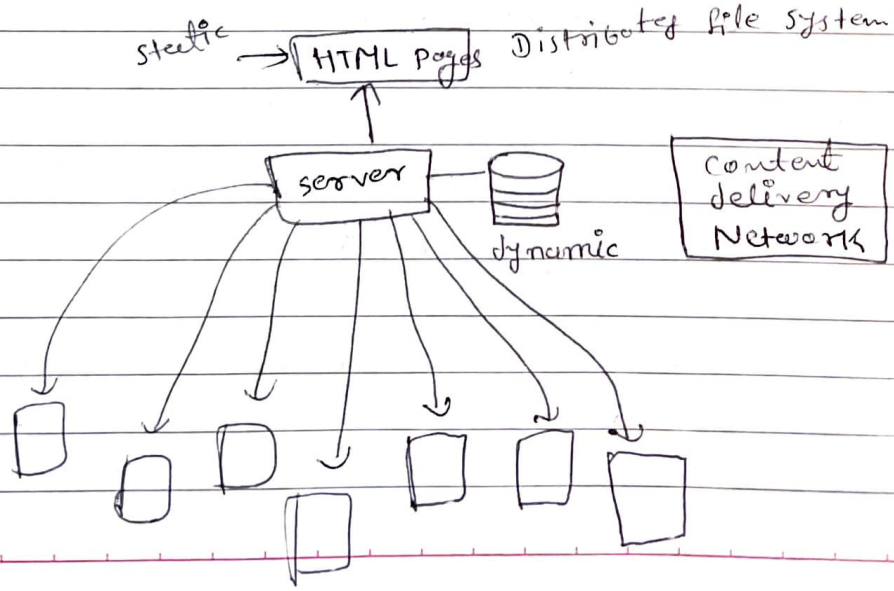
→ Async





Day-10

* Content Delivery Networks



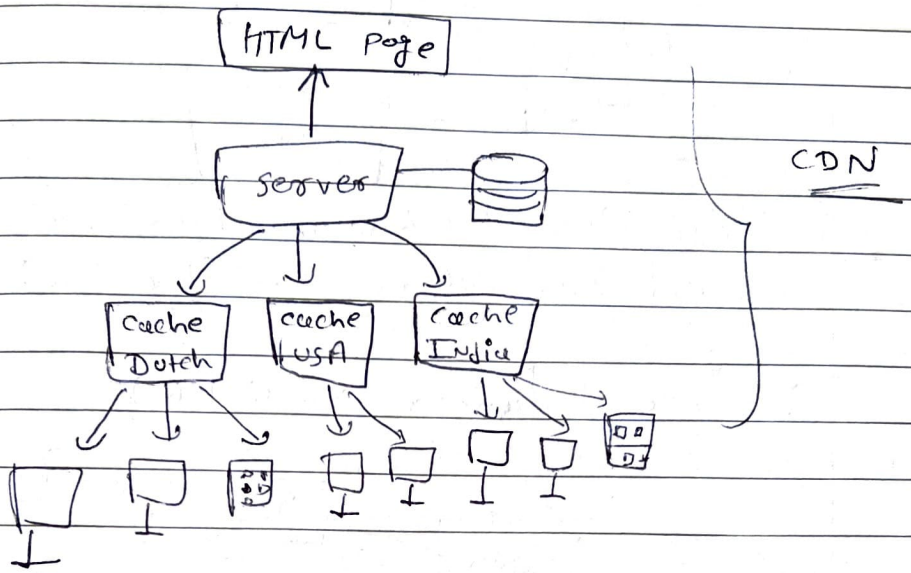
we need those things in our website.

1. caching
2. customized data (device, location)
3. fast

↓↓

solution

* solution 1 but it has some drawbacks.



difficulties

- Available in different countries
- follows regulations
- serves the latest content.

* **Content delivery N/w.**
  - hosting boxes close to users
  - follow regulations
  - Allow posting content in the boxes via UI

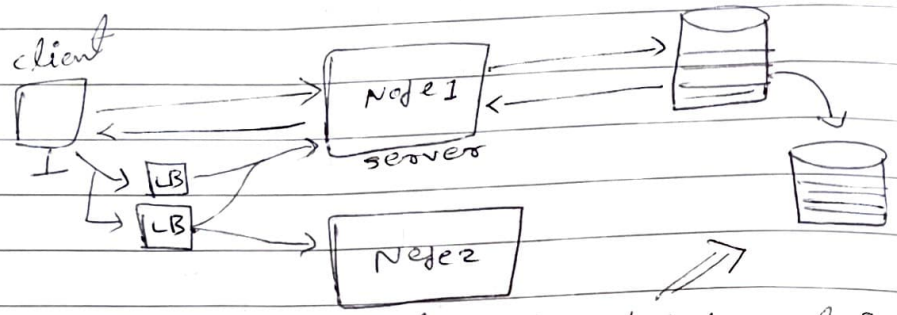
* CDN services => S3

Adv of CDN

- speed
- Low
- uptime
- Security

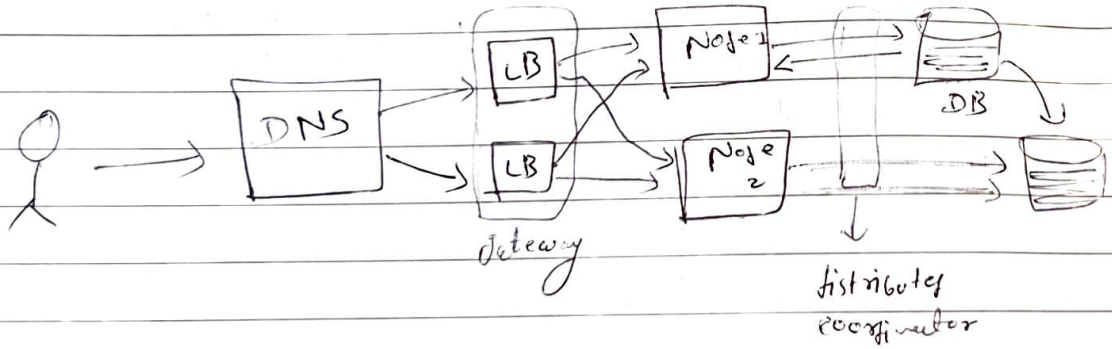
Day-11 * How to avoid a single point of failure in distributed system?

① Add more nodes if server fails

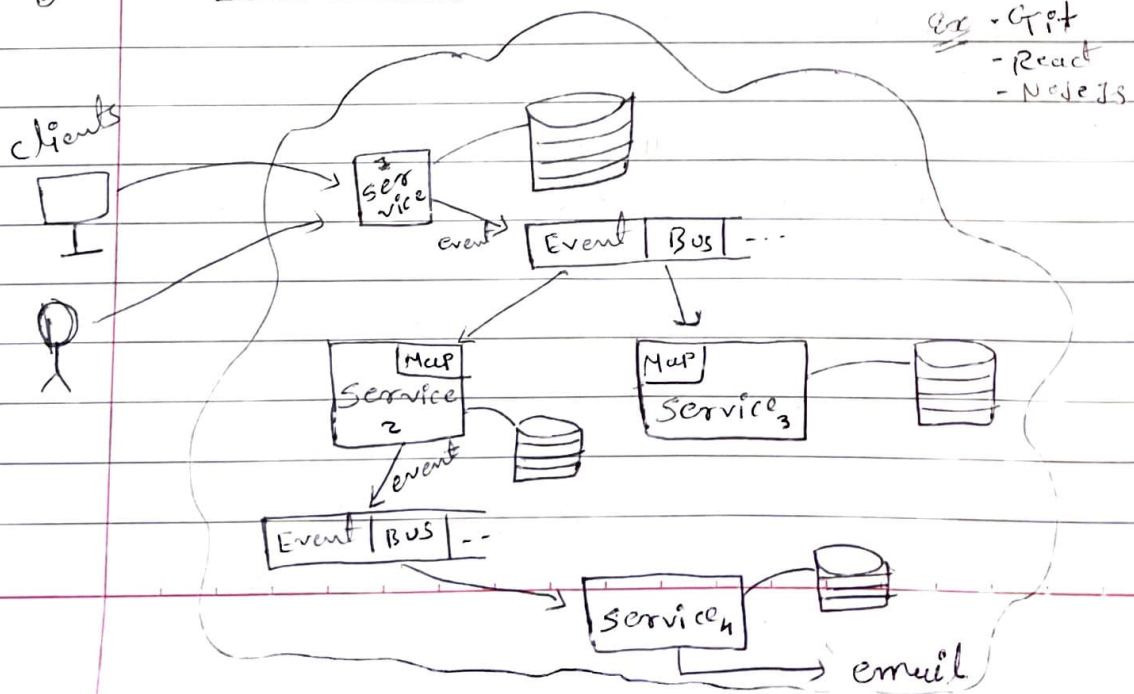


② Add other database if Main database fails. (Master-slave Architechture)

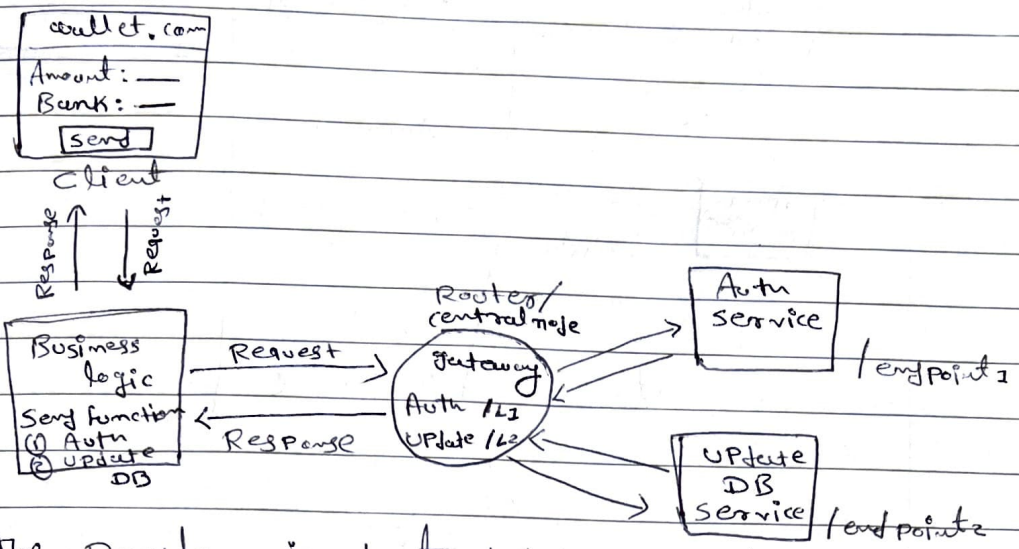
③ load balancer (Multi LB)



Day-12 Event-driven Architechture



⇒ let's understand Request driven model (Microservices)



⇒ the problem is that this is synchronise, so after completing one service, it takes & execute other service.

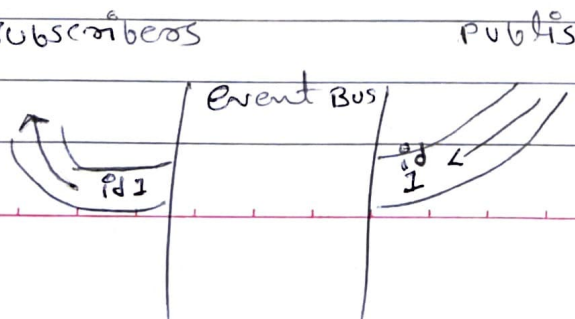
⇒ The issue is when we need to save ~~the~~ taken care of the previous service.

for eg in a wallet, i send money to other user, it Authorise me, update in database, and the next service fails, so, here it update the database, but money not sent due to the other service fails, and it send the error to user it's fails. so, what about updated database?

- Recovery is difficult in Microservices.

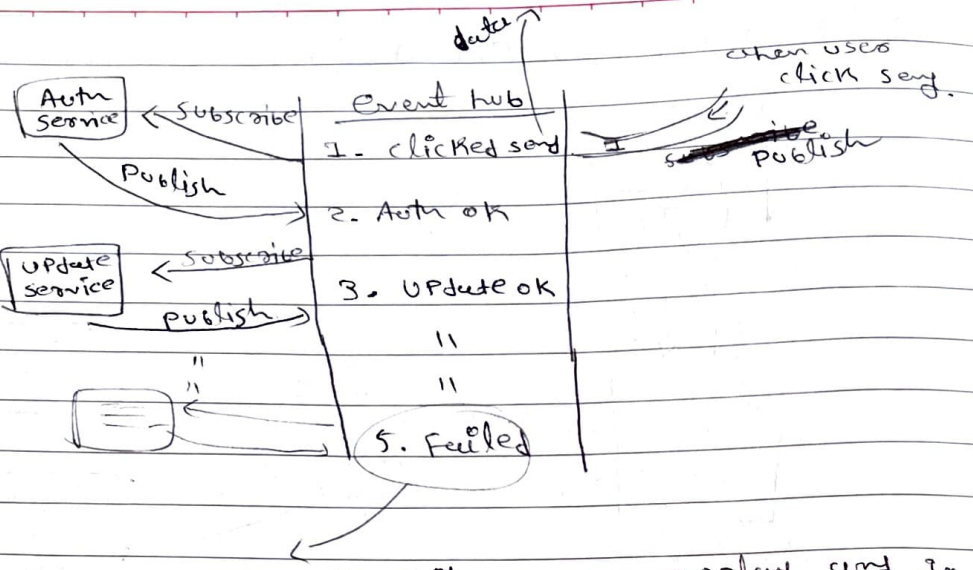
Now, let's implement EDA in microservices.

① let's understand public-subscribers model.



{ when:
where:
what:
}

DOMS | Page No.
Date / /



When it fails, we can replay, and go back to previous events, and rollback the changes.

Adv.

=> EDA has automated testing & recovery.
=> Availability
=> Easy roll-back
=> Replacements
=> Transactions
=> Stores Intent

dis.

=> Consistency
=> NIA to gateways
=> lesser control

$\log_2(10) = \log(2 \times 5) = 10$

$= \log_2(2) + \log_2(5)$

$= 1 + 2.3219$

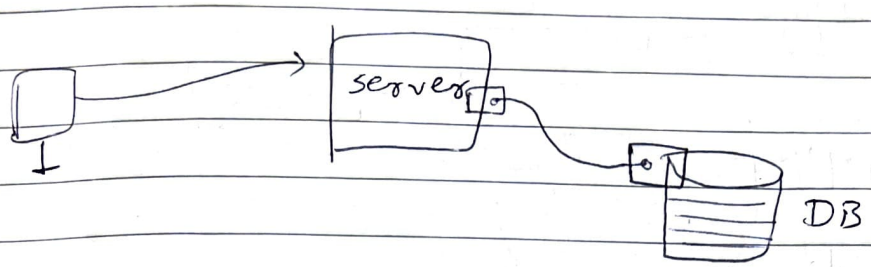
$= 3.3219$

DOMS | Page No.

Date / /

Day-13

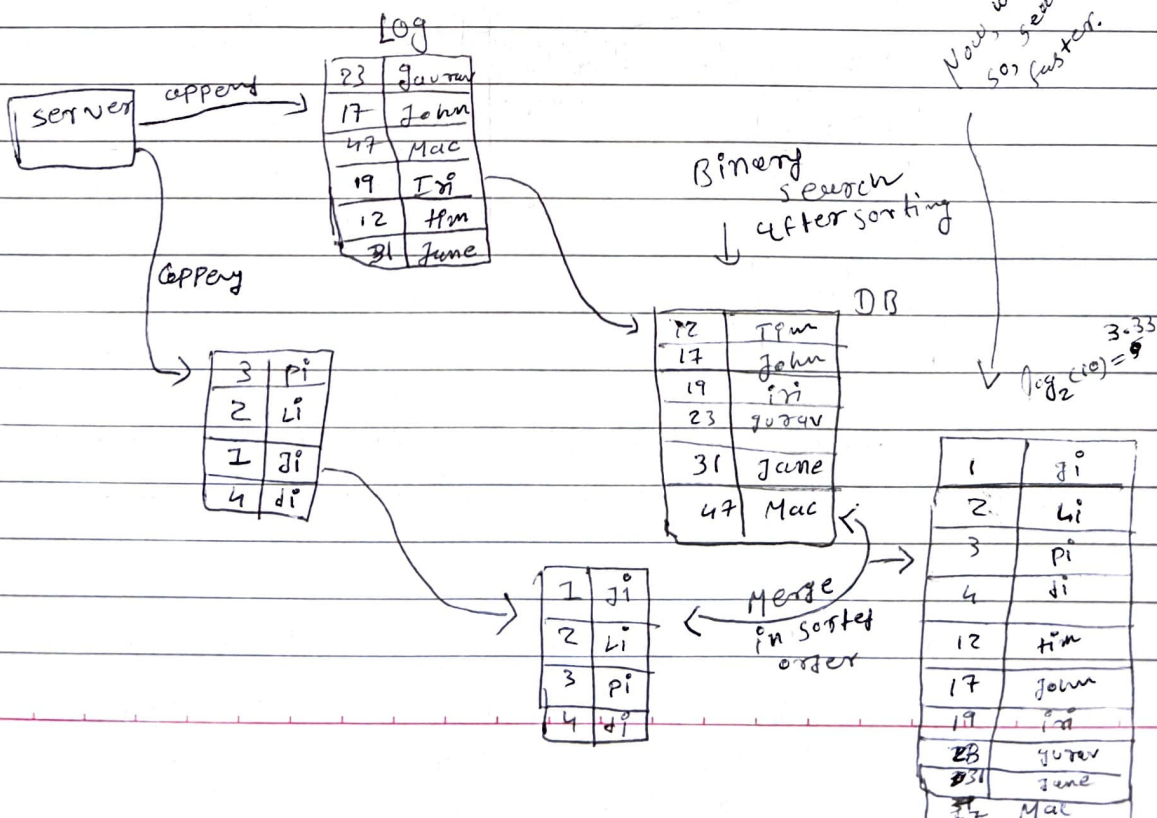
How database scale writes:-



1. Compense data queries into single queries.
2. linked list $\xrightarrow{\text{write}}$ O(1) (log)
3. Sorted array $\xrightarrow[\text{time}]{\text{search}}$ O(n)
4. we use linked list + sorted array in database to improve read fast.

Adv.	dis
1. lesser IIo operation	1. Additional memory
2. fast writes	2. slow reads.

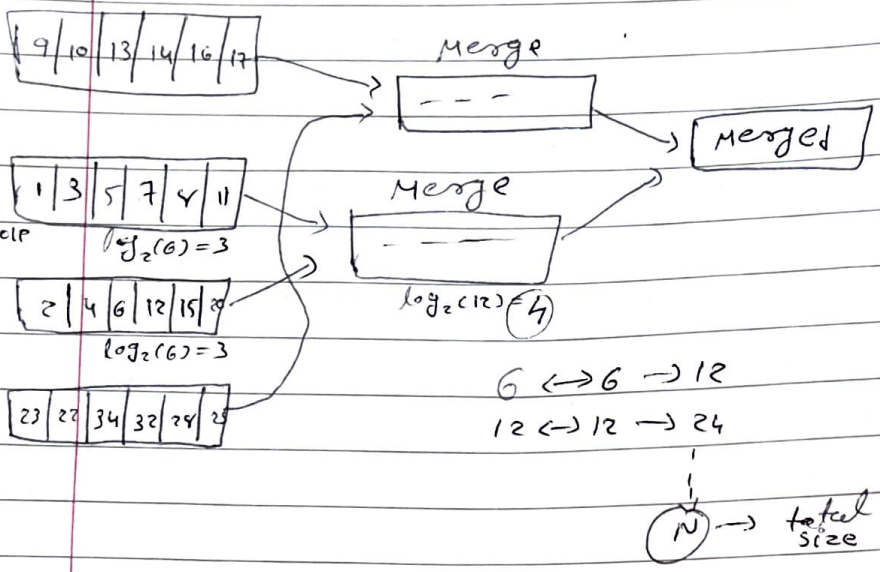
⇒ So, first we sorted the data, and then persist.



Bloom Filter

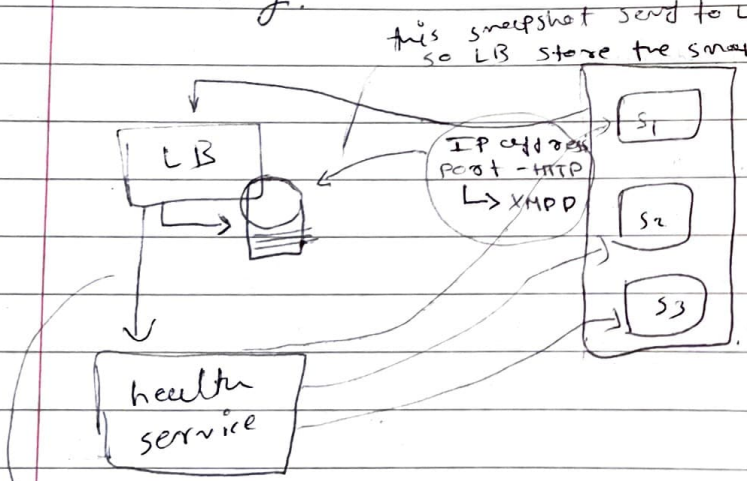
DOMS | Page No.
Date / /

So, in the background.



Day = 14 Service discovery & heartbeats in Microservices

⇒ When the data pipelines are slow & we need to process the data, then we use service discovery.



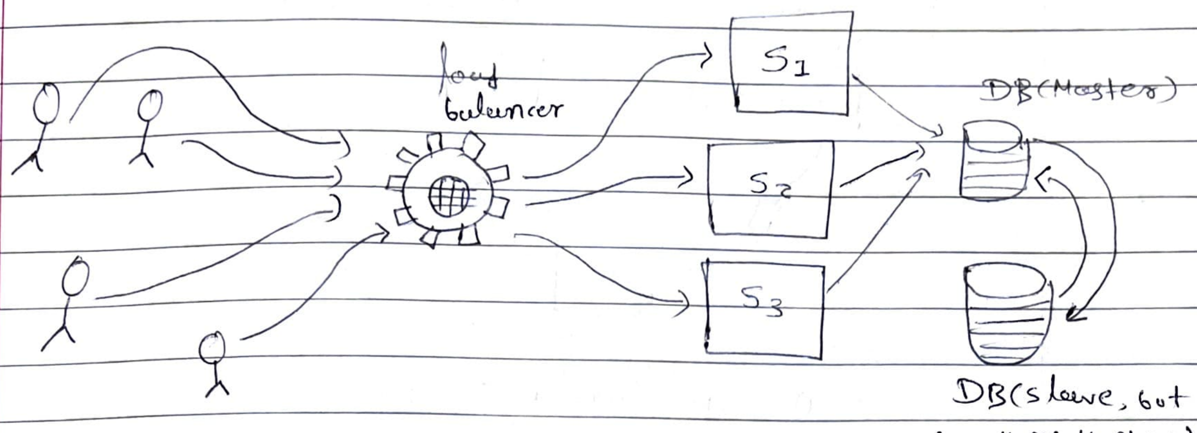
→ based on the snapshot health service communicate with particular service & check that is alive or not.

* health check is important because if you have a fleet of 10 server instance behind a LB, and one of them becomes unhealthy but still receives traffic, your service availability will drop to, at best, 90%.

* Methods for single point failure in database
  - sharding
  - Master-slave (distributed consensus)

Day = 15

* Distributed Consensus & Data Replication strategies on the server.



=> slave can copy data from master in two ways:-

① Synchronisely ② Asynchronisely.

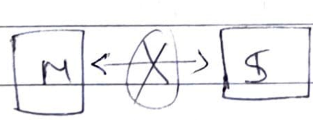
⇓ disa (under ①)
- taking time and load on master database

⇓ disa (under ②)
- because it's async so, master's transaction may not reflects instantly if it's fails.

=> Now, when master DB fails, we have slave DB to get the request from the server, & perform operations.

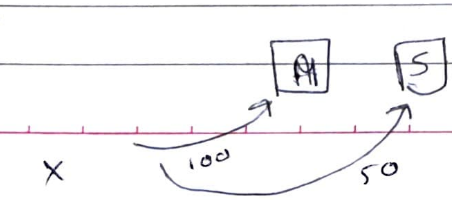
=> As slave work as a master, so, it also takes write operations, and updates to the original master.

=> This is going well, but the problem is that what about when the N/w or the connection between the master & slave fails.



=> they both get assume, and take control that I'm a master, and work as a master.

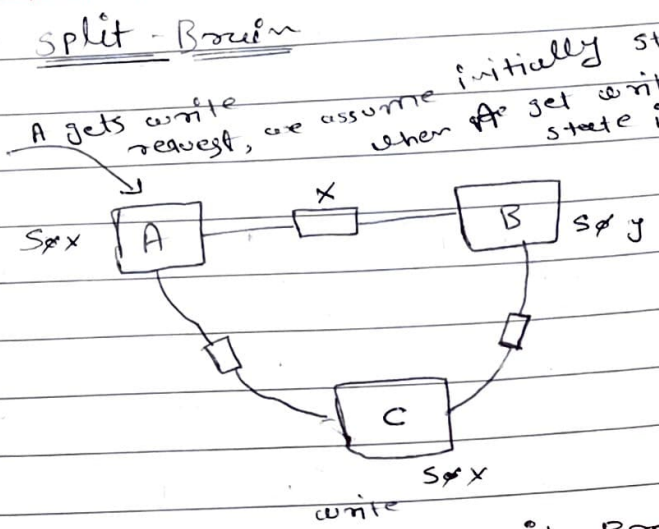
this is called split brain problem



x has total 120 balance,
first he request to deduct 100
120 − 100 = 20
second he request to deduct 50
120 − 100 − 50 => −30
Negative balance

Date / /

* Split - Brain



=> when A gets request, it propogate to C, so both's status is S_x.

=> when B gets write request, its status is S_y, any B tried to propogate request to the C, but it's not happen. because C ask to B about it's (B's) previous state, any it was S_0, so, C tells your status is not like me, so, you (B) need to update your state, then you can do the transaction.

=> So, now B rollback it's transaction from S_y to S_0, any sync up with S_x from C. So, new transaction of B is not happened.

-> Now, B can get new request, any perform the operation to changing it's state.

This is called "Distributed Consensus"

⇓

Multiple node agree on a particular value.

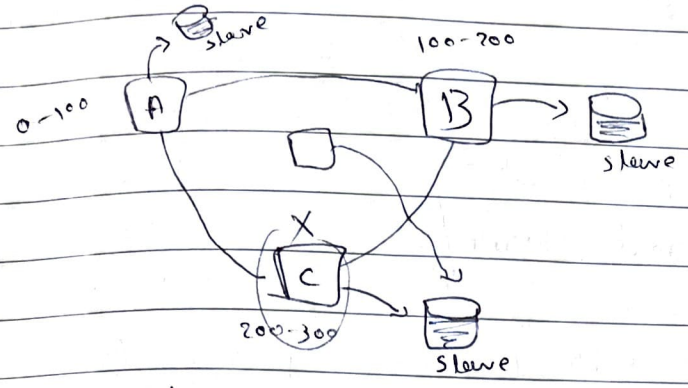
Ex:

-> 2PC, 3PC, MVCC, Saga

postgre

DOMS | Page No.
Date / /

=> we can also use sharding with Master-slave :-

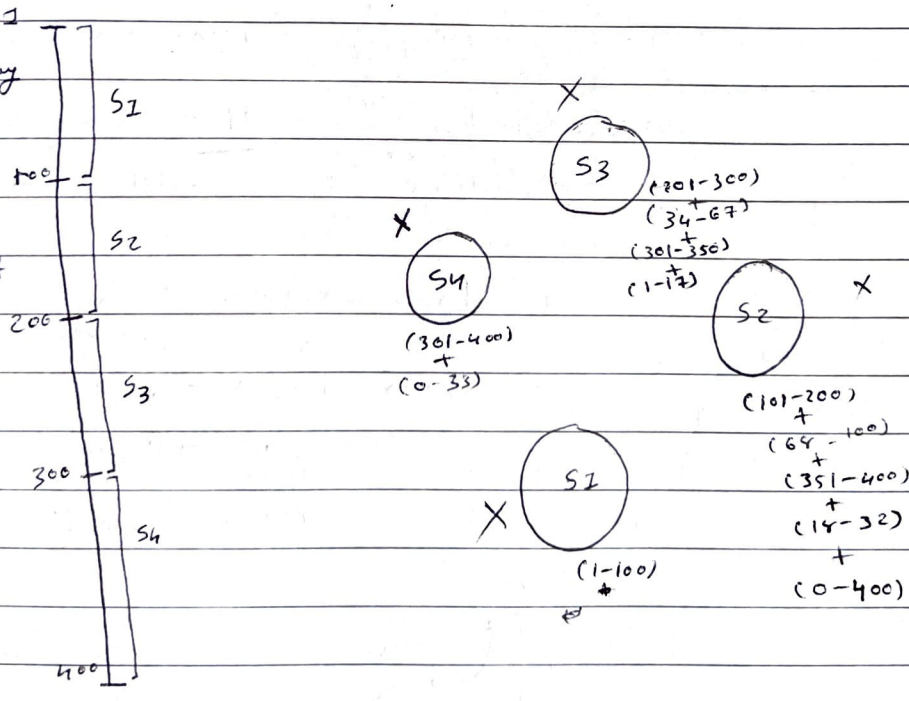


=> central co-ordinator redirect request to c's slave when c shard is fail.

Day-16

* How to avoid cascading failures in a distributed system.

(1) cascading failure
(2) viral / black friday
(3) Job scheduling (email ex.)
(4) popular post
(5) predictable load increase



=> when S1 crash, it's requests divide into S4, S3, S2
=> when S4 crash, it's request divide into S3, S2
=> " S3 ", all the requests handle by S2, coz it's a high chance that S2 will also die / crash.

this is called "cascading failure problem"

* Solution

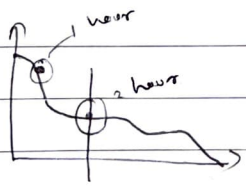
① Re-scale
② Auto-scaling
③ rate limiting
④ batch Processing
⑤ Approximate statistics
⑥ caching (Key-value pair)
⑦ Jenkins deployment
*⑧ coupling

Ex 1 if you want send mail to 1B users, then you sent by batch.

[10000] - 1 min
[10,000] - 2 min
||
||
||
[10,000] - 10,000 min.

Ex 2 Youtube use Jitter for shows the views count on Youtube videos

1 hour → 1000 views likes comments
2 hour → 1000 × 1.5
1500 → 1700
↓ Approx. ↓ Reality.



DOMS | Page No.
Date / /

Capacity planning (rules & tips)

→ capacity estimation → no. of transactions
→ Amount of data to be stored.
→ Availability

→ Metric system → Millions, billions, trillions

1 M = 10^6
1 B = 10^9
1 T = 10^{12}

→ Storage capacity

1 B = 8 bits
1 KB = 1024 B
1 MB = 1024 KB
1 GB = 1024 MB

⇒ Memorizing Exercise

- 1 Million / day = 12 / sec, 700 / Min, 4200 / hour.
- 10 " / day = 120 / sec

→ Rounding off

Power of 10's

~~5460~~
1000

~~960~~
1000

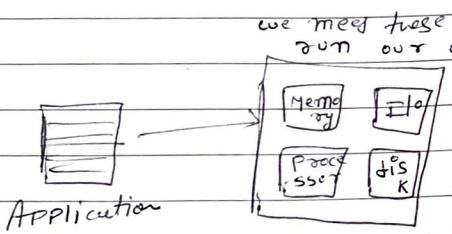
Power of 2's

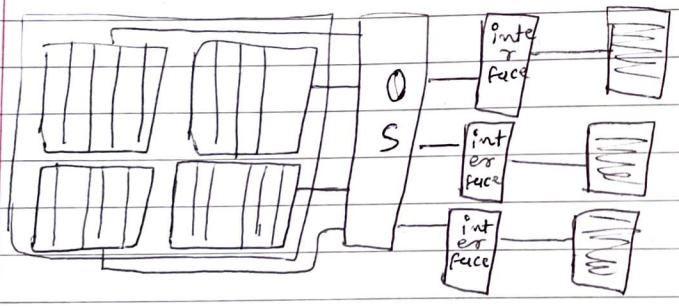
256
~~254~~
256
~~250~~
64
~~64~~

⇒ Latency numbers every programmer should know

⇒ Capacity estimation of what?
- no. of reads & writes
- no. of transactions
- huge data transfer
- N/w bandwidth
- no. of requests a server has to serve
- no. of servers needed
- how a LB can handle these many requests

* Container & visualization in cloud computing





⇒ we have multiple programs which are going to be taking up slices of the resources that OS can provide. and the remaining is going to be unused.

⇒ The problem is that when the memory runs out with one program, then every one else has the issue to perform their operation.

↓ solution

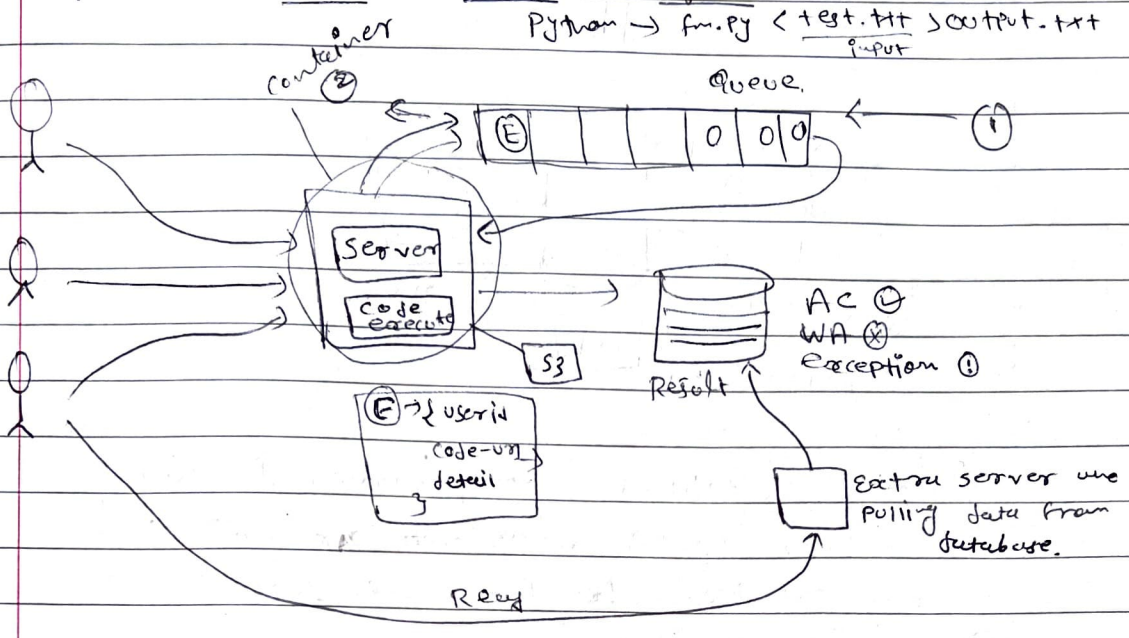
① VM ② container

→ database ⇒ NoSQL, SQL (dynamoDB, cassandra, etc.)
⇒ heartbeats ⇒ zookeeper
⇒ LB ⇒ ELB (elastic load balancer)
→ Message Queue ⇒ Rabit MQ, Kafka
→ CDN ⇒ Akamai, open connects

DOMS | Page No.
Date / /

* to write the data any ready test like tiktok videos, we need to use Amazon S3.

* Remote code execution engine

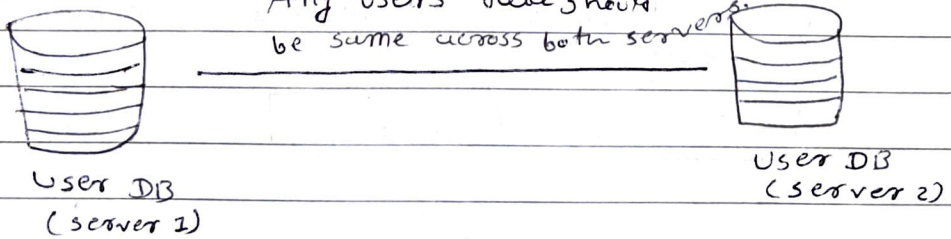


we need only two thing
1. Job scheduler
2. container

* Data consistency & tradeoffs in distributed system

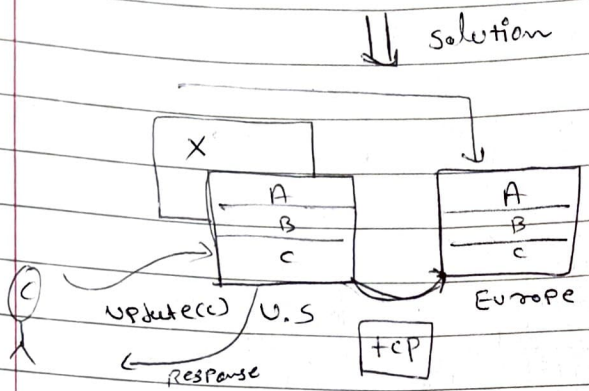
① what is consistency?

if we have two server then,
Any user's data should be same across both servers.



DOMS
Date / /
Page No.

* Issues of single server?
  - single point of failure
  - cost of vertical scaling is high (impossible)
  - high latency



⇒ copy of data in every server of every region. so, when us server fails, user can get the data from the Europe's server.

⇒ if user c want to update, there is some Mechanism between all the servers to update c in their server.

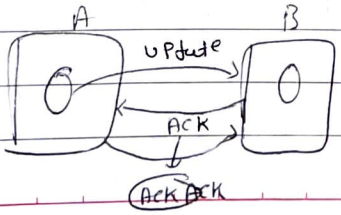
⇒ data available everywhere, so latency will low.

⇒ we use tcp protocol for updating data from 1 server to other.

C (c → c++) → US to Europe

* what happens if the message transmission fails?

- Two general problem

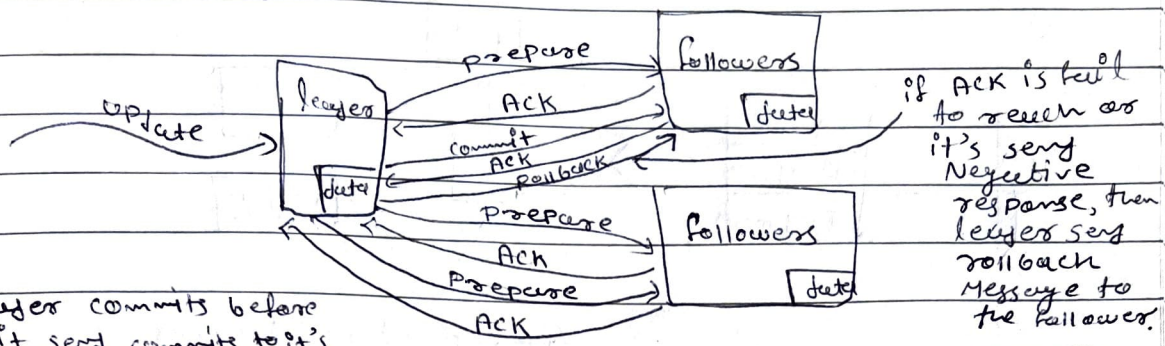


" Eventual consistency is better than absolute consistency "

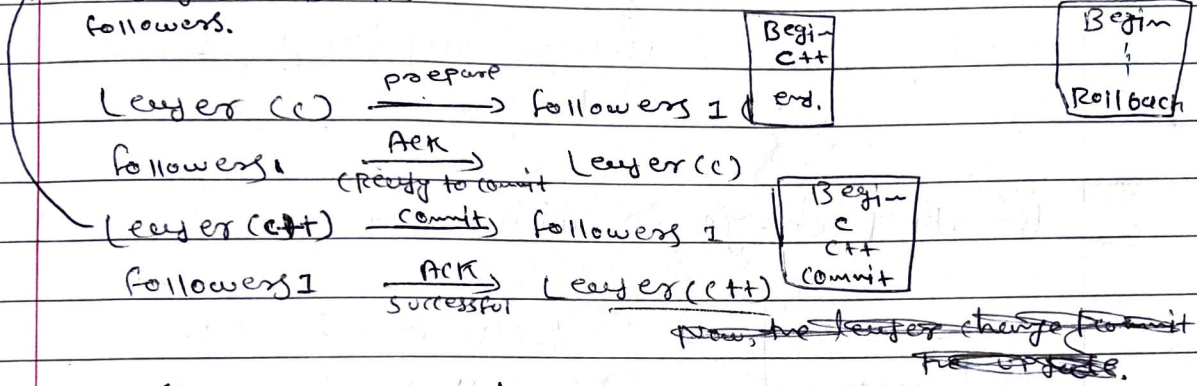
DOMS | Page No.
Date / /

=> Solution to the consistent data in distributed System?

*2 Phase commit. (Absolute consistency)
- 1 Phase (prepare)
- Retry (commits)

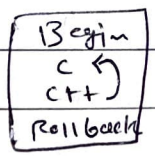


leader commits before it send commits to it's followers.



~~now, the leader change for commit the update.~~

=> if the rollback happens in Followers 1



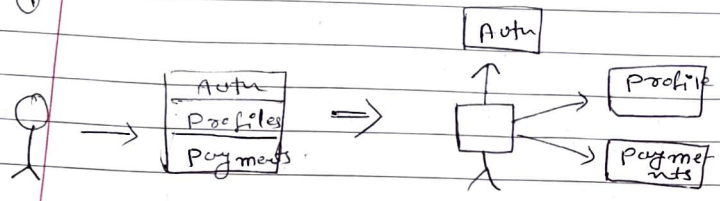
then it should ACK Rollback statement to leader & leader also rollback. So, both leader & Followers 1 has get (C).

Only problem is that, when L send commit, but it doesn't go through to F1, then we block that row in L, F1, F2 (all server's data), why it's not availability, so then User tried to access F, it will not available due to block.

Monoliths to Microservices

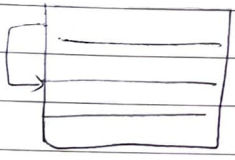
⇒ small team ⇒ Monolith
⇒ large team ⇒ Microservices

①



1. Separation of codes
2. Engineering / coding is easier
3. deployment is easy

② Breaking change in microservices



get Profile (int id)
↓↓
Only one function easy to handle changes from Module A to B)

1. duplication is lesser
2. small teams
3. contracts are in 1 place.

Micro service

getProfile (int id)

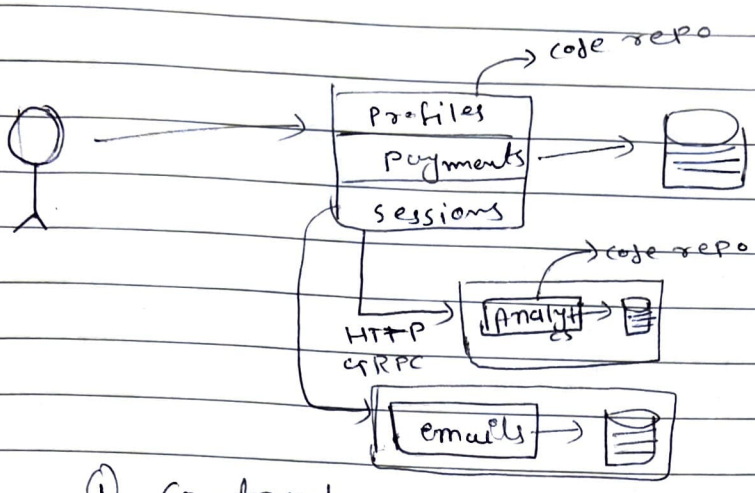
↓

getProfile (int id)
(string id)

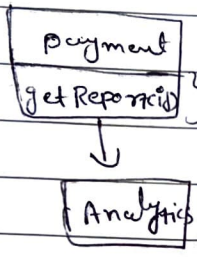
breaking change

DOMS | Page No.
Date / /

* Steps for Migration



① Contract



→ issue happen when the library is not updated.

② Router (LB or service Registry) ← Just like dockerhub)

③ simplify Deployment (Autodeployments)

① scp repo - dest cloud	① CI/CD pipelines
② ssh box	② Containers
③ run repo	③ Service Deployments

Ex Jenkins

④ Communication between services can be varied.

⑤ Logging