(1) Back of the Envelop - Estimation →

(Estimates) + (Thought Experiment) + (Common performance Numbers)

→ which design will meet your requirements.

→ Good sense of scalability Basics.

→ Following concepts should be aware →

- power of (2)
- Latency Numbers
- Availability Numbers

Power	Approximate Value	Full Name	Short Name
10	1 Thousand	1 Kilobyte	1 KB
20	1 Million	1 Megabyte	1 MB
30	1 Billion	1 Gigabyte	1 GB
40	1 Trillion	1 Terabyte	1 TB
50	1 Quadrillion	1 Petabyte	1 PB

→ Latency Numbers →

Operation Name	Time
L_1 Cache Reference (Level 1 Cache Reference)	0.5 ns
L_2 Cache Reference (Level 2 Cache Reference)	7 ns
Main Memory Reference	100 ns
Compress 1K bytes with Zippy	10 000 ns = 10 µs
Send 2K bytes over 1 GBPS Network	20 000 ns = 20 µs
Read 1MB sequentially from memory	250 000 ns = 250 µs
Round trip within the same datacenter	500 000 ns = 500 µs
Disk Seek	10 000 000 ns = 10 ms
Read 1 MB sequentially from the NW	10 000 000 ns = 10 ms
Read 1 MB sequentially from disk.	30 000 000 ns = 30 ms
Send packet CA (California) → Netherlands → CA	150 000 000 ns = 150 ms

After seeing latency numbers →

→ Memory is faster than disk. (RAM fast होता है Disk से)

→ Avoid disk seek if possible (Cache का use करो) (Memory (Ram) का use करो)

→ Simple compression Algorithm are fast (Compress कर दो)

→ compress data before sending it if possible

→ Data centers are far located (Time taking)

Availability Number →

→ High availability = System continuously operational for long.

→ Measured in % ⇒

100% availability = 0 down time

Mostly service = 99% to 100%

→ SLA (Service level Agreement)

→ Agreement b/w service provider and consumer / customer.

→ ~~Dis~~ available (uptime)

eg → Amazon Google Microsoft

Cloud service = 99.9% ↑

Availability	Downtime per day	Downtime per year
99 %	14.40 minute	3.65 day
99.9 %	1.44 minute	8.77 hours
99.99 %	8.64 seconds	52.60 minutes
99.999 %	864.00 millisecond	5.26 minutes
99.9999 %	86.40 millisecond	31.56 seconds

---

(2) Twitter QPS and Storage Estimation →

Q → Estimate Twitter's QPS (Query per second) and storage Requirement

Note → Following numbers are taken e.g. → only

**** Assumptions → (V V V Imp)

- Monthly active user = 300 million
- Daily users = 50 %
- Tweets per day on Avg per user = 2
- Tweets containing media = 10 %
- Tweets stores for = 5 years

Estimations →

Query per second (QPS) estimate →

→ Daily Active Users (DAU) = 50% of (300 million) = 150 million.

→ Tweets QPS = 150 million * 2 Tweets / 24 hours / 3600 seconds

≈ 3500 Tweets / sec

→ Peak QPS (100% User) = (2 * QPS) ≈ 7000 Tweets / sec

Storage Requirement Estimation →

Note → we will only estimate media storage here

→ Average tweet size →

→ tweet_id = 64 bytes
→ text = 140 bytes
→ media = 1MB

→ Media storage = (Per day active users) * (no of tweets per user) * (10% contain media) * (size of media)
= (150 million) · (2) · (10%) · (1MB) = $(30 \times 10^6) = (30\ TB)$ per day.

∵ $1\ TB = 10^6\ MB$

5 year media storage = (30TB) * (365) * (5) = (54750 TB) ≈ 55 PB

∵ 1 PB = 1000 TB

→ Interview Tips →

→ Rounding & Approximation → 99989 / 9.2 ✗
≈ 10000 / 10 ✓

→ Always write down your Assumption

→ Label your units → size 5 ✗
size 5 MB ✓

→ Commonly asked question → QPS, Peak QPS, Storage, Cache, number of servers etc.

---

③ Red Flags and Myths →

Dream Company → selected for interview
- DS
- Algo
- Space & time Complexity
- System Design.

→ Intimidating after
→ Design a "well known product "ABC" "
→ How ? 1 hour ? → Man, it was built by (100 - 1000)s of engineers.

Good News →

No one expects you to design a perfect real world system in 1 hour

e.g:- Real world system design is complicated

Google Search :- Deceptively simple But Amount of technology under it is astonishing

If no one expects us to design in an hour, then what is the purpose of System design Interview ?!

→ Simulated real life problem solving
→ Two co-workers (you & Interviewer) collaborate on an ambiguous problem and come up with sol^n.
→ Open ended problem
→ No perfect answer
→ Process and effort > final result.
→ Demonstrate your design skill, Defend your design choices, Respond to feedbacks.
→ They want to accurately assess your ability.

Myth →

"System Design Interview is all about your technical design skills" ⊗

It gives strong signals about your ability to →
→ Collaborate → you can collaborate in a very good manner
→ Work under pressure
→ Resolve ambiguity
→ Ask good questions.

Red flags →

↳ over Engineering is a disease
↳ Too much design purity
↳ Ignoring tradeoffs
2↳ Ignoring compounding costs of over engineered systems.
↳ others: Narrow Mindedness stubbornness etc.

---

④ Design News Feed System (Step 1 of 4) →

4 step process for effective System Design Interviews →

Step (1) Understand the problem & Establish Design Scope
→ Ask good question (most imp skill)
→ Dont jump to answers without clear understanding of requirements.
→ Clarify requirements and assumptions
→ What kind of question to ask ??
↳ What specific features to build? (कौन से specific features होंगे)
↳ How many users do the product have?
↳ How fast company anticipate to scale up? (कितने user बढ़ेंगे)
↳ what is company technology stack?

✱ News Feed System →

Q → Is this the mobile app? web app? or both?

A → Both (Mobile App + Web App)

Q → What are the most important features of the product?

A → Ability to post
→ Ability to see friend's news feed.

Q → News feed sorted in any particular order? first closefriends 's feed then group's feed?

A → Reverse chronological order (to keep things simple) ①②③

Q → How many friends can a user have?

A → 5000

Q → what is the traffic volume?

A → 10 millions daily active user (DAU)

Q → Can feed contain image videos OR just text?

A → It can contain media file (Both images & videos)

याद करो above questions, twitter estimation में हमने देखा था

So similar pattern के question you can ask.

---

(5) Design News Feed System (step 2 of 4) →

(step 2) Propose High level Design and Get Agreement →

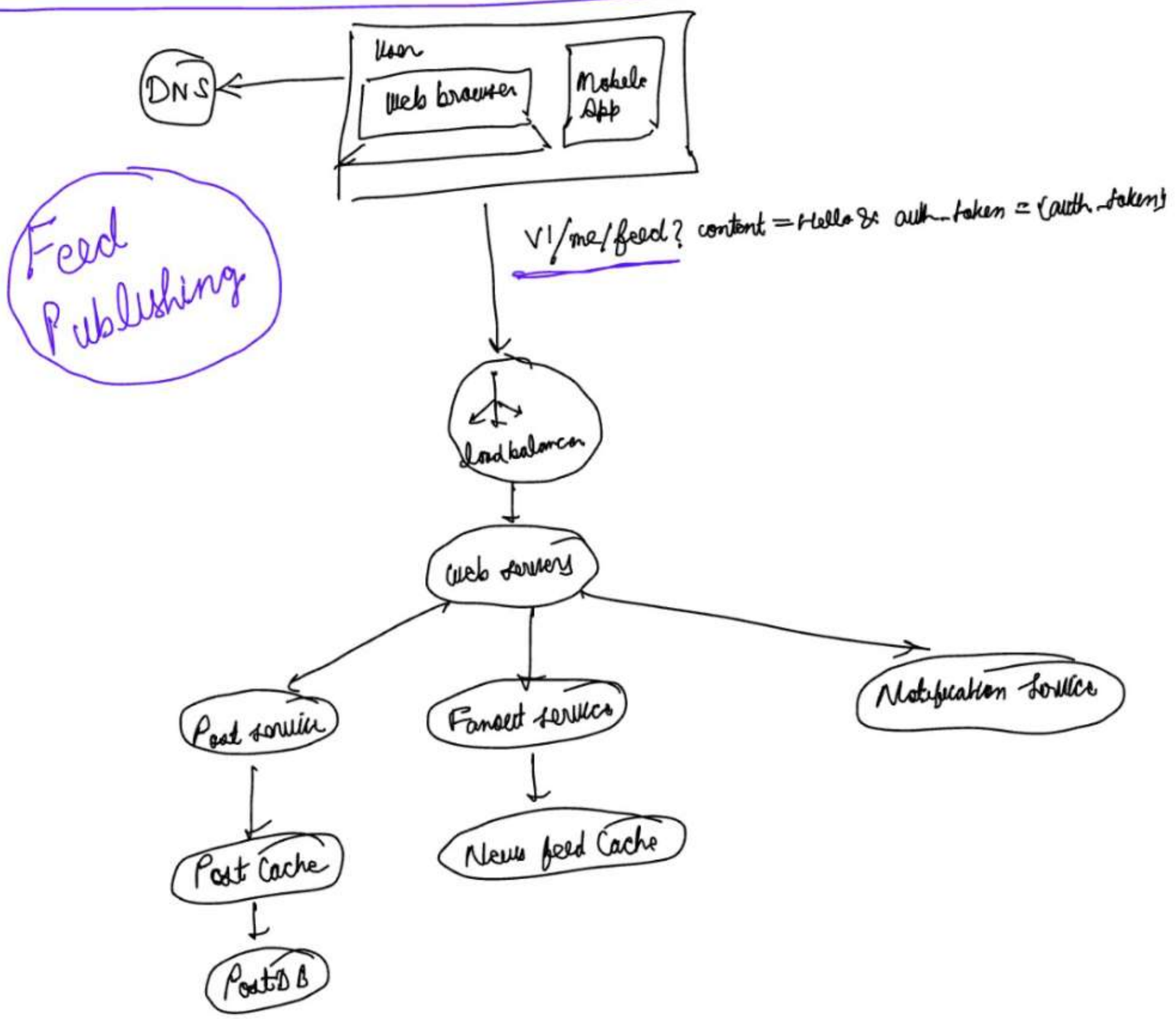
→ Make High level design Reach an agreement with interviewer.
→ Blue Print → Ask
→ Box Diagrams with Key Components.
  ↳ Clients (web/mobile)
  ↳ API
  ↳ Web Servers
  ↳ DataBase Cache
  ↳ CDN
  ↳ Message Queue etc.
→ Ask if Back of the envelope is required.
  ↳ if yes → Then do the calculations to evaluate if your blueprint fits the scale constraints.

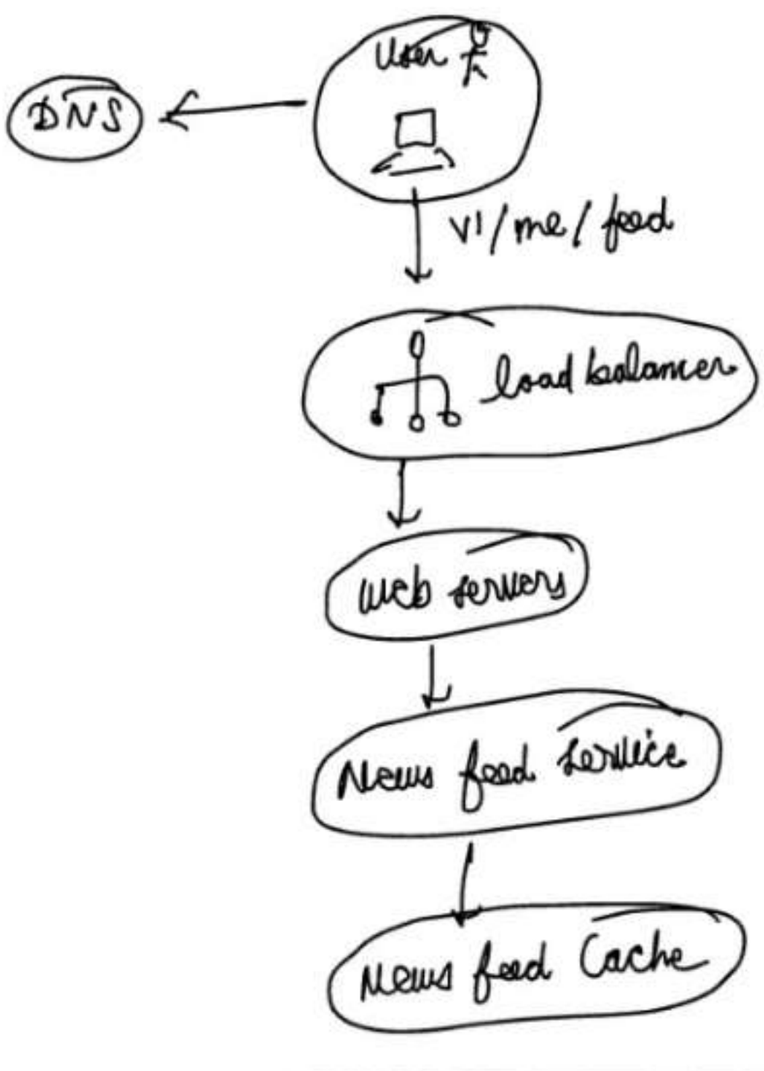
✱ News Feed System (High Level Design) →

Note → Few Components in design, I will cover later bcz they are long topic and need to be cleared properly in seperate lecture.

High level 'News Feed System' will have 2 flows
  ↳ Feed Publishing → Published post → DB/cache
                    → Friend's News feed.

↳ News Feed Building : Aggregating friend's post in "reverse chronological order".





6 Design News Feed System : (step 3 of 4) →

(step 3) Design Deep Dive →

By this time following should have already been achieved.

- ↳ overall goals and scope — Agreed.
- ↳ High level blue print.
- ↳ Feedback on High level design.
- ↳ what area to focus on — Based on Interview feedback

→ Identify & prioritize components → Interview Points

→ May focus on → High level design (HLD)

→ Back of the envelope estimation

Most of the time → Any particular component

e.g → ⓐ URL Shortner — Hashing is important

ⓑ Chat system — Reduce latency support (online / offline) status etc.

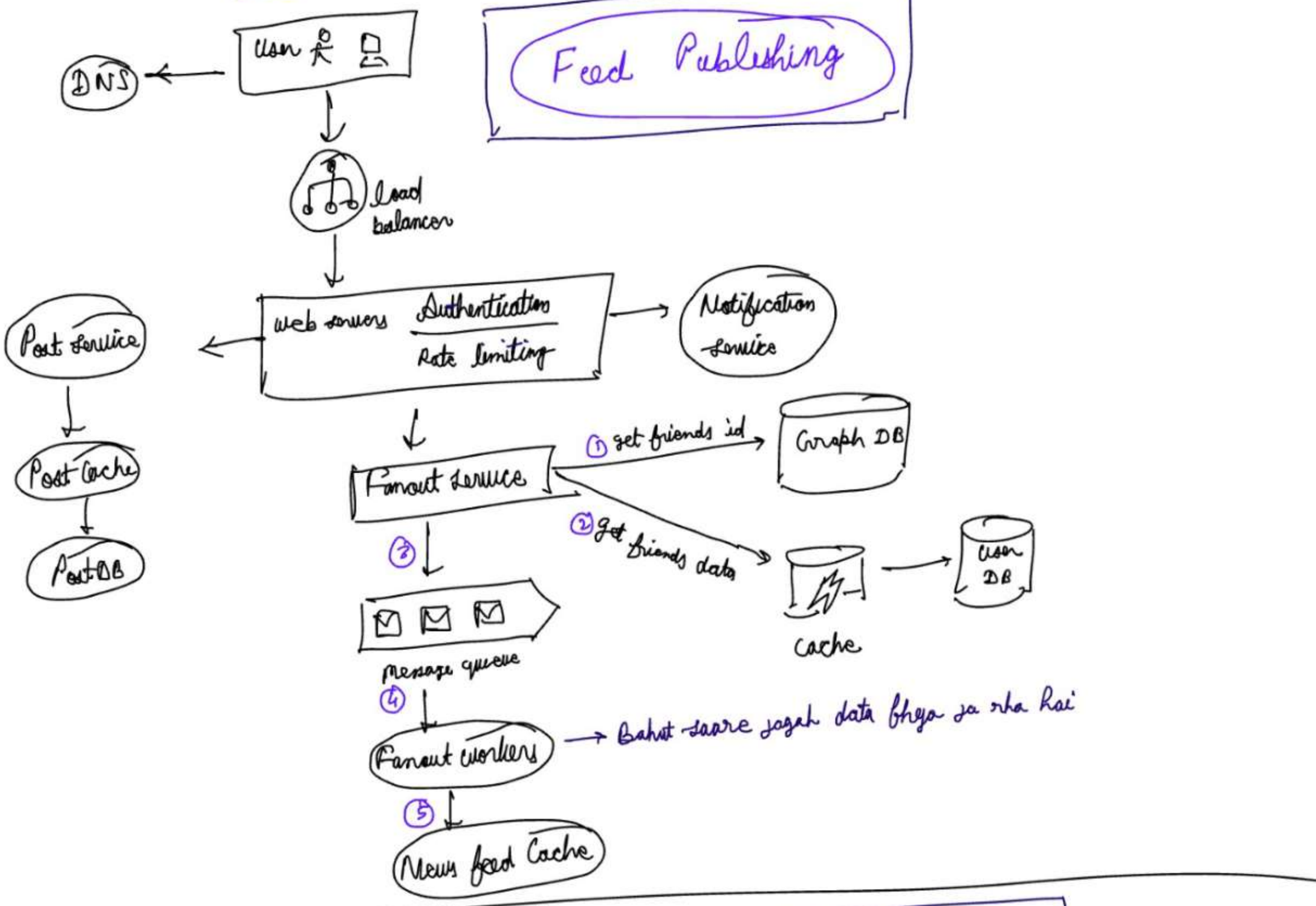
→ Dont waste time in un-necessary component details

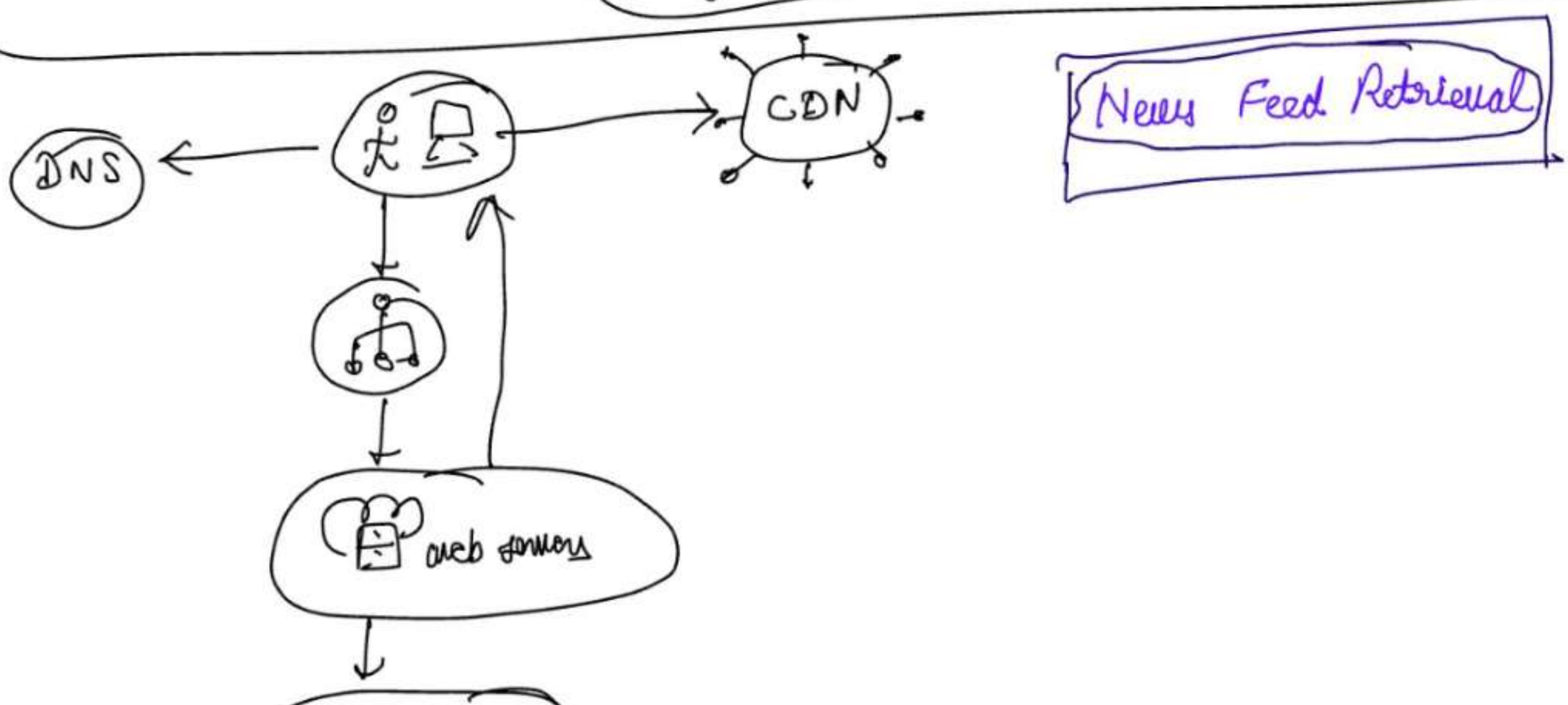
✱ News feed system (Design Deep Dive) →

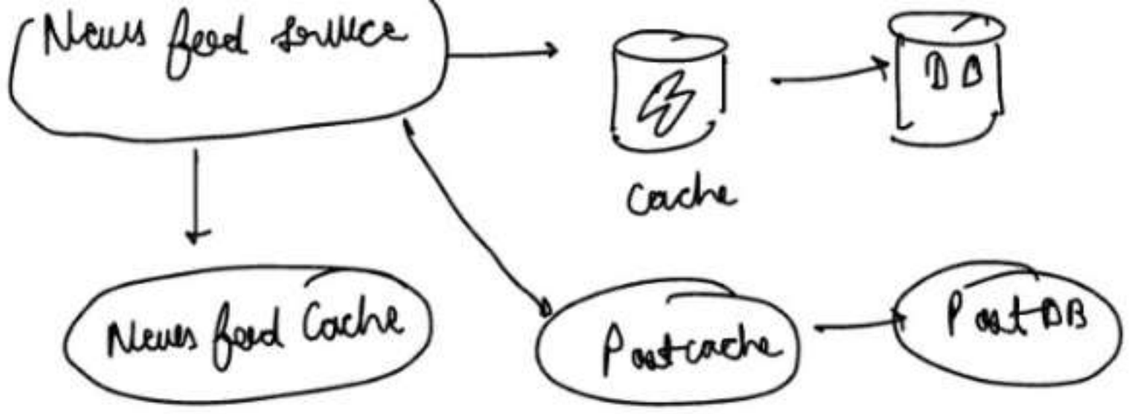
2 imp use case

↳ Feed Publishing

↳ News feed Retrieved.







(7) Design News Feed System (step 4 of 4) →

Step 4 Wrap up.

→ Interviewer might ask Follow-Up question.
- ↳ Improvements / system bottlenecks
  - ↳ Never say your design is perfect.
- ↳ Recap of your design.
- ↳ Error Cases (Network loss / server failure) etc.
- ↳ How do you monitor metrics / logs?
- ↳ Next Scale Curve. 1 million to 10 millions users

Dos	Dont
→ Ask for clarification	→ बिना Basic पूछे मत जाओ
→ Understand requirements	→ Dont directly jump to sol^n
→ Communicate - talk out your thoughts	→ Too much detail on un-necessary
→ multiple Approaches (if possible) (else not)	→ Not asking for hints.
→ Agreement → Detail each component → most critical one first.	→ Not asking feedbacks early and often.

* Time Allocation on Each step →

1. Understand problem + Establish Design scope (3–10 min)
2. Propose high level design + Agreement (10–15 min)
3. Design Deep Dive (10–25)
4. Wrap (3–5 min)

(8) Design Rate Limiter →

In Network System →

Rate Limiter → Used to Control the rate of traffic sent by client OR service.

In HTTP world →

Rate Limiter → Limits the no of client requests allowed to be sent over a specified period.

```
if count(API request) > Threshold
{
    "Access Call Blocks"
}
```

Examples →

→ No more than 2 post / second

→ Maximum 10 accounts / day from the same IP address

Before designing, see the benefits of Rate Limiter

→ Prevent DoS Attack →

→ Every org. uses Rate limiting. (Bot API call नाहो)

→ Example → Twitter → no of tweets = (300 per 3 hours).

→ Blocking access intentional OR Unintentional requests.

→ Reduce Cost →

→ limiting excess request = (Few Services) + (More resources for high priority APIs)

→ Important if we use 3rd party APIs → ~~पैसे लगते हैं~~

→ Prevent Server Overloading →

→ Filter out excess requests

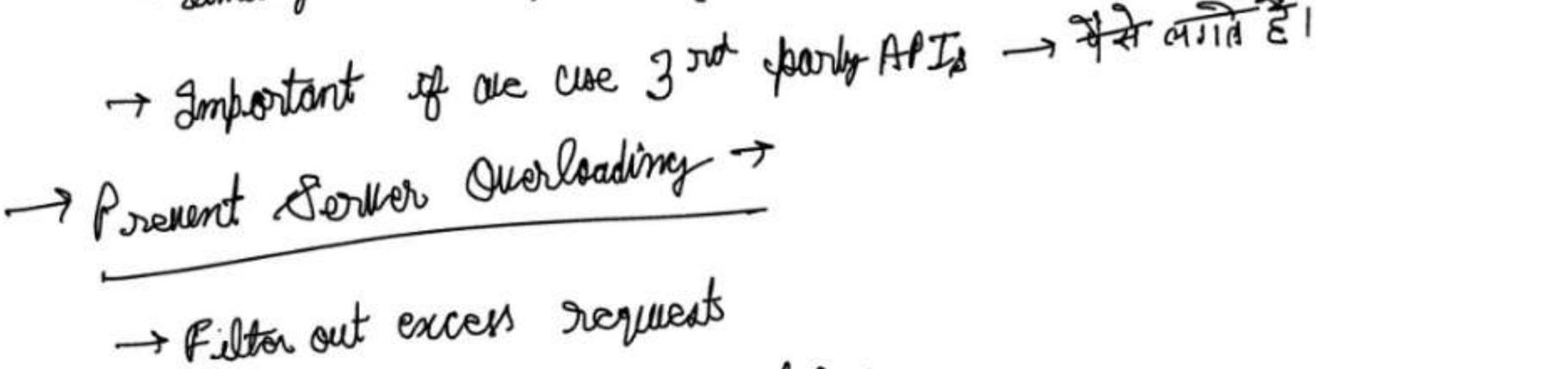
→ Caused by Bots OR users misbehavior

---

(9) Design Rate Limiter (Step 1 of 4) →

Step (1) Understand the problem and Establish Design Scope →

Note →



Q → What kind of Rate limiter ?
  ↳ Client side rate limiter
  ↳ Server side rate limiter → Mostly यही use होता है

A → Server side API rate limiter

Q → Limit API requests based on
  ↳ IP? (same IP)
  ↳ UserID ? (same userID)
  ↳ other properties ?

A → Flexible enough to support different sets of limiting rules.

Q → What is scale of system ?
  ↳ start up
  ↳ Big Company (large user base)

A → Able to handle large no of request.

Q → Will system work in distributed system ?

A → Yes → बड़ी company distributed environment में ही काम करती है।

Q → Rate limiter separate service होगी ??? या फिर should it implement in application code ?

A → Its a design decision, (up to you).

Q → Do we need to inform users who are throttled (limited) ? (जिनके लिये limit किया है user को पता होना चाहिये)

A → YES

✡ Requirements →

→ Limit excess requests

→ Large User Base
  ↳ should not slow down HTTP response time. (slow नही होना चाहिये)
  ↳ Use as little memory as possible. (Rate limiter के लिये कम memory use होना चाहिये)
  ↳ High fault tolerance. (Back up जैसा कुछ होना चाहिये)
    ↳ If any issue (Cache server goes offline, server down etc).
    ↳ Must not affect entire system.

→ Exception Handling → Show clear exception to user (user को inform करना है कि उसकी request block हो गयी है)

→ Distributed rate limiting → Shared across multiple servers. (Sharable across servers)

---

(10) Design Rate Limiter (Step 2/4) → (Part 1) → where to put Rate limiter →

~~Step~~ (2) Propose High level design & Get Buy in →

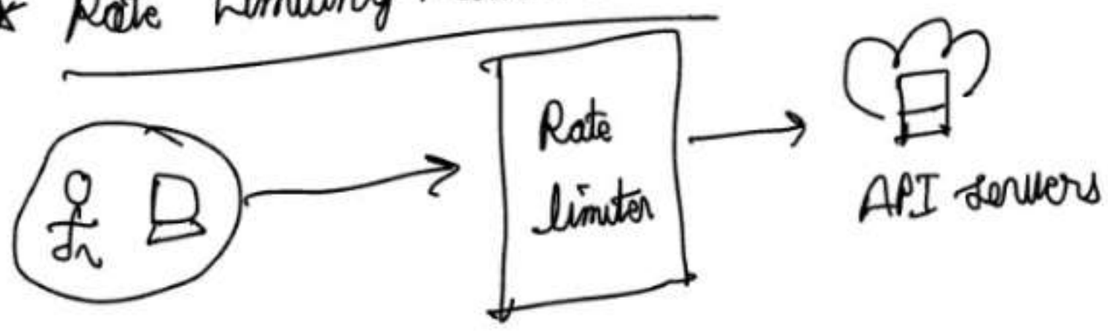
Note → → Where to put Rate limiter ?
  → Rate limiter algorithm
  → High level Architecture

Where to put Rate Limiter ?

↳ Client Side → We generally have no control on client side. (Client side mein nhi rkhenge)

↳ Server-Side →

✱ Rate Limiting Middleware →

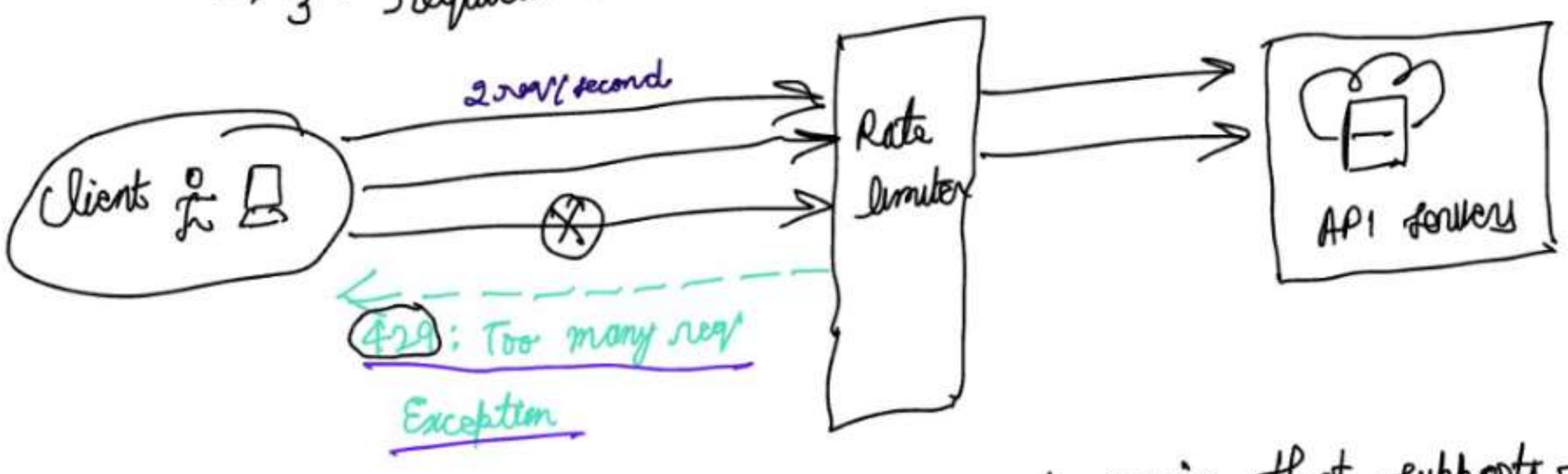


Example →

Assume → API allows 2 request / second.

→ Client sends 3 request within a second

→ 3rd request throttled with error: HTTP Status code 429.



✱ API Gateway (Middleware) → Fully managed service that supports →

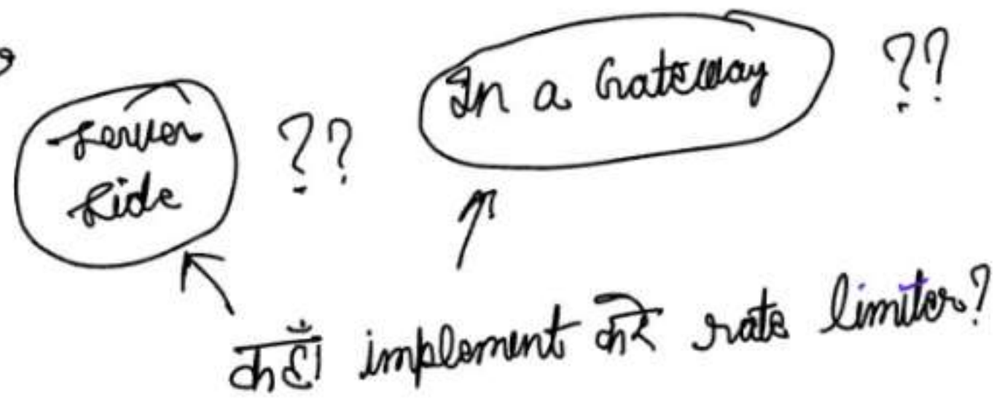
↳ Rate limiting

↳ IP whitelisting

↳ Authentication

↳ Static Content etc.

So,



कहाँ implement करें rate limiter?

Chill !!! It depends Bro →

↳ Current tech stack in your service.

↳ Programming language is efficient?

↳ Rate limiting algorithm that suits your Business. (server side / OR / gateway side ?)

↳ Already API Gateway use हो रही है then implement in it.

↳ Use Commercial API Gateway to save time & resource ?

(11) Design Rate limiter (2/4) part 2 → Token Bucket Algorithm →

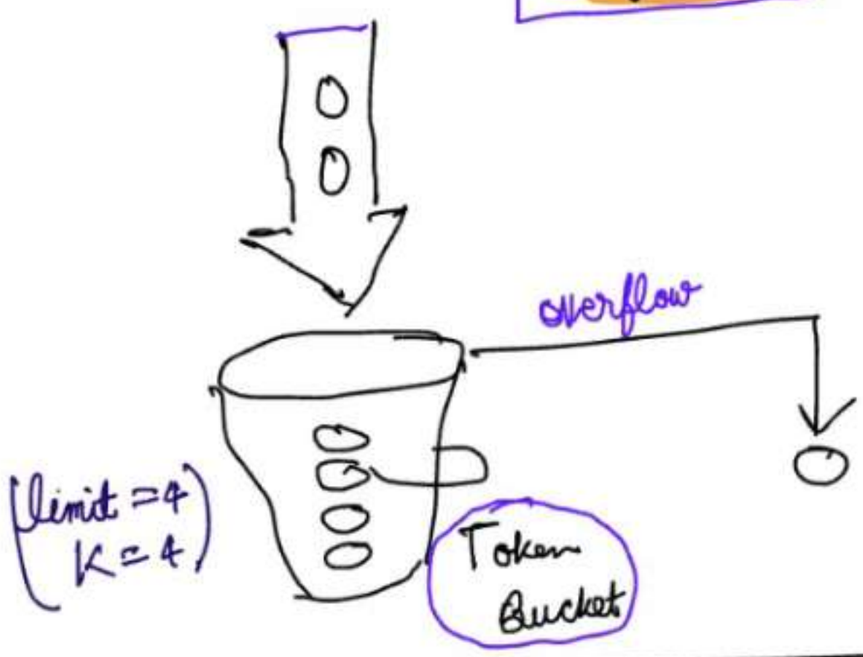
Step 2 : part 2 → Proposed HLD & get Buy in →

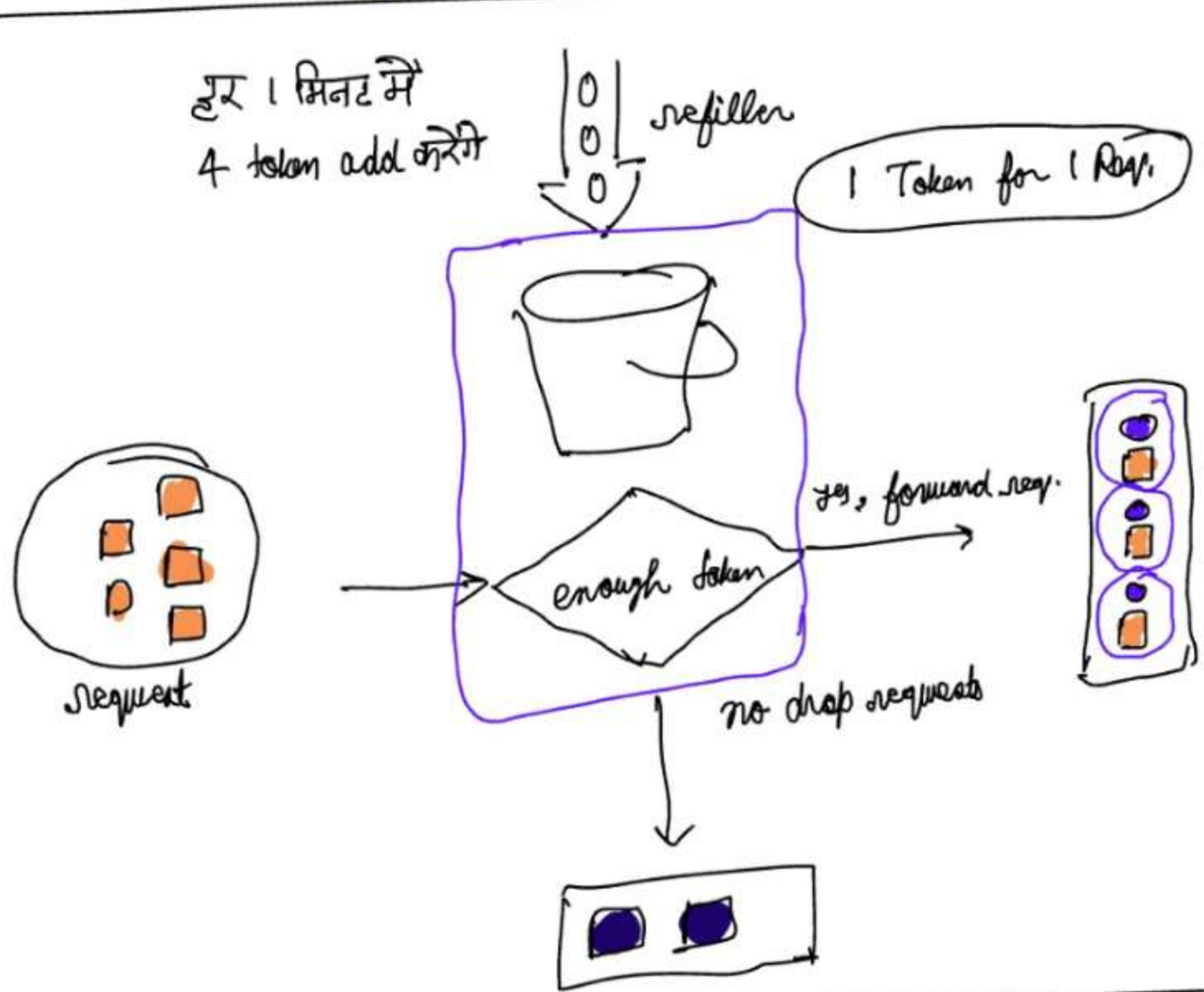
Note →

→ where to put Rate limiter ?
→ Rate limiting Algorithms ?
→ High level Architechture ?

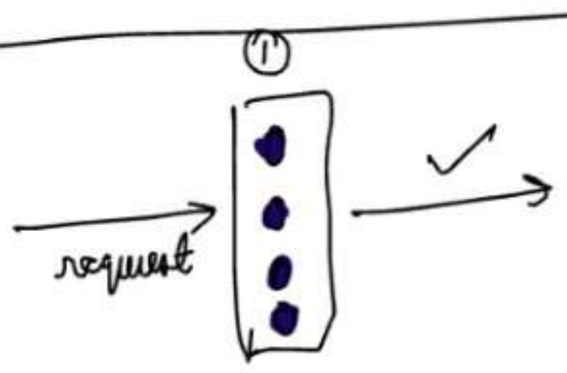
(1) Token Bucket Algorithm →

→ Widely used & Simple
→ Used by companies like Amazon Stripe etc.
→ Takes two params "bucket size" "Refill rate"



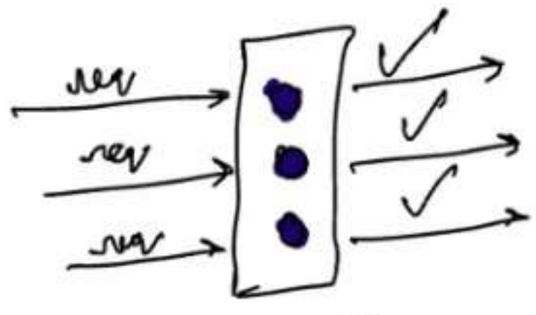


1 token ■ = 1 request



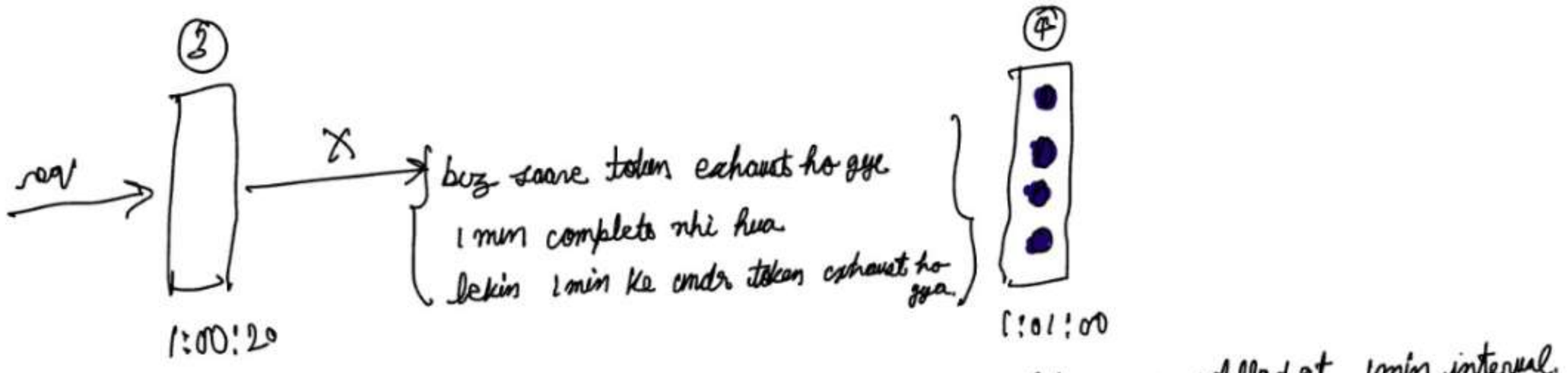
1:00:00

→ start with 4 token
→ The request will go through
→ 1 token is consumed.



1:00:05

→ start with 3 token
→ All three requests will go through
→ 3 tokens are consumed



→ Start with 0 token

→ The req will be dropped.

→ 4 token are refilled at 1 min interval.

---

* How many Buckets ?

→ Depends on rate-limiting rules

Example → Different API ends points : Different Buckets

If you have 3 API's, use 3 Buckets.

→ Throttle Based on IP addresses :- 1 Bucket / IP address.

→ System allows max 10000 req/s / second → A 'Globle Bucket' shared by all requests.

Pros :-

→ Simple

→ A req. can go as long as we have tokens.

Cons :-

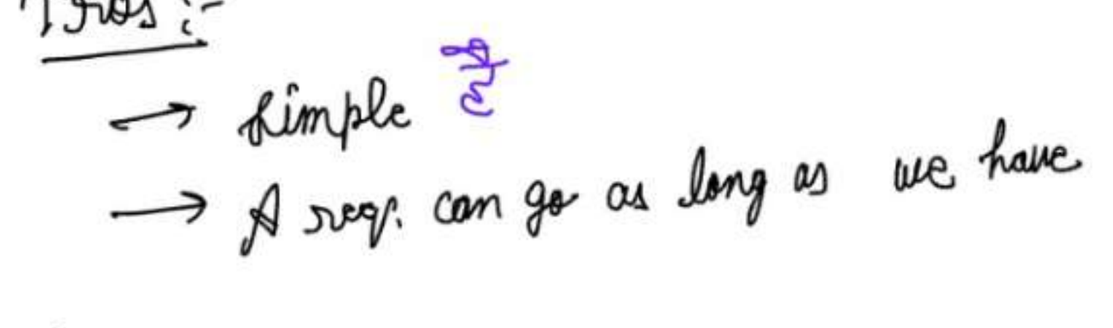
→ Bucket size and Token Fill rate timing is challenging → Take 2 parameter (Bucket size, Token Fill)

→ Sends packet at an average rate. (Rate बहुत खास नहीं होता)

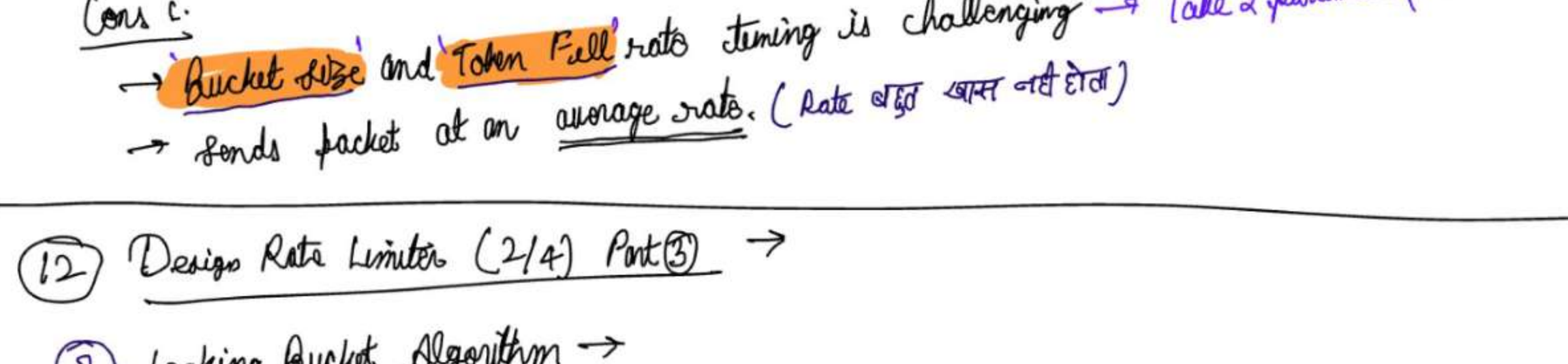
---

(12) Design Rate Limiter (2/4) Part (3) →

(2) Leaking Bucket Algorithm →



→ similar to Token Bucket but, requests processed at fixed rate.



→ Using FIFO queue (First in First out)

Takes 2 parameter →

↳ Bucket size → Equal to Queue size

↳ Outflow rate → No of request at fixed rate.

* Shopify (e-commerce) → uses leaky bucket for rate limiting

Pros :- → Memory efficient. Limited Queue size.
→ Stable outflow rate.

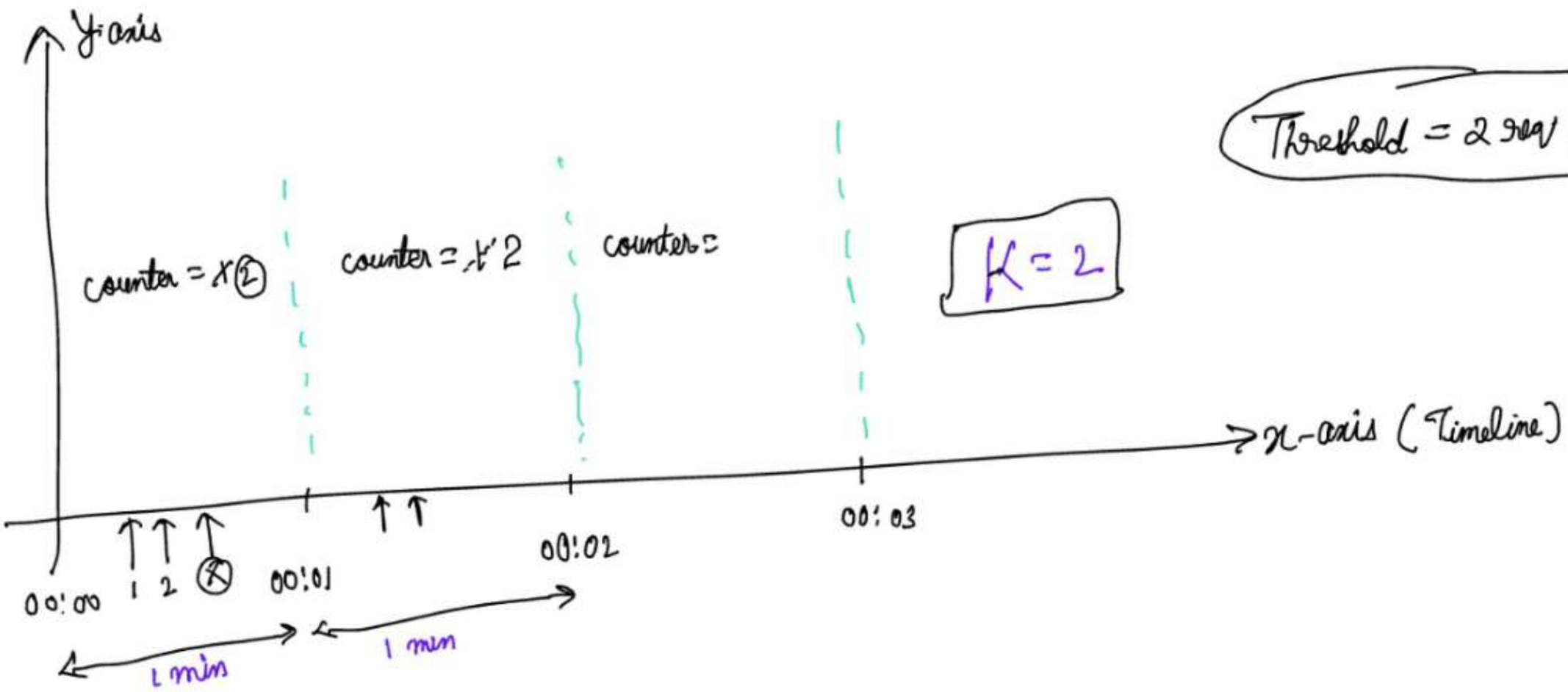
Cons :- → Takes 2 parameter. Tuning is challenging (Bucket size, outflow rate)

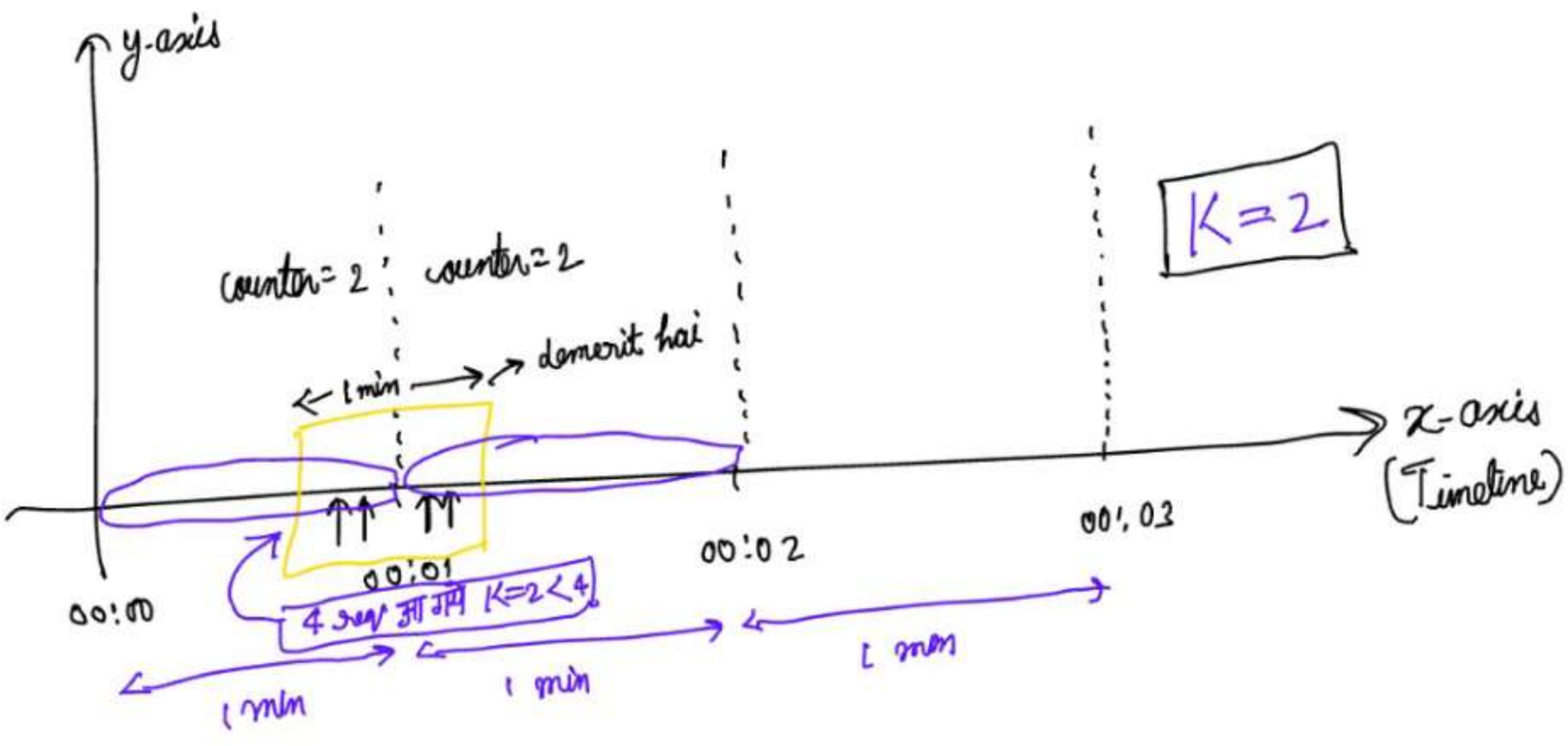
---

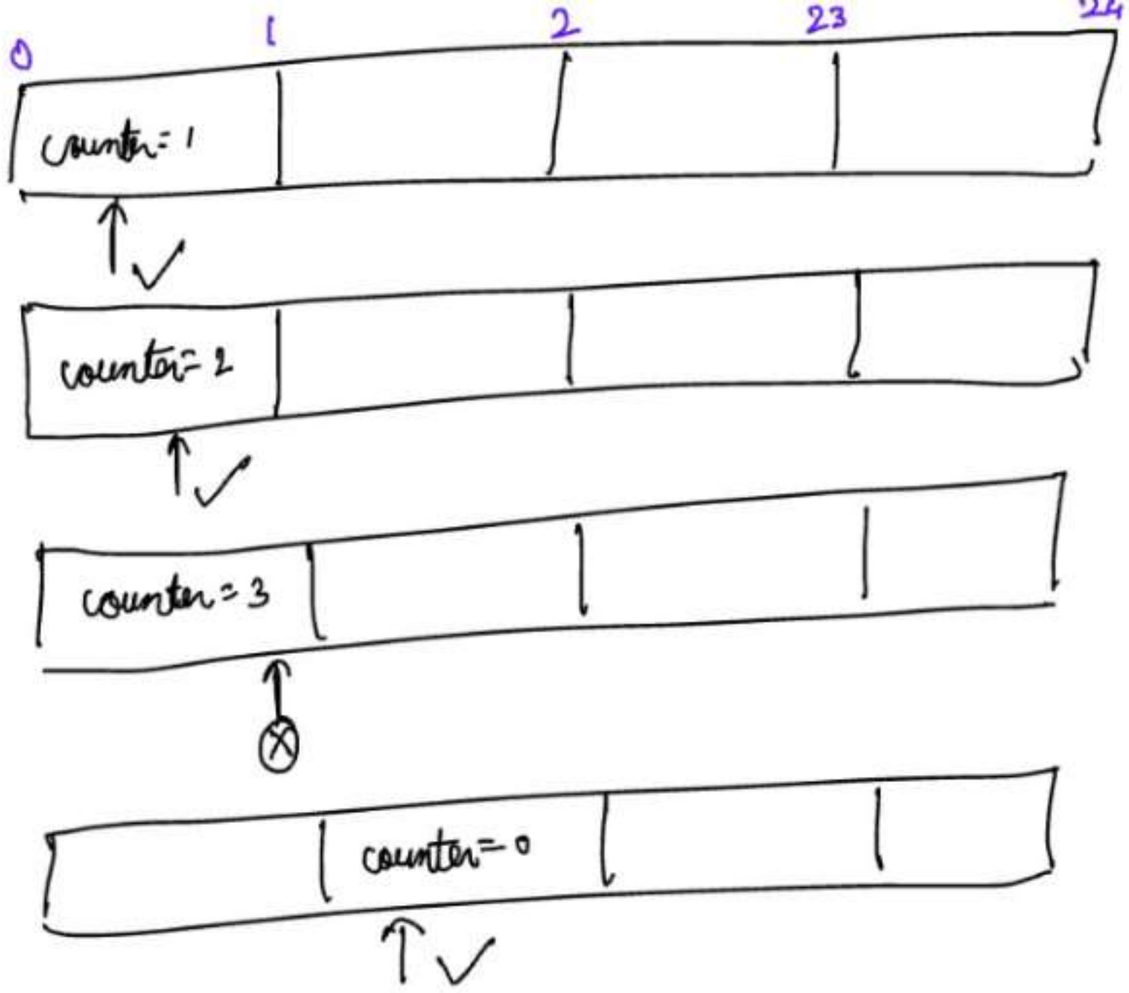
(13) Design Rate Limiter (2/4) part 4 → Fixed Window Counter Algorithm →

(3) Fixed Window Counter Algorithm →

→ Timeline – Fixed time windows (Timeline को fixed windows में बाँट देते है)
→ Each window has a counter
→ Counter increment +1 per request.
→ Counter reaches threshold – new requests dropped – Until new window starts.

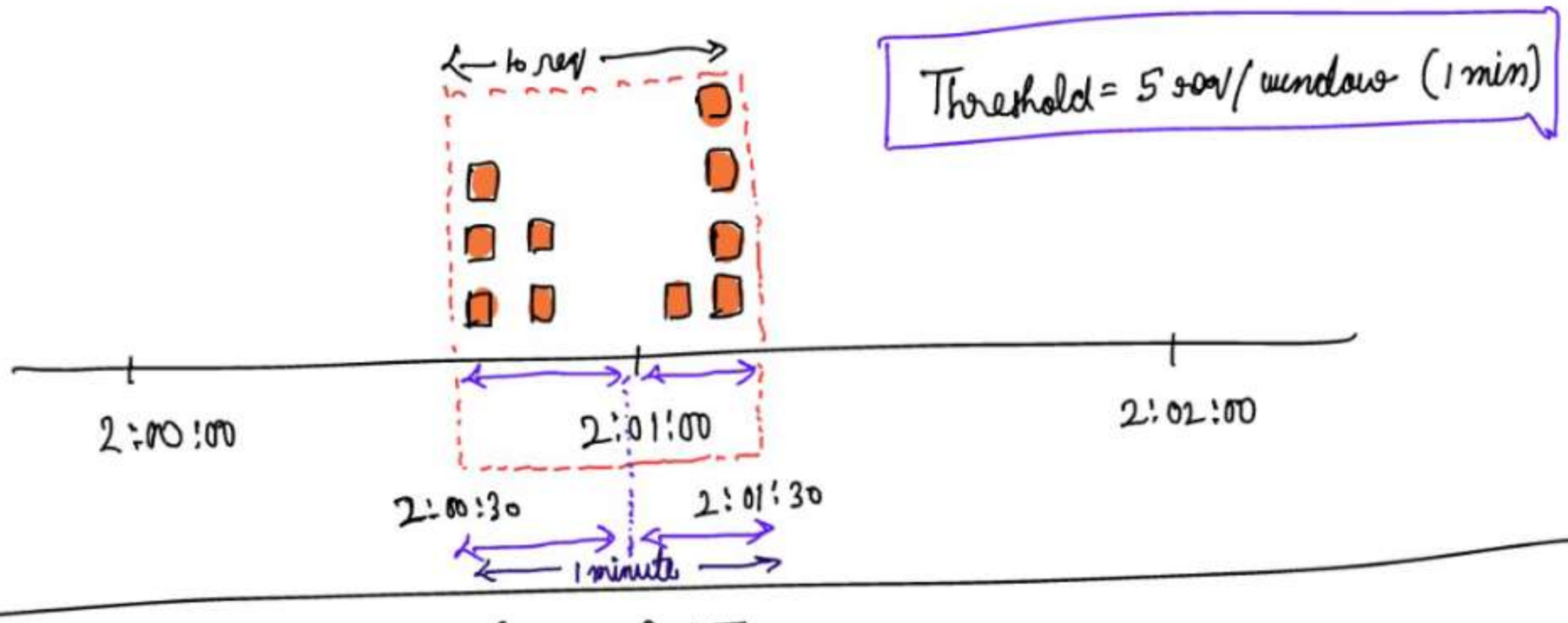






Pros :- Easy to understand & Implement.

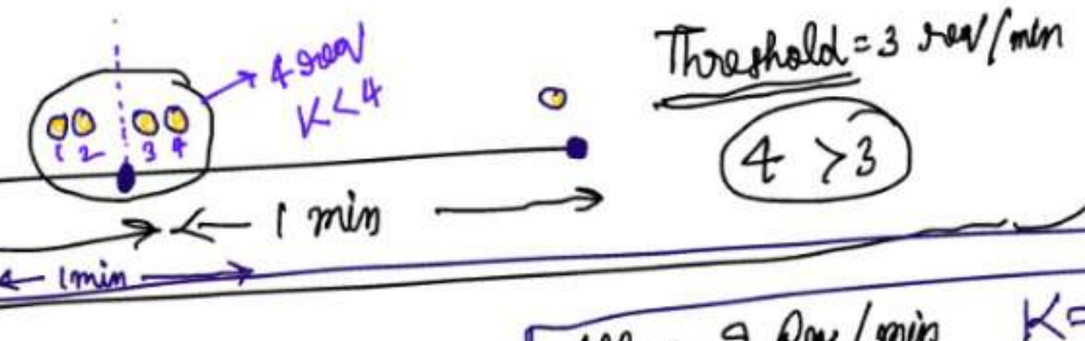
Cons :- Traffic Burst at edges of a window cause more request than the Threshold.



(14) Design Rate Limiter (2/4) Part IV →

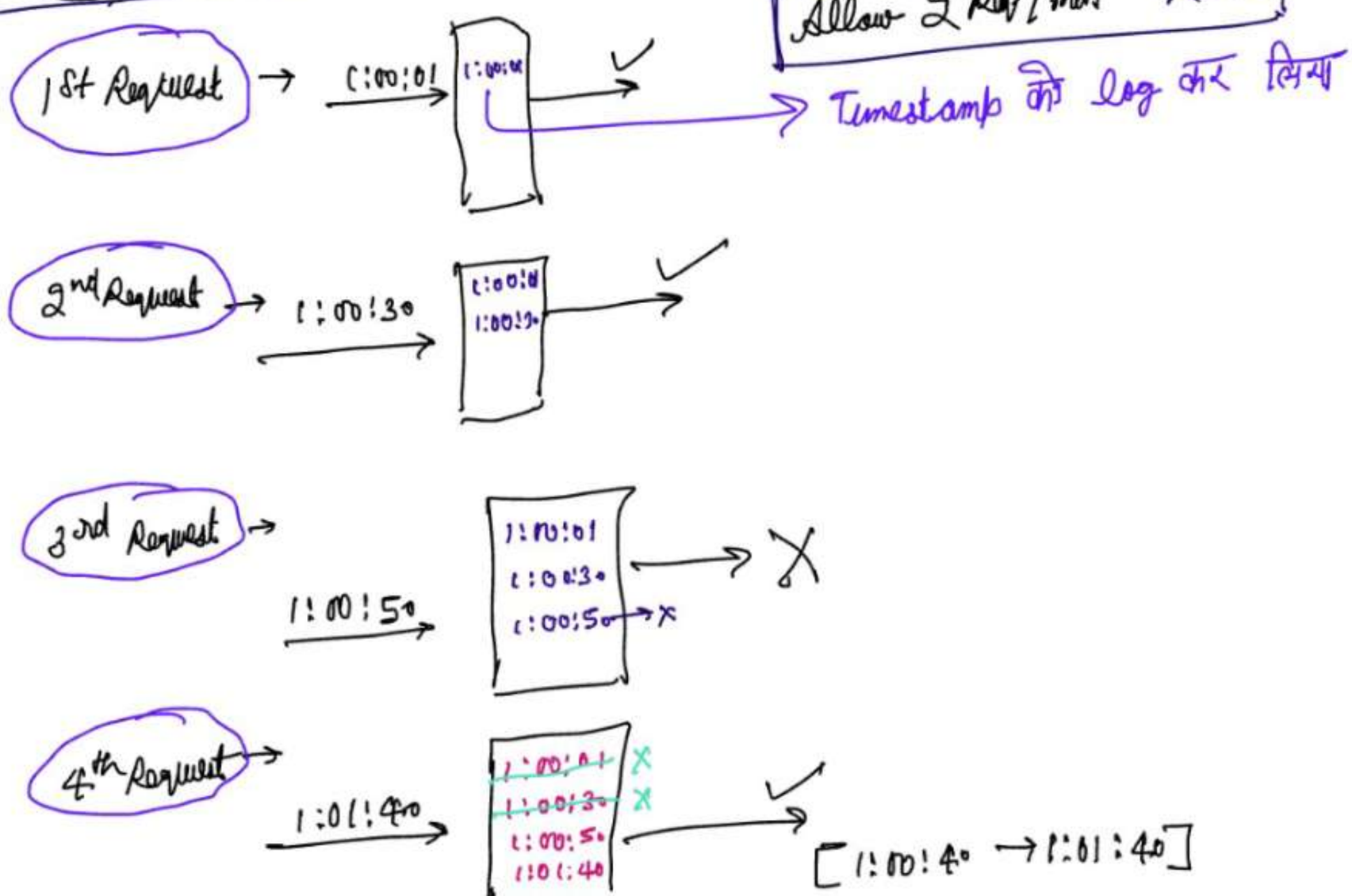
(4) Sliding Window Log Algorithm →

→ fixed window counter Algorithm का drawback था



→ To fix this we use Sliding window log Algorithm.

Allow 2 Req/min K=2



Timestamp को log कर लिया

[1:00:40 → 1:01:40]

→ log every req's timestamp (हर request का timestamp save करके रख लेंगे)

→ New Request — remove outdated timestamps (पुराना log हट्या देंगे)

→ New Request का timestamp add कर लो

→ Timestamps का count ≥ threshold → Rejected

Pros →

→ Accurate

→ Will not exceed threshold

Cons →

→ Even if Req. Rejected, logged in.

→ Cache. (Memory usage high) (Memory लग रखी है)

---

(15) Design Rate Limiter (2/4) (part 6) →

High level Architechture →

Basic Idea → High level में जाते तो

→ Counter → To keep track of no of requests.

→ same user, same IP etc.

→ counter > Threshold : Discard Request.

Interviewer → Where shall we store counters?

candidate → एकदम से जवाब बदल दिये वक्त बदल दिया

Interviewer: Where to store counter?

candidate: using a database

Interviewer: Dont you think it will be slow due to slowness of disk access.

candidate: In-Memory Cache
- ↳ Fast अति!
- ↳ Time based expiration strategy support.

Interviewer: Can you give an example?

Candidate: Redis Cache is a popular option.

Interviewer: Can you tell me more about redis?

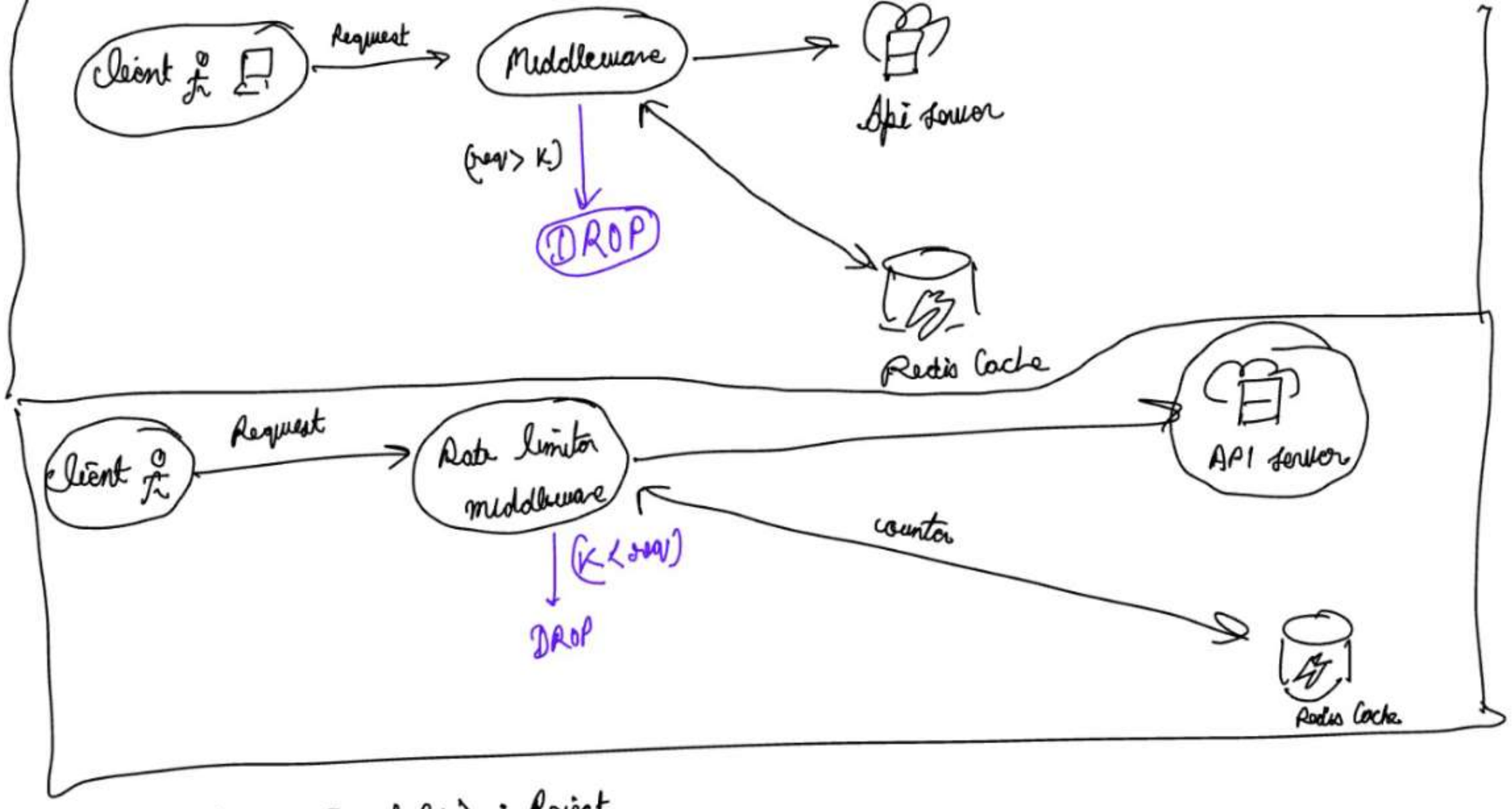
Candidate: → In-Memory fast store

→ offers 2 commands

↳ INCR : counter++

↳ EXPIRE: Timeout for counter

High Level Diagram →



if (counter > Threshold) : Reject

---

(16) Design Rate Limiter (3/4) Part ① →

part ① → Design Deep Dive →

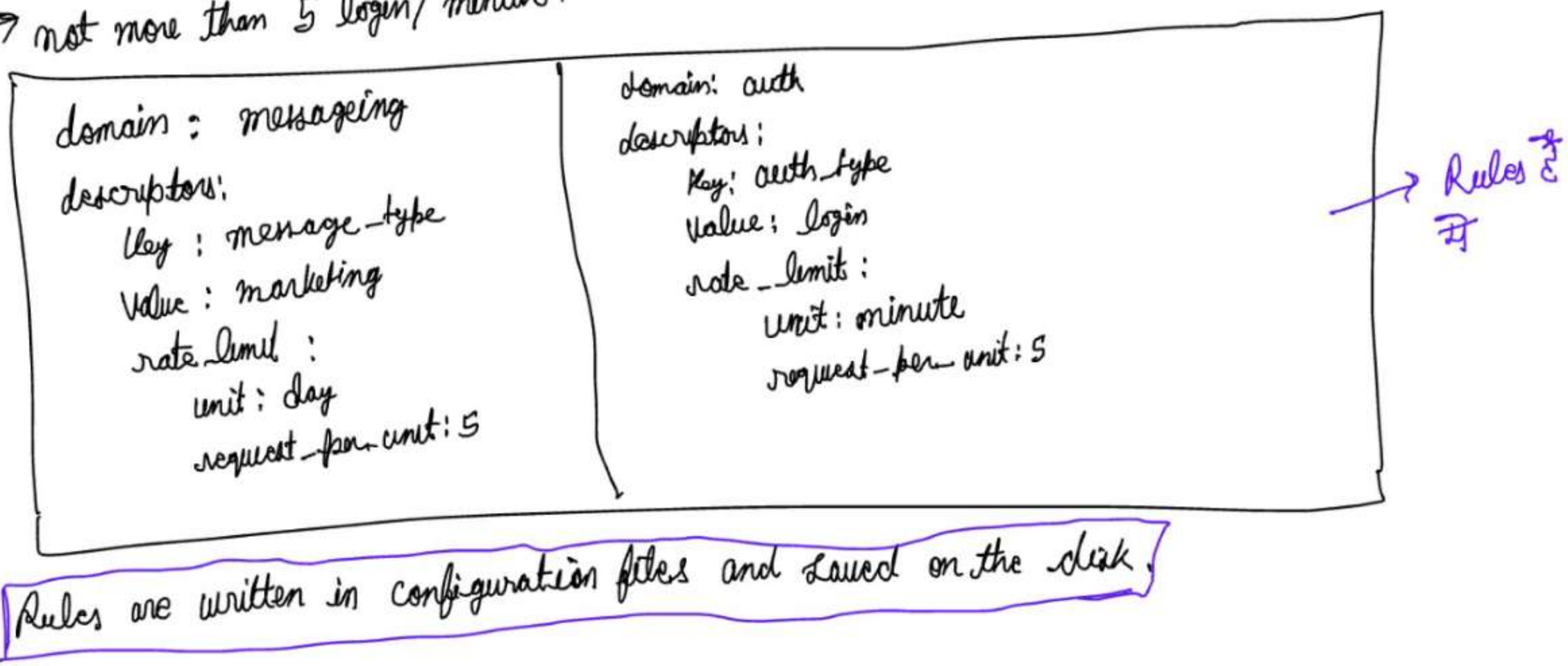
Interviewer → How are rate limiting rules created ?
कहाँ store करोगे rules को ?
जो request limit हुए उनका क्या करोगे ?

Candidate → First , lets talk about limiting rules

Rate limiting Rules →

Example :- lyft (UBER , OLA)

→ max 5 marketing message / day
→ not more than 5 login / minute.



Rules are written in configuration files and saved on the disk.

Candidate: Coming to the next question

* Exceeding The Rate Limit →

जब कोई request limit हो जाती है, then what.....

→ Send HTTP response code 429 to client

→ 429 : Too many request

→ Depends on use case what to do next :-

↳ Enqueue → processed later (Amazon का order enque होता है)

Example → Some of your orders got rate limited.

Interviewer:- How will the client know ?

↳ उसको Request rate limit हो गया ?

↳ उसके पास कितने Request बचे हैं before being rate limited ?

Candidate → Via "HTTP Response Headers" sent by rate limiter.

Headers	Meaning
X- RateLimit Remaining	कितने Req बचे हैं।
X- Rate limit -Limit	Max kitne req you can send
X- RateLimit -Retry -After	Kitne time baad you can send req.

So if the client has sent too many request,

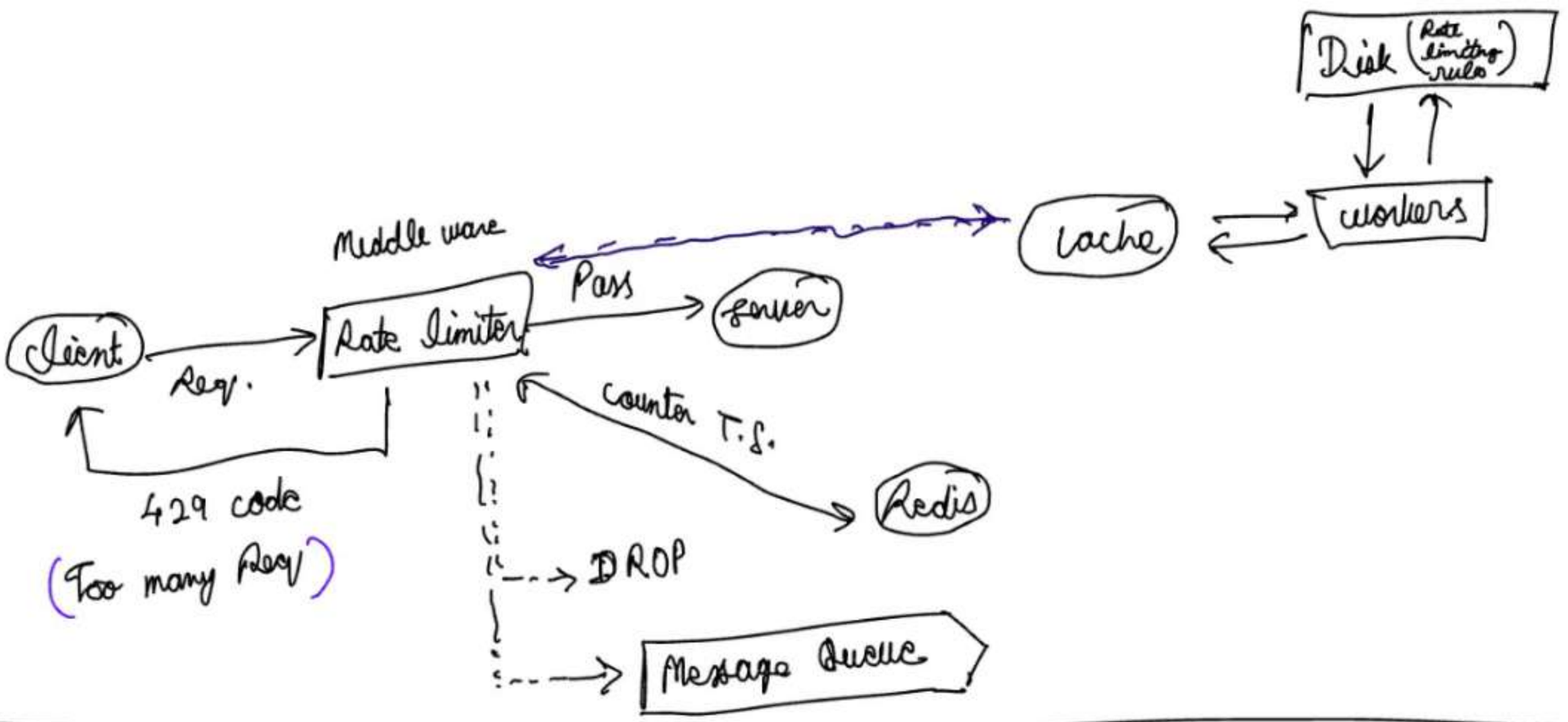
→ 429 (Too many Req)
→ X- Rate Limit -Retry after

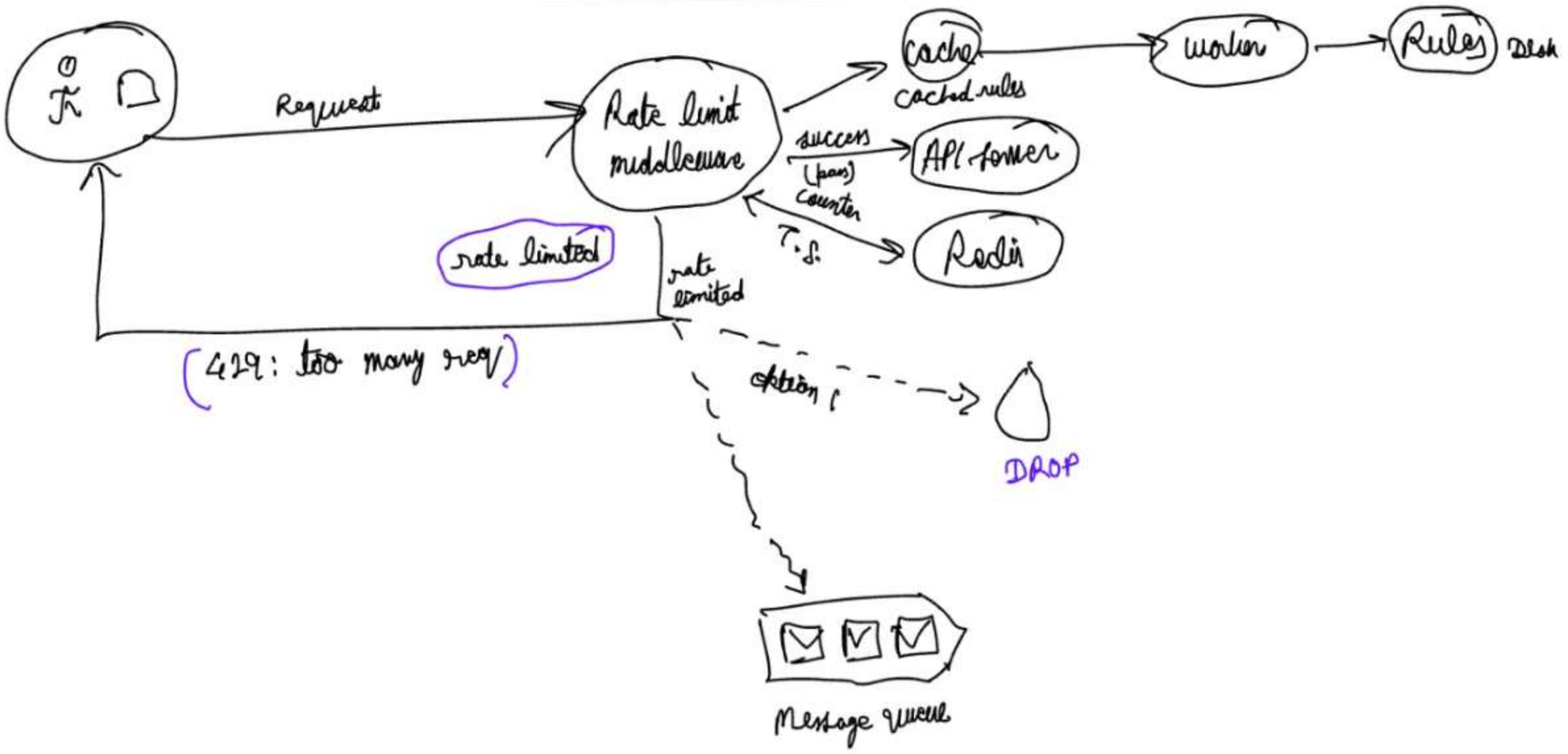
(will be sent to client)

17 Step ③ Design Deep Dive (Part 2)

Detailed Design →

→ Rules को Disk में रखना पड़ा (Remember ???)
→ Disk slow है तो Cache बीच में रखलेंगे
→ Req तो आता रहेगा - Use Workers → time to time disk se Rate limiting rules nikal ke cache me daalte rhega time to time
→ Client होगा
→ Server होगा जिसको client req भेजेगा
→ बीच में Rate limiter
→ याद करो Rate limiter, counter Redis में रखता था
→ Request rate limit हुआ तो
↳ (429) code send
↳ Drop it
↳ or enqueue it for later process





(18) Design Rate limiter (3/4) part 3 → Rate limiter in Distributed Environment →

Q → Any challenges for Rate Limiter in Distributed Environment ?

A → There are mainly 2 challenges :-

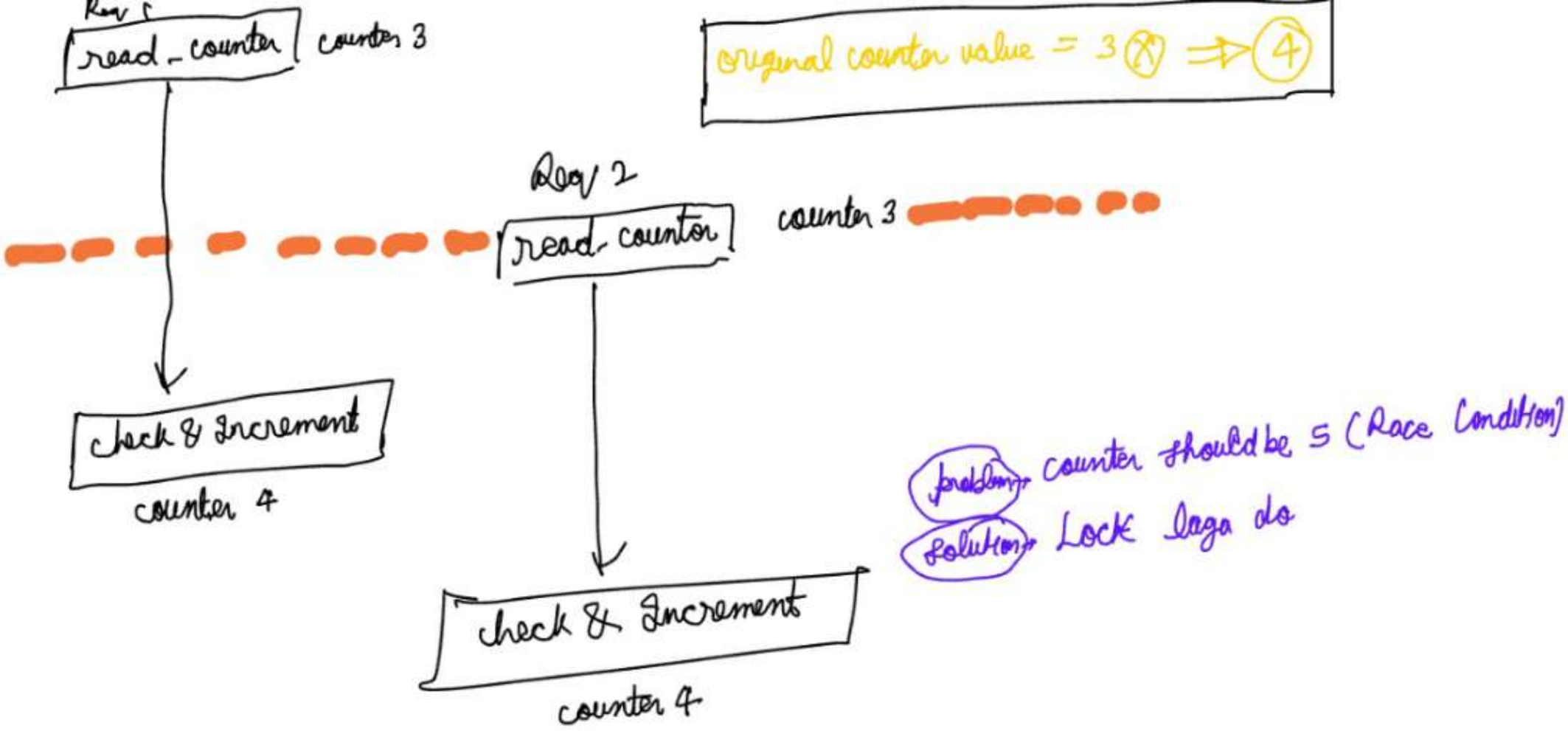
- (i) Race Condition
- (ii) Synchronization Issue

Q → Can you elaborate ?

A → Race Condition → Rate limiter at High level
→ Read Counter from Redis
→ Check if (counter+1) > Threshold.
→ If Not, INCR

अगर concurrently request आयेंगी then

original counter value = 3



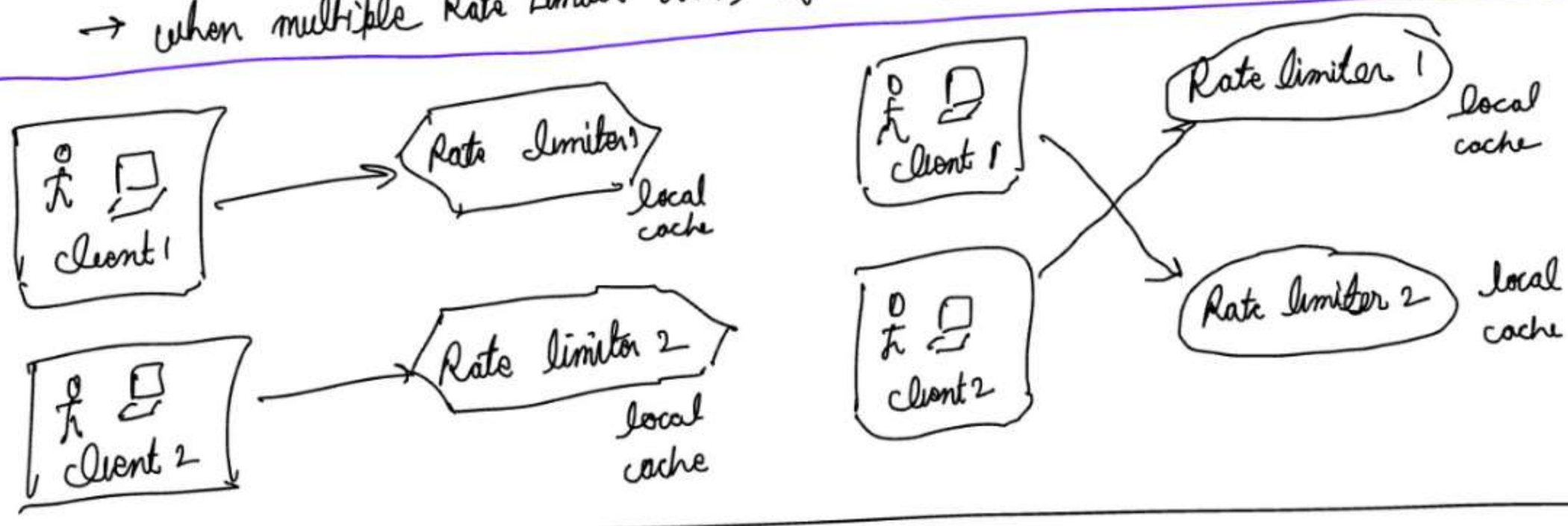
Note → Locks are obvious sol^n, But it slows down the system.

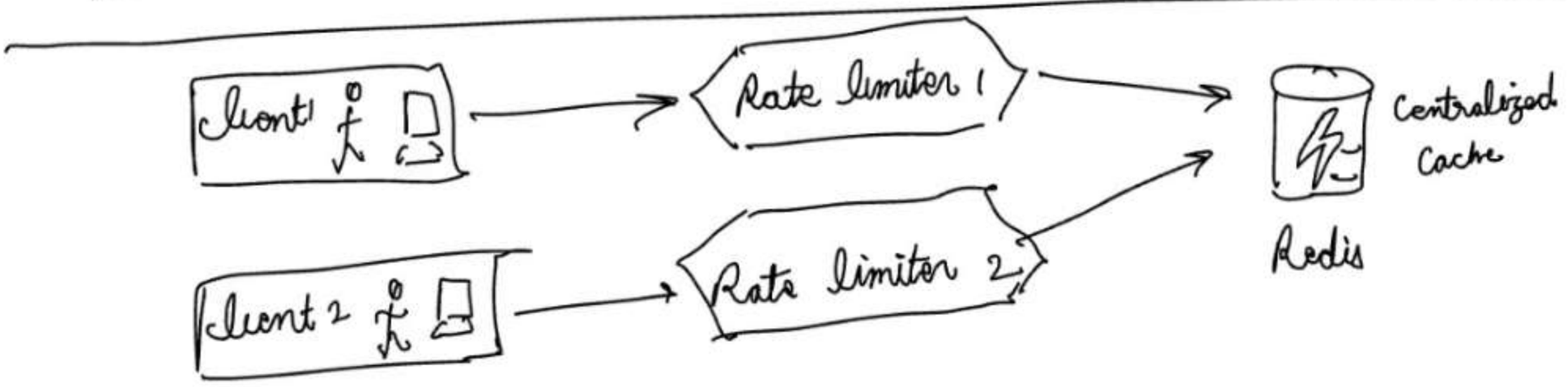
✱ Synchronization Issue →

→ Millions of Users

→ 1 Rate Limiter is not enough

→ when multiple Rate Limiter used, Synchronized must be there





✱ Synchronization Issue →

→ millions of users

→ 1 Rate limiter is not enough

→ when multiple Rate Limiter used, Synchronization must be there

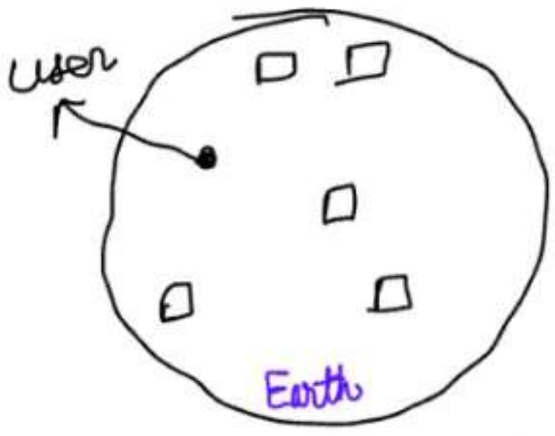
(14) Design Rate Limiter (3/4) (part 4) Performance optimization & Monitoring →

Q → Any areas where we can improve ?

A → → Multi Data Center Setup.

→ Better Synchronize data.

→ latency ↑ ; users far away from data center



Edge server locations around world.

Interviewer → How will you check if your rate limiter is effective ?

Candidate → By monitoring metrics (gathering analytics data)

Mainly 2 things to ensure :-

* Rate limiting Algorithm – Effective ?
* Rate limiting Rules – Effective ?

Interviewer → Rate limiting rules are too strict ?

Sudden increase in traffic in flash sales ?

क्या करोगे ?

Candidate → Relax the rules a bit bcz otherwise many valid request might be throttled.

Replace algorithm to support burst of traffic – TOKEN BUCKET फिर दोबारा

↳ Burst of traffic support कर सकते हैं।

---

(20) Design Rate limiter (4/4) wrap up →

Additional talking points → If time allows !!!

* Hard Rate limiting ⇒ Request <= Threshold (Always)
* Soft Rate limiting ⇒ Can exceed threshold for a short period.

Interviewer → As a client, what will you do to avoid your requests being Rate limited?

Candidate → * Use client Cache → Avoid frequent calls to API.

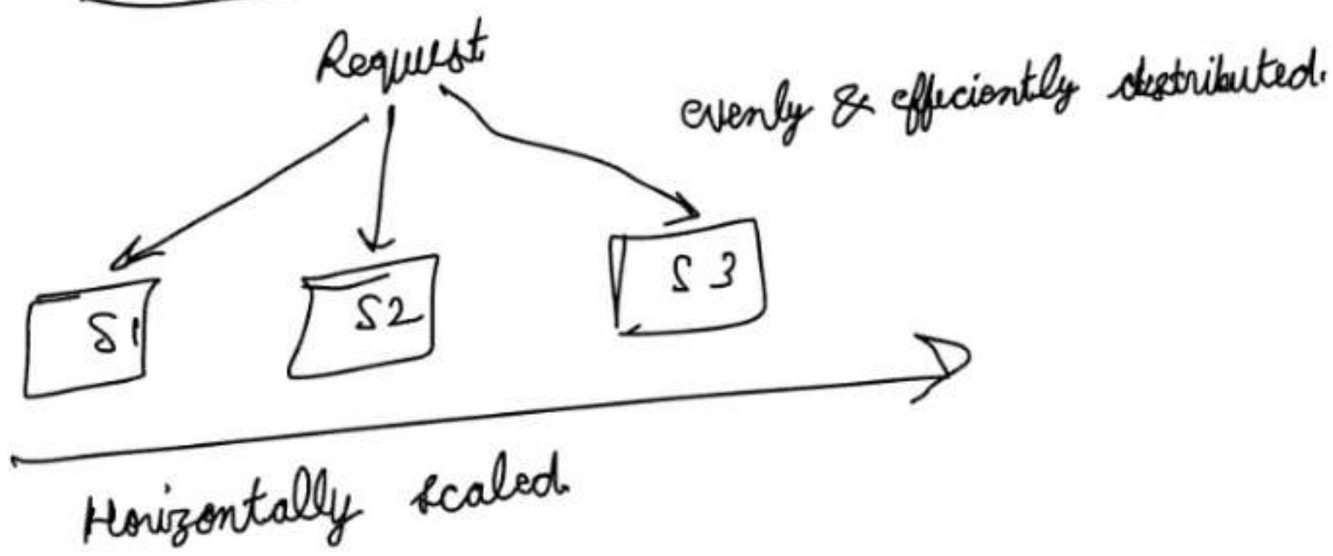
* Do not send too many requests as per the threshold limit within a time frame.
* Code to catch exceptions.
* Add sufficient back off time to retry request call.

Recap →

→ Rate limiting algorithms
→ System architecture
→ Distributed Environment
→ Optimization & monitoring

(21) Design Consistent Hashing (what the problem ? why need it?) →

* Remember Horizontal Scaling →



e.g. →

Employee Record

Email_Id	Details
xyz @ gmail . com	~
abc @ gmail . com	—
def @ gmail . com	—
ghi @ gmail . com	—

→ no of records ↑↑↑ Increasing ↑↑↑

→ cant store in one server

→ Distribute data in multiple servers

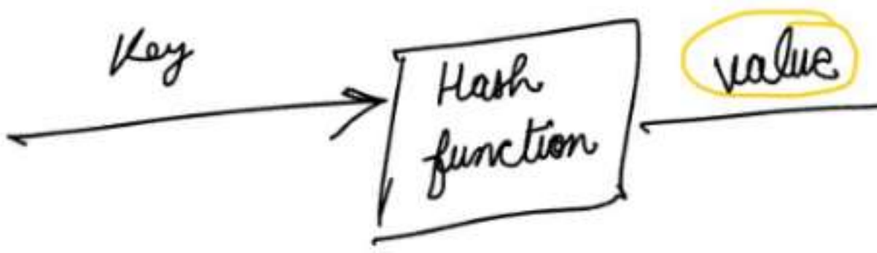
→ How to know which server to look up for particular email-id ?

Problem !-

we will use email-id as the key

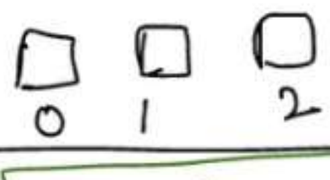
cache servers = n

Server Index = hash(key) % n



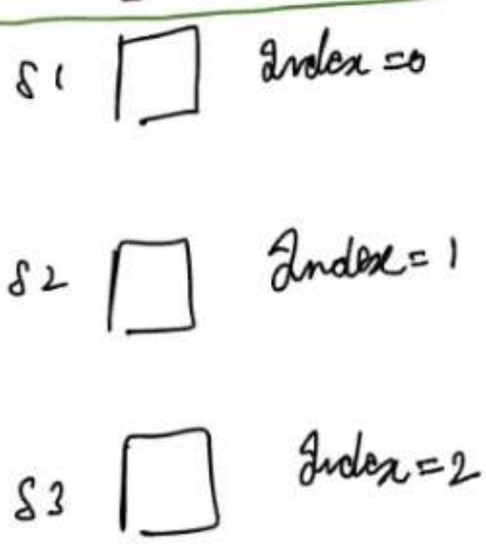
Index = value % n

= value % 3 ⇒ {0, 1, 2}



Email (Key)	Hash	Server
john @ example . com (S3)	89 % 3 = 2	2 (S3)
mark @ example . com (S1)	30 % 3 = 0	0 (S1)
adam @ example . com (S3)	47 % 3 = 2	2 (S3)
smith @ example . com (S2)	52 % 3 = 1	1 (S2)
alex @ example . com (S1)	75 % 3 = 0	0 (S1)
bob @ example . com (S2)	22 % 3 = 1	1 (S2)

Server [Index = Hash % 3] (N=3)



Email (Key)	Hash	Server
john@example.com	89%2=1	1 (S2) ~~S3~~
mark@example.com	30%2=0	0 (S1)
adam@example.com	47%2=1	1 (S2) ~~S1~~
smith@example.com	52%2=0	0 (S1) ~~S2~~
alex@example.com	75%2=1	1 (S2) ~~S1~~
bob@example.com	22%2=0	0 (S1) ~~S2~~

↳ Rehashing

Index = Hash % 2

N = 3 → 2

S1 □ index = 0

S2 □ index = 1

S3 [X] cache miss

crashed

John

(22) Design Consistent Hashing How it works ?

Previous Video → Rehashing करना पड़ा था after 1 server failure

"Consistent Hashing" comes to the rescue → It is independent of no of servers

Email (Key)	Hash	Server
john@example.com	89%3=2	2 (S3)
mark@example.com	30%3=0	0 (S1)
adam@example.com	47%3=2	2 (S3)
smith@example.com	52%3=1	1 (S2)
alex@example.com	75%3=0	0 (S1)
bob@example.com	22%3=1	1 (S2)

Index = Hash % 2

S1 □ index 0

S2 John index = 1

S3 Crashed cache miss

John

→ Does not depends on no of servers

→ Use a Hash function.

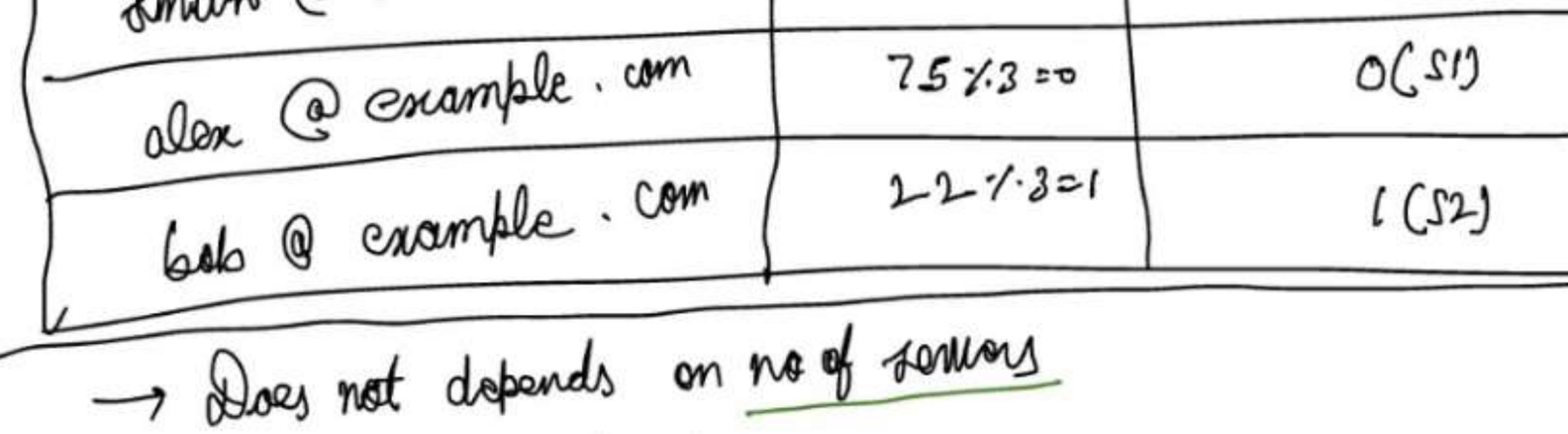
↳ maps servers in the Hash Ring

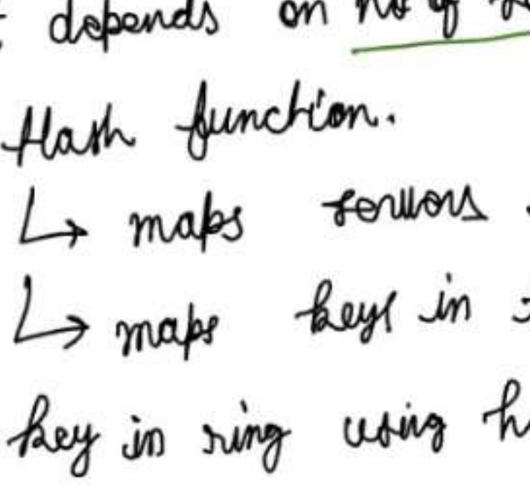
↳ maps keys in the Hash ring.

→ Locate key in ring using hash function

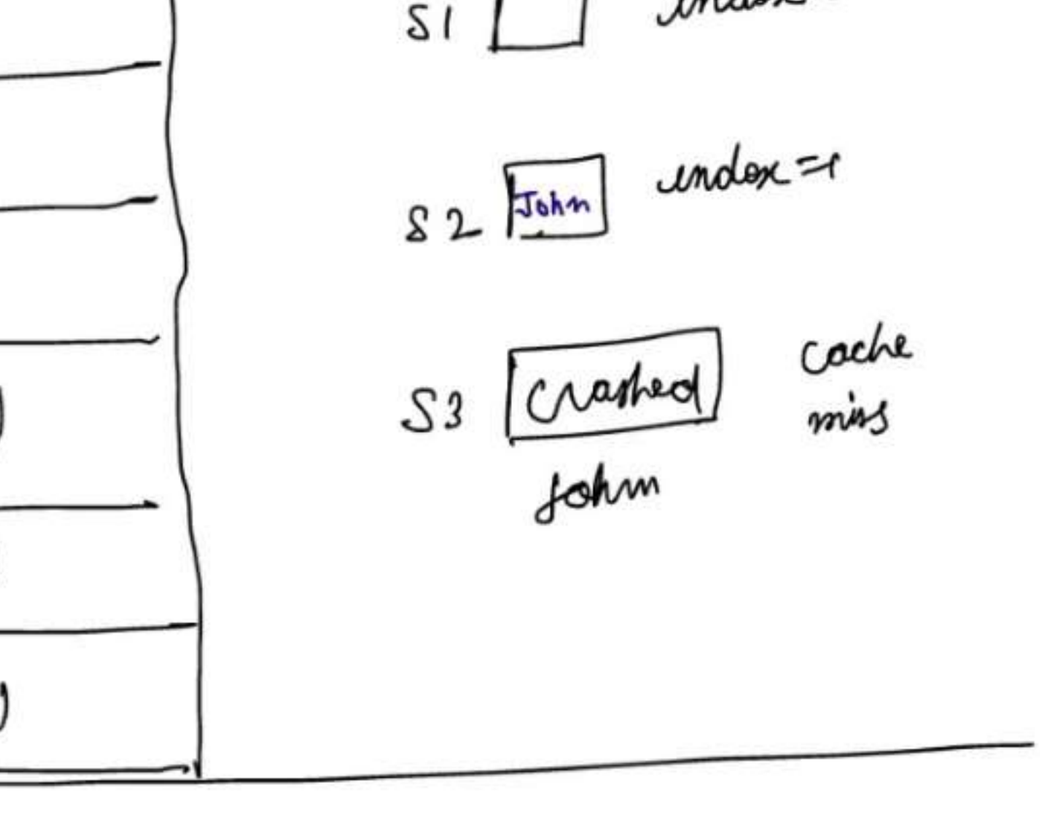
→ Go clockwise until you find a server.

hash(key) = $[x_0, x_n]$





Hash ring



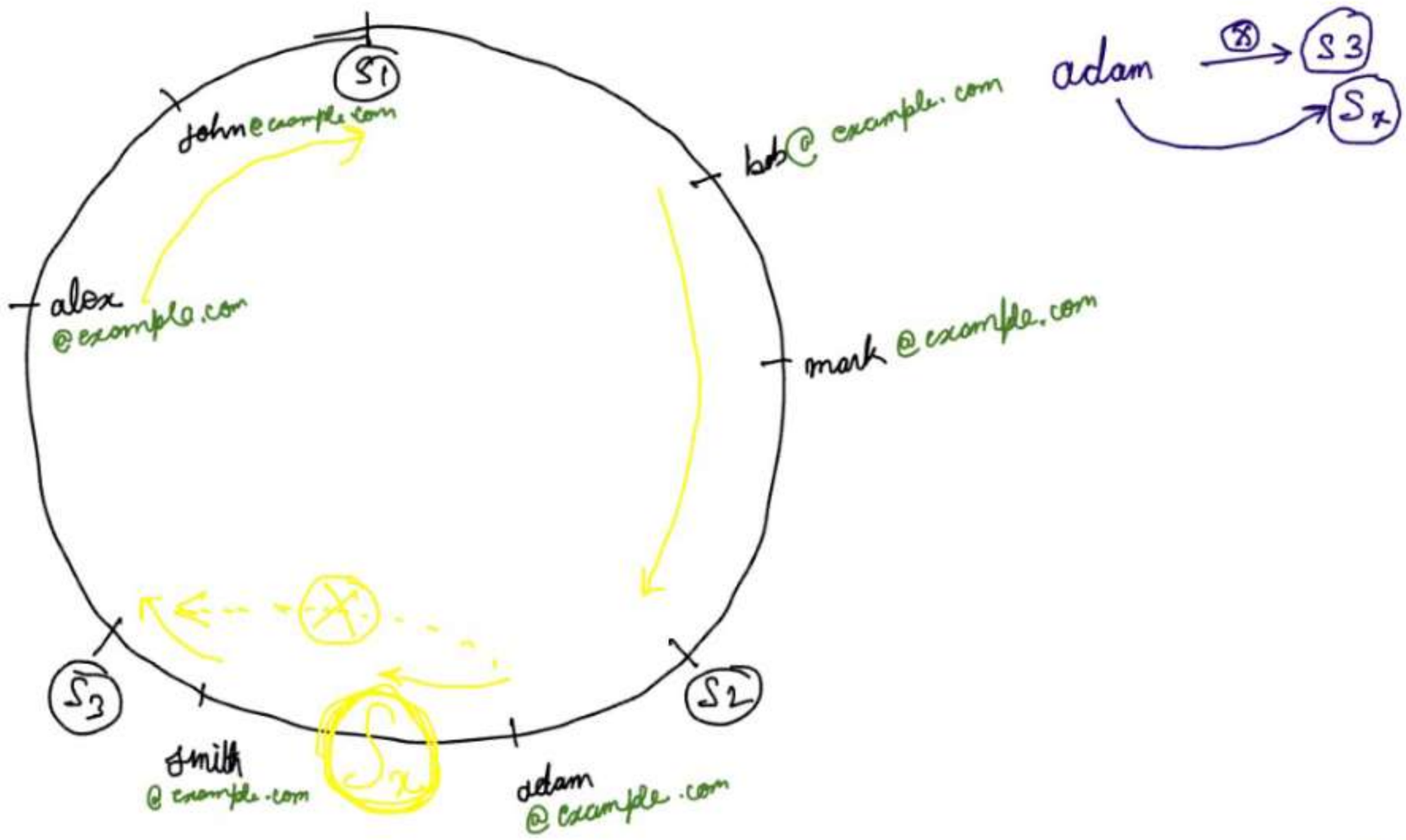
→ Hash Server

→ Add a Server

→ Hash Key

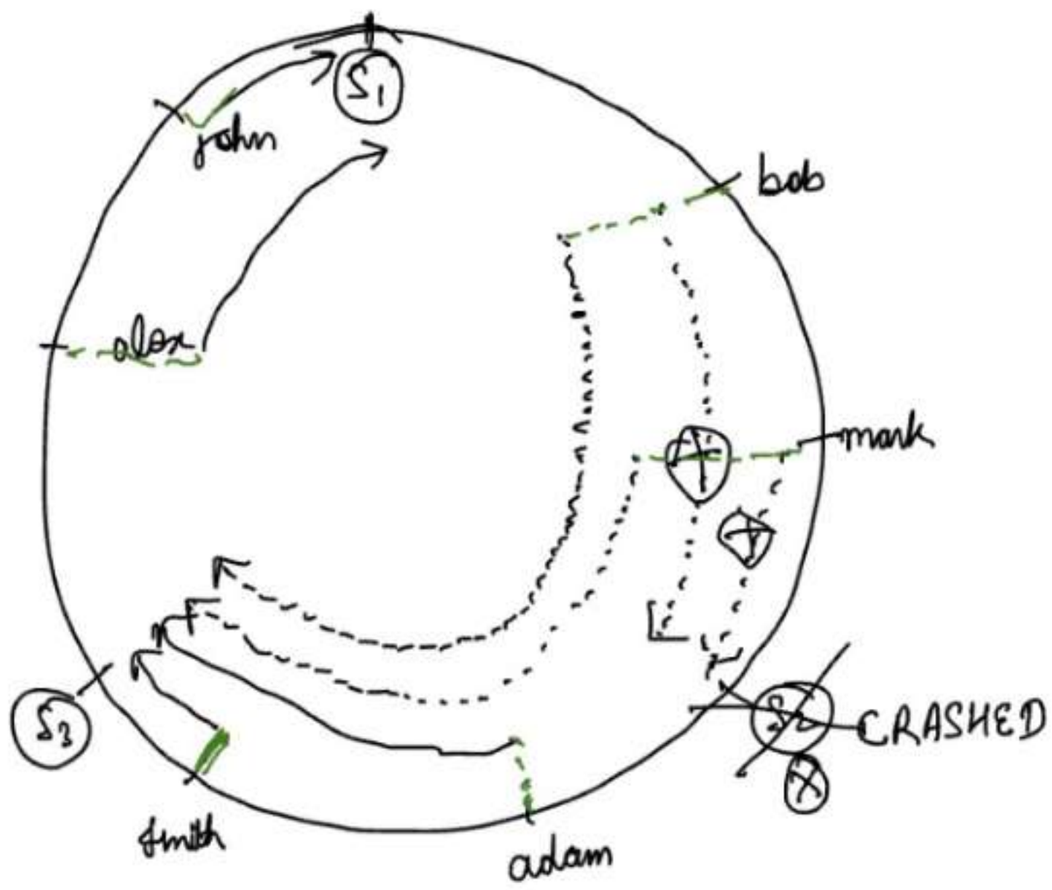
→ Remove a Server

✡ Adding a Server →



adam → ⊗ → S3
→ Sx

✡ Removing a Server →



S2 Crash कर गया

Bob & Mark → ⊗ → S2
→ S3

---

(23) Design Consistent Hashing (Two Problems with Basic Approach) →

Two Issues → 😟

Recap → map servers and keys on to a ring. [Basically ring nhi hota 1D space hota hai]

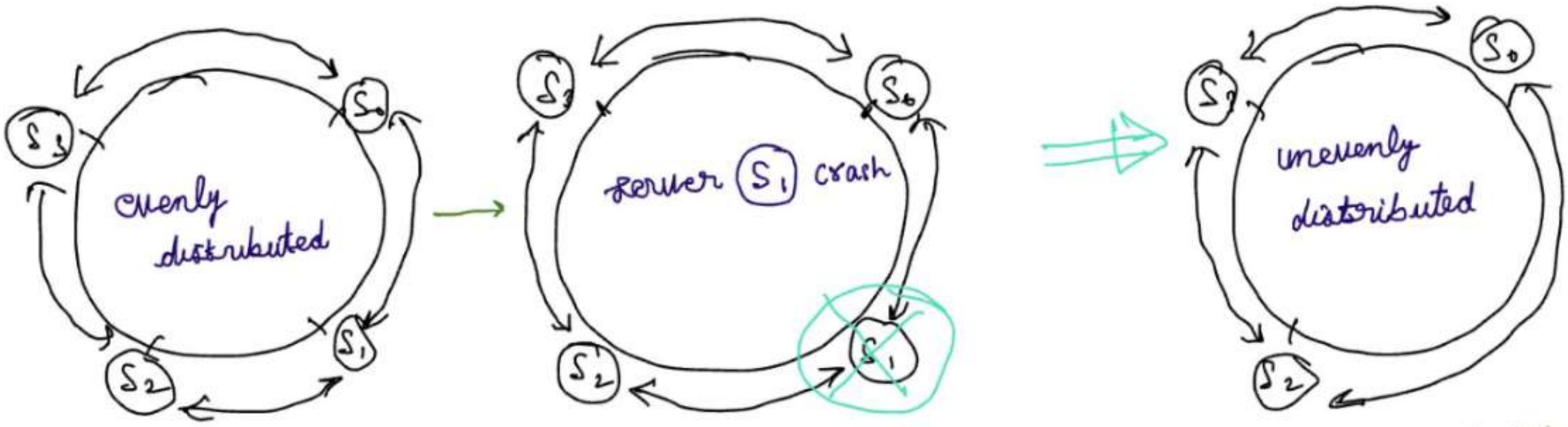
→ Using uniformly distributed Hash function.

→ To find server, key से clock-wise चलो ।

Interviewer → Can you point out problems above??

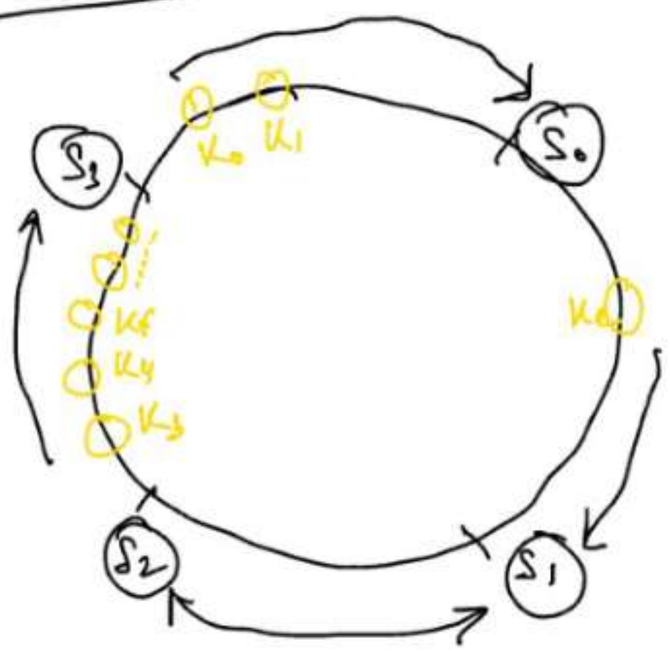
* 2 problems with above Basic Approach →

Problem ① Ring में सभी servers के बीच का partition हमेशा same होना impossible है।
→ क्यों कि servers add & remove भी हो सकते हैं।



हमेशा evenly distribute नहीं होगा

Problem ② Keys का Non-Uniform distribution भी तो हो सकता है।



$K_0\ K_1 \rightarrow S_0$

$K_2 \rightarrow S_1$

$K_3\ K_4\ K_5 \rightarrow S_3$ → इस पर load ज्यादा आ जायेगा

Q→ What to do to solve those problems above?

A→ Virtual Node

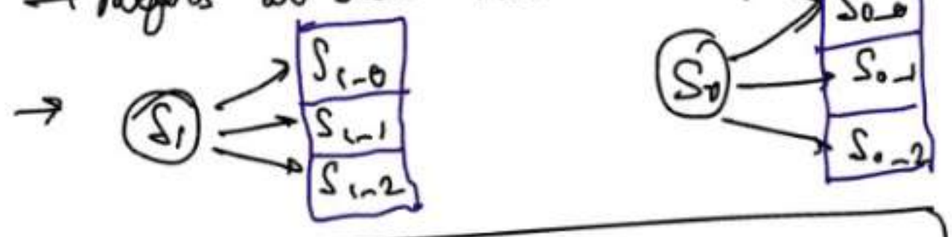
---

(24) Design Consistent Hashing (Virtual Nodes) →

Virtual Nodes →

→ Refers to real node actually

→



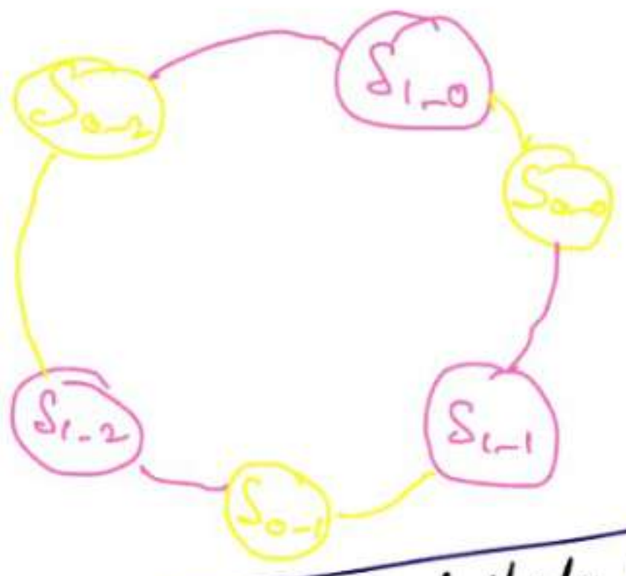
} 3 Virtual Nodes created
Real Scenario mein virtual node bahut zyada hota hai.

$S_{1-0}\ S_{1-1}\ S_{1-2}$ represents server S_1
virtual Node

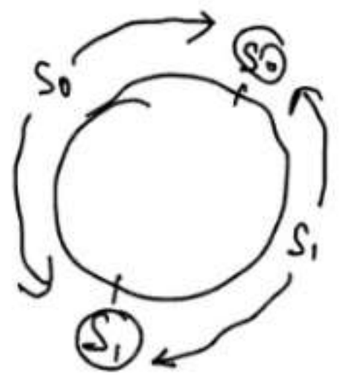
$S_{0-0}\ S_{0-1}\ S_{0-2}$ represents server S_0
virtual Node

Note → Real World System # Virtual Nodes ⇈ → Real world system mein virtual nodes bahut zyada hote hai.

→ Multiple small ranges ~~of~~ partitions →



with virtual Nodes

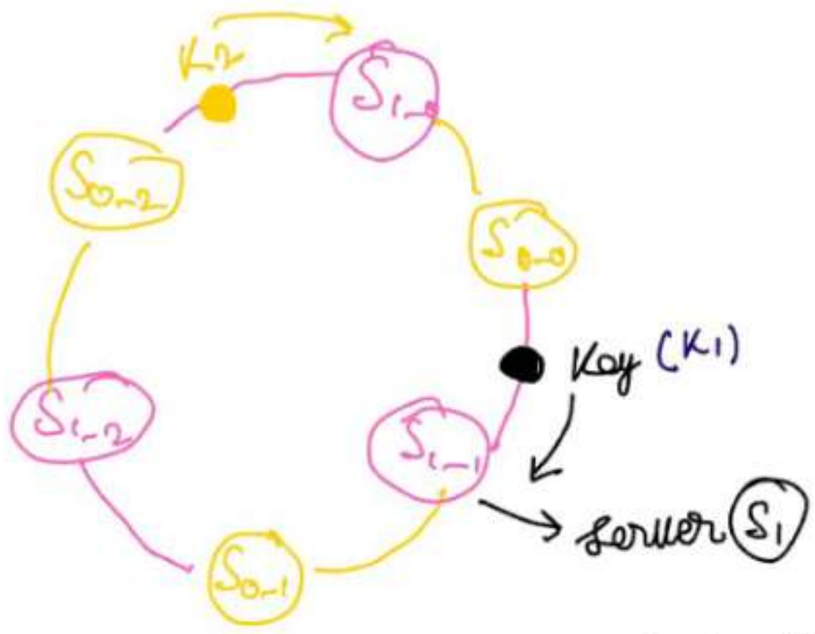


Without virtual Nodes

* Locating which server has data for a key.

$K_2 \rightarrow S_{1\_0}$ (S_1)

$K_1 \rightarrow S_{1\_1}$ (S_1)



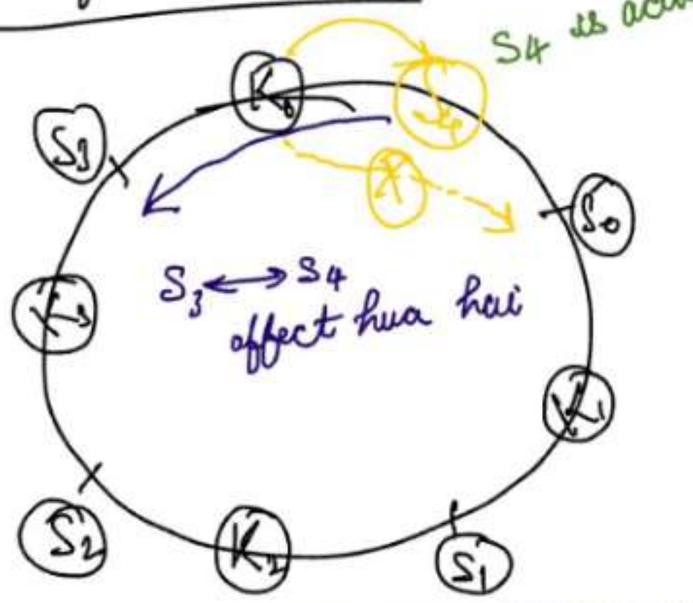
Interviewer → How will you find affected Keys ?

When :-

↳ Server is added

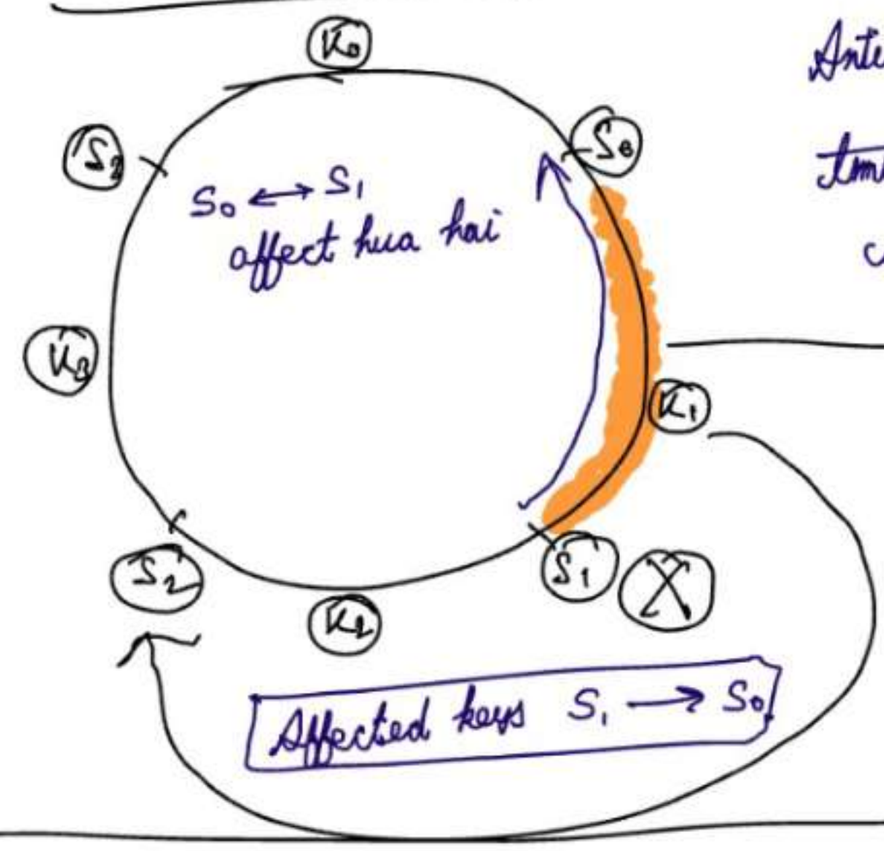
↳ Server is removed

* Adding a server →



Affected Keys → $S_4 \rightarrow S_3$

* Removing a server →



Anticlock wise jao

timko affected keys data

chal jayega

→ affected region.

---

(25) Design Consistent Hashing (wrap up) →

Note → Always do a quick recap / summary for interviewer

Discord mein consistent hashing use hota hai.

* We now know
  → Why need ?
  → How it works ?
  → Benefits
    ↳ Minimum keys are redistributed (add, remove) fewer
    ↳ Since data can now be evenly distributed
      → Easy to "Horizontally Scale".
    ↳ Avoid Hotspot Key Problem.

* Widely Used in real-world Systems →
  → Amazon Dynamo DB
  → Apache Cassandra
  → Discord Chat Application.

---

(26) Design Key-Value Store →

→ Also known as - Key value Database
→ Non-relational Database (NoSQL Database)
→ Key-value pair ⟶ Requirement Gathering
→ Key must be Unique → Requirement Gathering
→ Key → Plain Text → e.g - last_login_at
        Hashed Key → e.g → 259CFD25
→ value → strings, list, object etc.

Key	Value
145	Sachin
260	Amar
330	Akbar
210	Anthony

Interviewer → I want you to design a key-value store that supports :-
→ put (Key, Value) // insert value associated with "key"
→ get (key) // get "value" associated with "key"

Candidate → What should be the size of key value pair?
Interviewer → Less than 10KB
It should be able to store big data

Candidate → What about 'Availability', 'Scalability'?

Interviewer → **Availability** → HIGH, respond quickly

Scalability → Should be scalable to support large data set.

Note → Scaling should be 'Automatic' → Traffic के हिसाब से server add or delete होना चाहिए

i.e. → addition / deletion of servers automatically based on traffic.

Candidate → What about consistency & Latency?

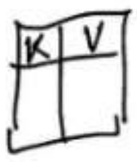
Interviewer → **Tunable Consistency**

Low Latency

---

(27) Design Key-Value Store (part 2) single server key value store →

→ Easy to implement

→ Hash Table use करते हैं 

→ will keep everything in memory

→ Memory access is fast but memory limited space constraint.

Interviewer → Any improvement you can do?

Candidate → Data को compress करके store करेंगे

→ Frequently used data = Memory में रखेंगे

→ Less frequently used = Disk में store करेंगे

Interviewer → Will it solve all our problem?

Candidate → No,

Even with above optimization

Single server will easily reach its capacity limit.

Interviewer → Then anything else we can do?

candedate → We will use **Distributed Key-value store** (Across many servers) → Same data को multiple servers में distribute करेंगे

→ एक ही server पर सारा load नहीं होगा

Interviewer → Sounds good. Enlighten me more on this.

(28) Design Key value store (Part 3) → Distributed Key value store and CAP Theorum →

→ Distributed Hash Table

→ Distributed Key value pairs across many servers. (कई सारे servers के across distribute करता है).

→ Distributed System design के लिये → we must know CAP Theorum

CAP Theorum ⇒

Ⓒ → Consistency → All candidate see the same data at the same time no matter which node they connect to

Ⓐ → Availability → Any client requesting data gets response even if some of the nodes are down.

Ⓟ → Partition Tolerance

↳ Partition → Communication break b/w 2 nodes.

↳ Tolerance → तब भी system continues to operate.

CAP Theorum → one of the 3 properties has to be sacrifised to support 2 properties.

"pick 3 impossible"

→ Now a days, Key-value stores are available as →

Ⓐ CP systems → sacrificed → Ⓐ → Availability

Ⓑ AP system → sacrificed → Ⓒ → Consistency

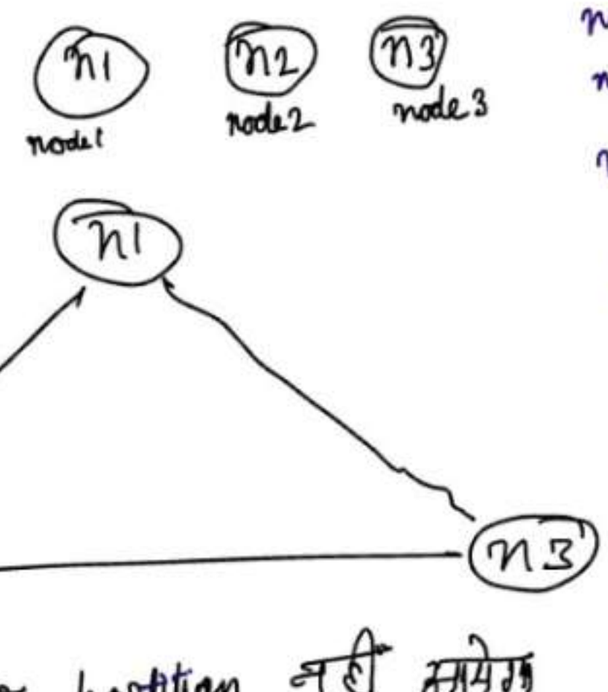
Ⓒ CA system → sacrificed → Ⓟ → Partition Tolerance

↳ since n/w failure is unavoidable (CA) is not possible → Real life में CA possible नहीं है।

* Example से समझते है :-

→ Distributed System में data is usually replicated multiple times.

→ Assume data replicated on 3 replica nodes :-



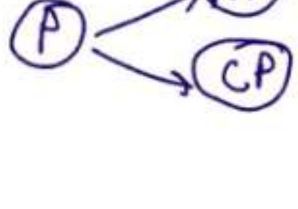
node1 = n1
node2 = n2
node3 = n3

जो data n1 में है वो data n2 & n3 में भी है।

C
A
P } तीनो है

But Real life में ऐसा नहीं होता

Agar को Ⓟ तो रखना ही पड़ेगा



} यही 2 संभावना होंगी Real life में

Ideal situation → n/w partition नहीं आयेगा

Data written on n1

replicated to n2 & n3

Consistency & availability achieved

Real World Distributed System →

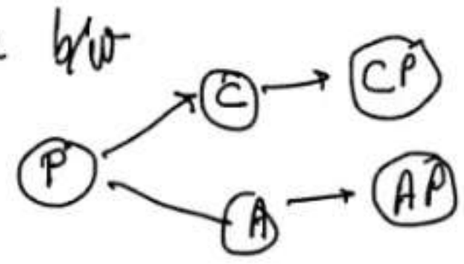
→ Partition cant be avoided

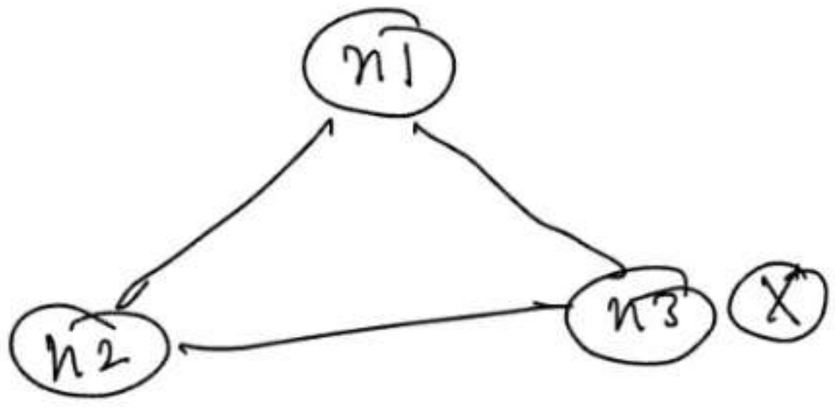
→ यानि Partition tolerance तो रखना पड़ेगा

Now you need to choose b/w

@ Consistency

@ Availability





goes down.

→ n3 goes down, cant communicate to n1 & n2

→ Clients writes on n1 OR n2 ; wont propagate to n3

→ Clients writes on n3 , wont propagate to n1 & n2

→ If we choose consistency (C P) →

→ you want consistency

→ you will Block all writes operation on n1 & n2 (Bank System) (C P system)

→ system unavailable हो जायेगा

e.g → Bank System

↳ up to date Balance info.

↳ जब तक inconsistency ख़त्म return error until resolved

→ If we choose Availability (C A) →

→ system keeps accepting reads

→ stale data

→ writes will be done on n1 & n2

→ will be synced to n3 when resolved.

Note → CAP का correct combo as for your need is a crucial step for Distributed Sys Interview

→ so discuss / ask your interviewer की उनको कौन सा चाहिए!! → CP
AP
CA → Not possible in Real life.

(29) Design Key-Value Store (Part 4) Data Partition and Data Replication →

Core Components & Techniques to build Key-Value store →

→ Data Partion
→ Data Replication
→ Consistency
→ Inconsistency Resolution
→ Handling failures
→ System Architecture Diagram
→ Write Path
→ Read Path.

Data Partition & Data Replication →

* Data Partition →

→ Large Application
→ Infeasible to fit data in single server.
→ Split data (smaller partition) ⟹ (multiple servers)
→ Two challenges :-
  ↳ Distribute data across multiple server evenly (data को evenly distribute करोगे)
  ↳ Minimize data movement when nodes are added OR Removed. (data movement कम से कम होना चाहिए)

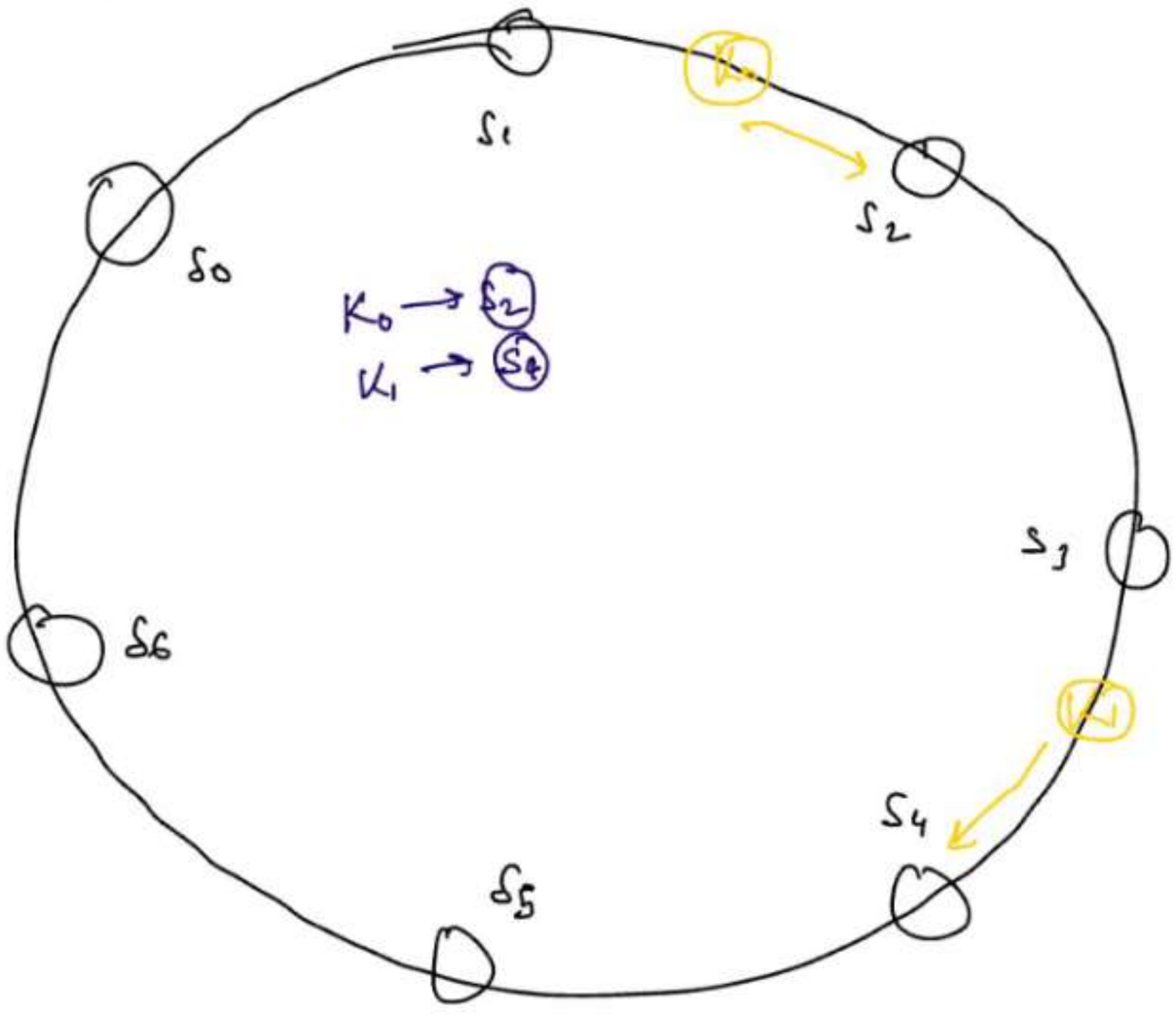
याद करो, who helped us to fix above challenges ??

Yes ☺ → Consistent Hashing.

Revise करते है थोड़ा. →

↳ Server placed in Hash ring.
↳ Key is hashed in same ring.

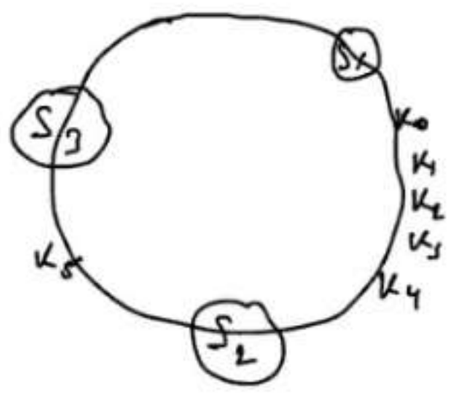
Data → clockwise जाओ .....



$K_0 \rightarrow S_2$
$K_1 \rightarrow S_4$

Benefits →

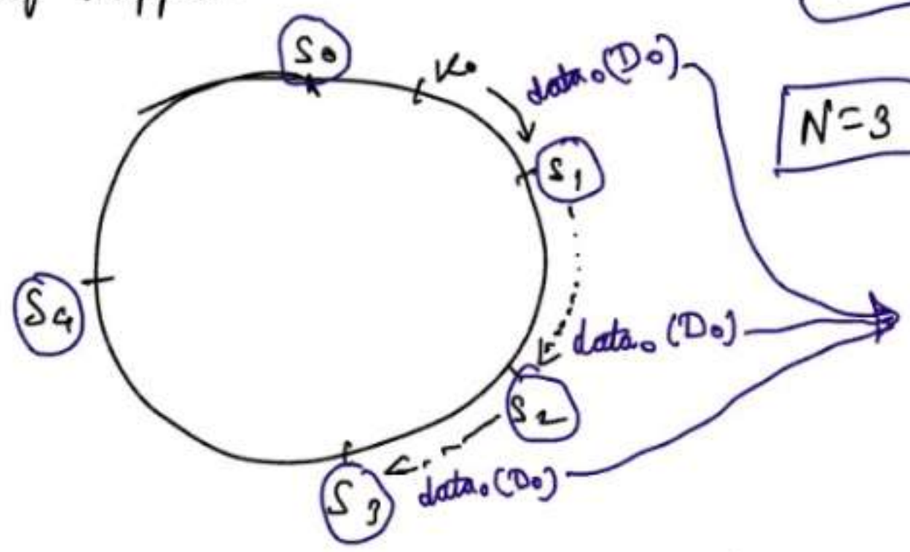
→ जब server add/Remove होता है तब हम automatic scaling कर सकते है
→ virtual node solved uneven distribution in Hash ring



* Data Replication →

→ For high availability & reliability, data must be replicated to (N) servers.
↓
configurable hai as per our need

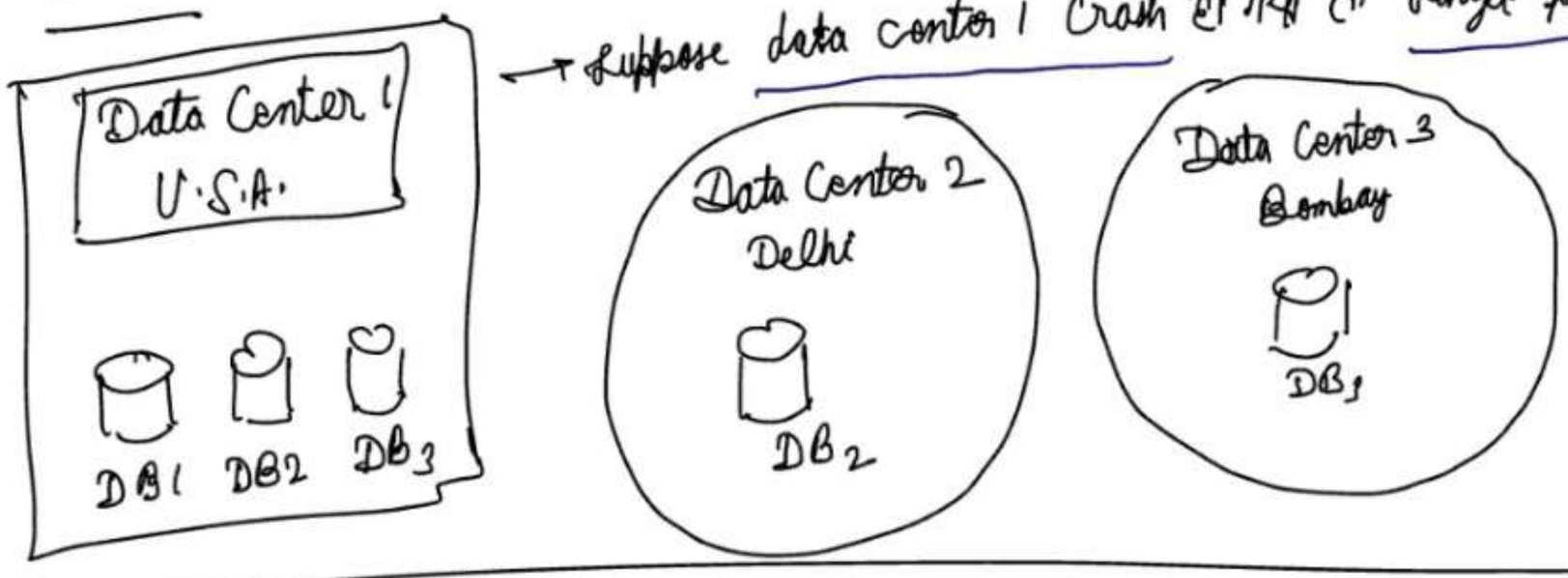
→ Key mapped → walk clockwise → replicate to N servers
↳ Replicate karna, mtlb backup banana



हमने Data0 (D0) को (S_1, S_2 S_3) में Replicate कर दिया
→ multiple copies bana dia hume.

→ Replicate करो तो अलग अलग data center (different location) में करो।

क्यों ? →



→ Suppose data center 1 crash हो गया तो single point of failure हो जायेगा

(30) Design Key-Value Store (Part 5) (Consistency, Quorums, Tunable Consistency) ⇒

→ Tunable Consistency

→ Consistency →
- → Data replicated है multiple nodes में
- → Consistency का ध्यान देना पड़ेगा ना
- → Synchronized होना चाहिये।

N: no of replicas
W: Write quorum of size W
R: Read quorum of size R

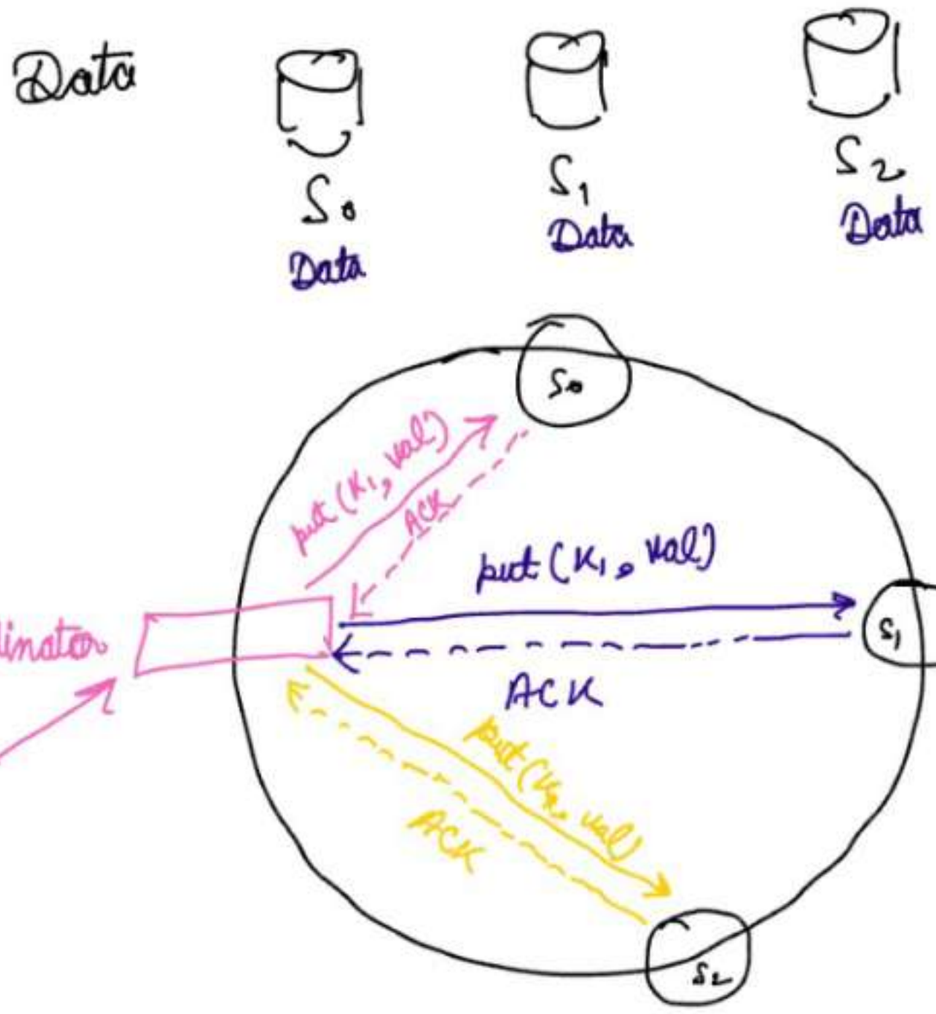
ये Quorum क्या है?
↳ एक तरह का Agreement की (W) no of ACK आना ही चाहिये।

"W": For write operation to be successful कम से कम ("W") replicas से ACK आना चाहिये for write operation
"R": For read operation to be successful कम से कम ("R") replicas से Response का wait करो।

Example से समझो :-

Lets take $N = 3$

We will replicate our "Data" in 3 servers (S_1, S_2, S_3)



$W = 1$ → सिर्फ S_1 से ACK आ गया तो इस write operation को successful मानेंगे

S_0 | S_1 | S_2 → Replica Nodes. We will not wait for S_1 & S_2 का ACK

Co-ordinator → ये एक तरह का Middle ware है b/w clients & Servers

The Configuration of (W) (R) & (N) is a typical tradeoff b/w (Consistency) & (Latency)

$W = 1$: write operation is returned quickly bcz बाकि replicas का wait नहीं करेगा

$R = 1$: Read operation is returned quickly bcz बाकि replicas का wait नहीं करेगा

Low Latency (quick response)	But Not good Consistency

W or $R > 1$ ⇒ Consistency तो अच्छा हो जायेगा But latency बढ़ेगा क्यों कि more replicas के Response का wait करना पड़ेगा.

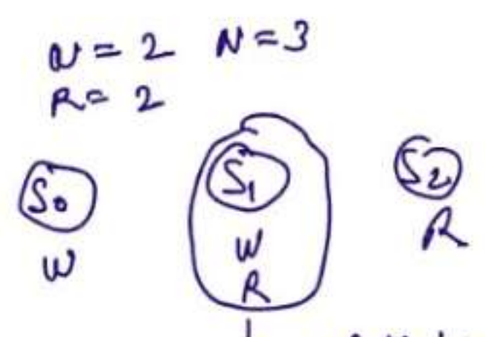
$(W + R) > N$ ⇒ Strong Consistency Gauranteed क्यों?

e.g → $N = 3$, $W = 2$, $R = 2$

$W + R > N$

$2 + 2 > 3$ ✓

→ कम से कम एक overlapping node तो मिल ही जायेगा



$W + R > N$ → इस case में कम से कम एक overlapping node जरूर मिलेगा जिसमें Read(R) भी हुआ है और WRITE(W) भी हुआ है

How to configure | N | W | R | to fit our use case.

* $R = 1$, $W = N$ } Fast Read
* $W = 1$, $R = N$ } Fast writes
* $W + R > N$ } Strong consistency gauranteed
  e.g → $W = R = 2$, $N = 3$

* $W + R \leq N$ } strong consistency not gaurantee

→ Depending on requirement you can tune W, R & N to achive your desired consistency.

↳ tunable consistency

Interviewer → Do you know about "Consistency Models" ?

Candidate → It says "~~bdtm~~" level of consistency है।

i.e. → Degree of data consistency.

* Strong Consistency → Client never sees out of date data. (updated data always)
कितना भी Read operation करो हमेशा updated data मिलेगा

* Weak Consistency → Subsequent read operation may not see latest data always.
There are high chances कि updated data नहीं मिलेगा

* Eventual Consistency → Given enough time, all updates propagated to replicas & they are updated.
थोड़ा सा time दोगे, It will replicate updated data to all server (node).

Interviewer → Do you think "Strong Consistency" is good ?

Candidate → "Strong Consistency" के लिये we will have to block read/write untill current write is updated to all replicas. → latency ↑ होगी

Not good for highly available system.

Interviewer → What will you use for "Key-Value" store ?

Candidate → We should use "Eventual Consistency"

C.g. → Dynamo & Cassandra also adopted Eventual Consistency.

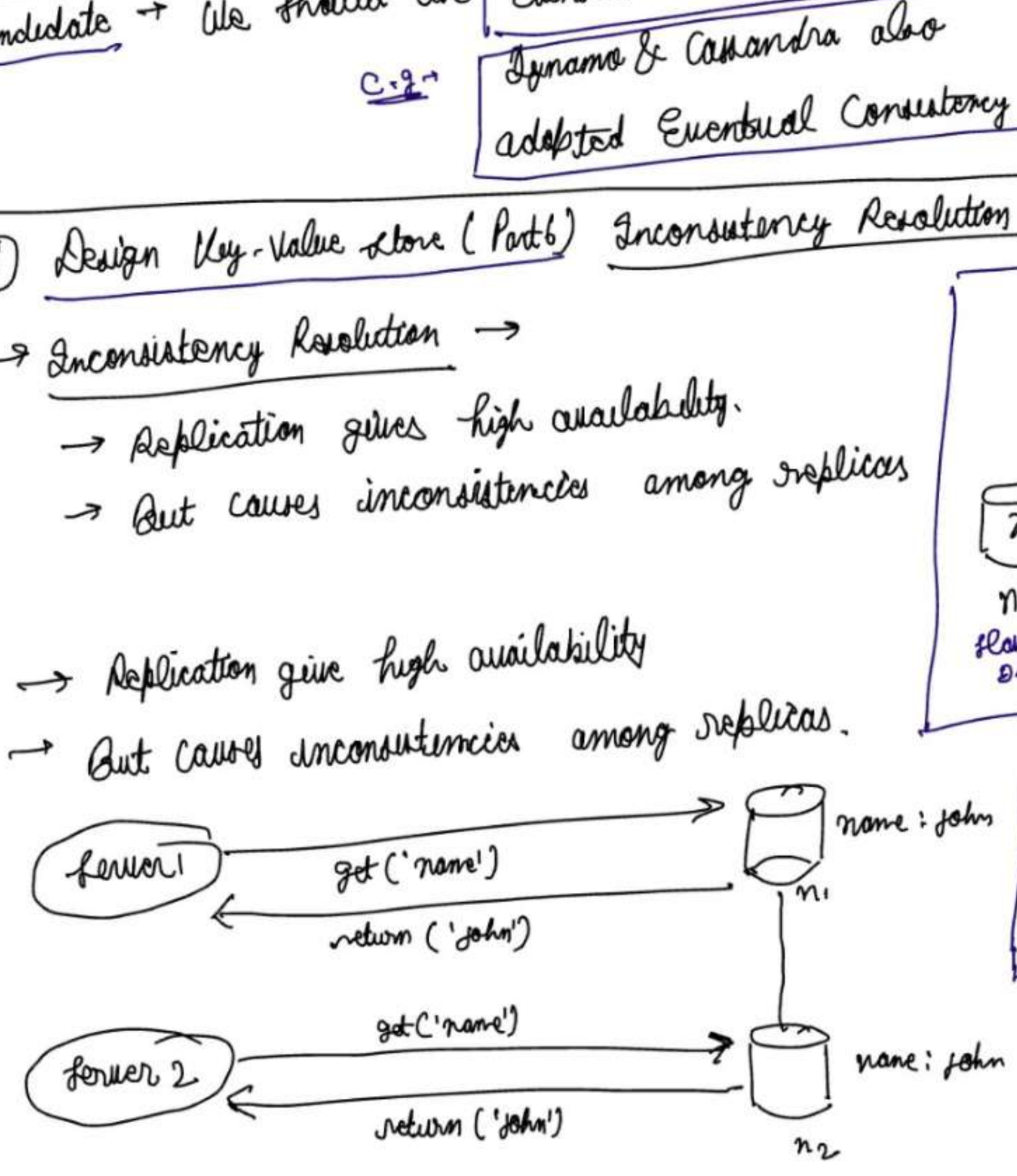
---

(31) Design Key-Value store (Part6) Inconsistency Resolution & Vector clocks →

→ Inconsistency Resolution →

→ Replication gives high availability.

→ But causes inconsistencies among replicas



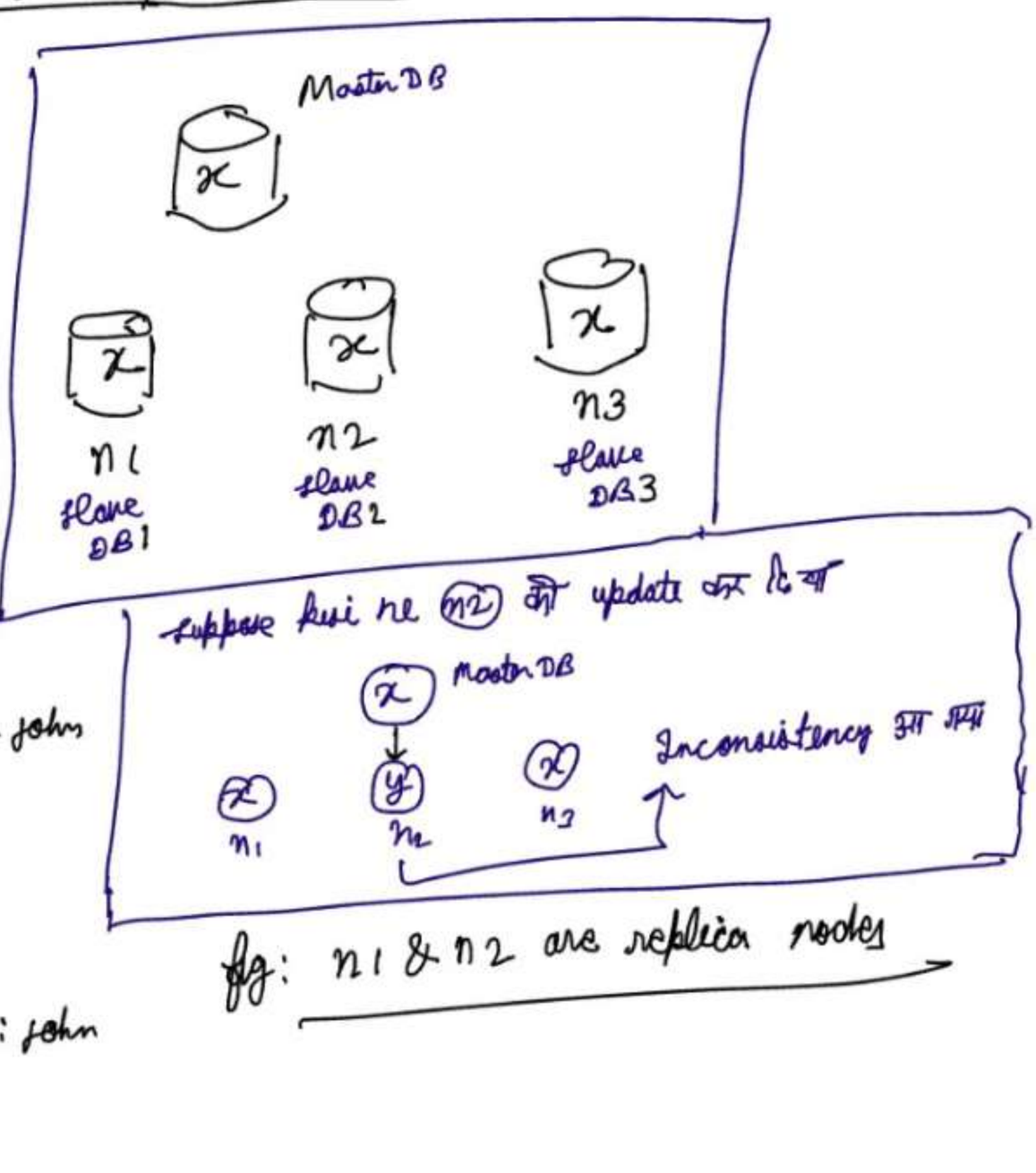


fig: n1 & n2 are replica nodes

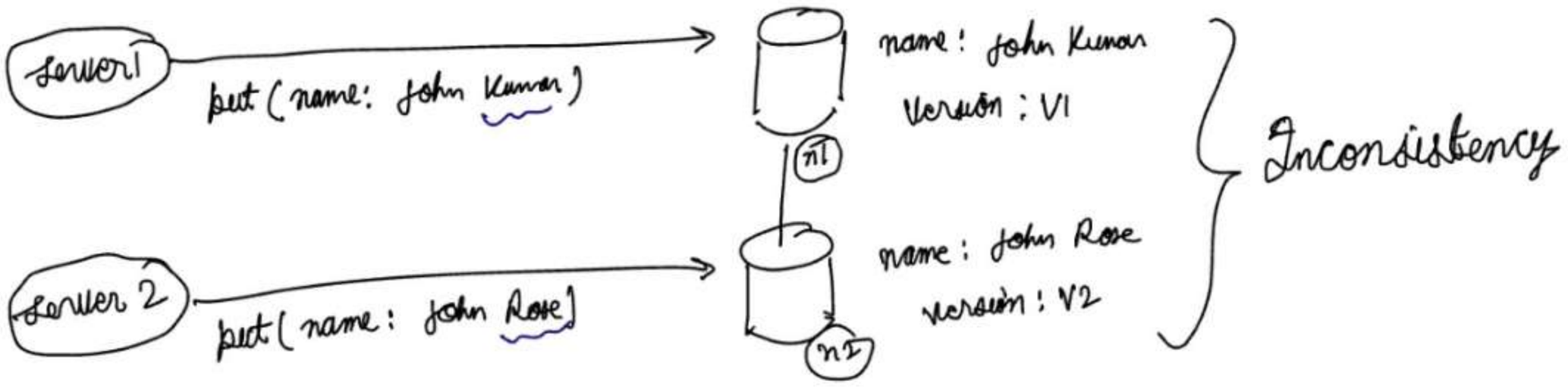


fig: n1 & n2 are replica nodes

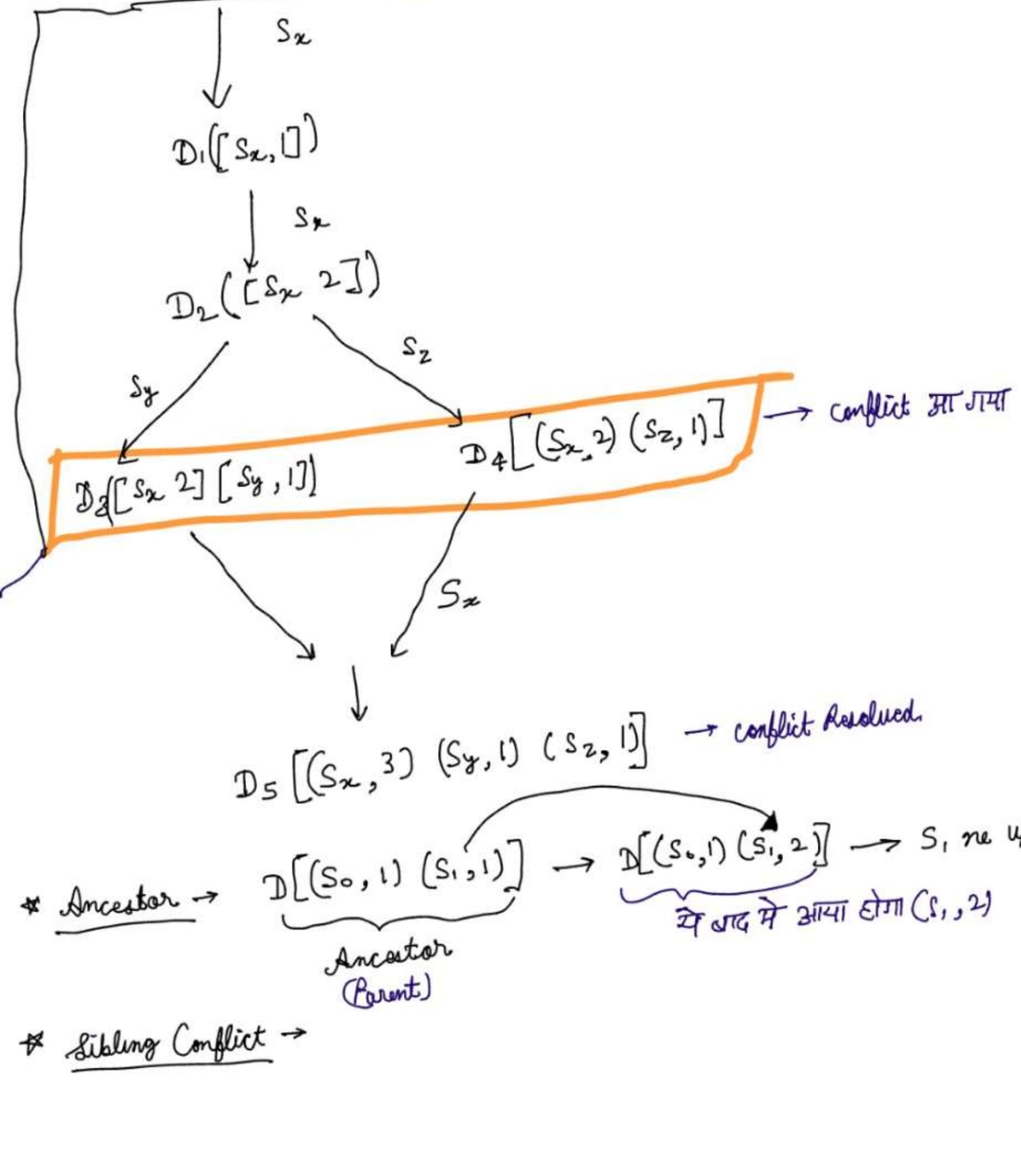
→ conflict आ गया

→ somehow अगर conflict resolve कर पाय by knowing past versions.

Vector clock → To Resolve conflict.

A pair [server, version]
associated with a data item.

$D[S_x, 1] \xrightarrow[\text{update}]{} D_1[S_x, 2]$



* Ancestor → $D[(S_0, 1)\ (S_1, 1)] \rightarrow D[(S_0, 1)\ (S_1, 2)]$ → S_1 ne update ~~किया~~ होगा $(S_1, 1) \rightarrow (S_1, 2)$

Ancestor (Parent)

ये बाद में आया होगा $(S_1, 2)$

* Sibling Conflict →

Disadvantage →

→ Complexity → Client needs to implement conflict Resolution logic.

→ [Server, Version] pairs grow rapidly

→ Threshold set कर लेंगे and limit exceed करे तो oldest pair को remove कर लेंगे.

But वही है कि resolution में inefficiency आ सकती है।

But according to Dynamo Paper, Amazon has not faced such problem yet in production

---

(32) Design Key - Value store (Part 7) : (Handling Temporary failures) (Gossip protocol) (Hinted) ⇒

Handling Failure → Temporary
→ Permanent.

→ Handling failures →

large system → failures is common

* Failure Detection → In a Distributed System

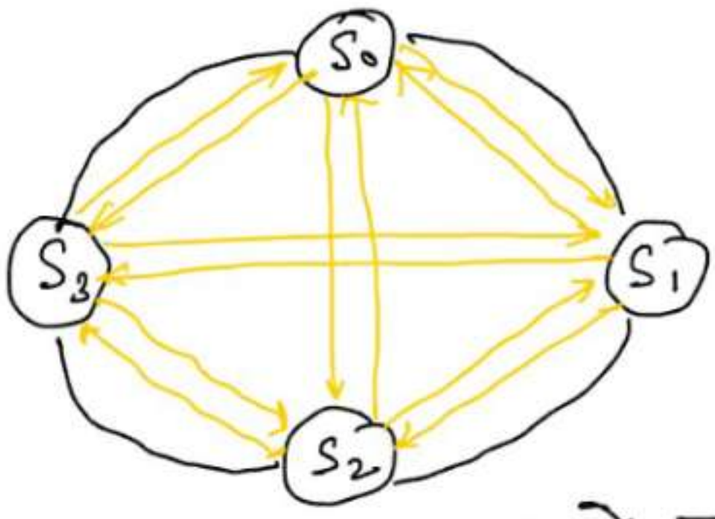
Server 1 — down है कि नहीं ??

Server 2

Server 3

Sabko, sabke baare mein pata ho.

km se km 2 log batayenge ki server down hai main tabhi yakin karunga.



All to All Multicasting (सबको, सबके बारे में पता हो)

* Gossip Protocol →

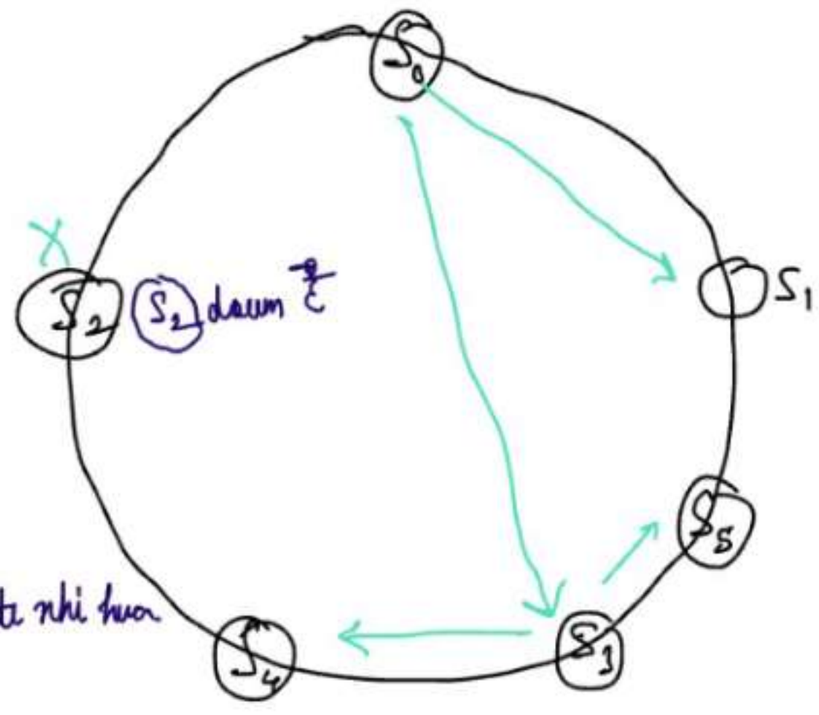
→ Every Node has a membership list [MemberID, Heart Beat Counter, Time]

Member ID	Heart Beat Counter	Time
0	10232	12:00:01
1	10224	12:00:10
2	9000	11:58:02
3	10237	12:00:02
4	10234	12:00:34

S0's Member list.

offline
HeartBeat काफी time से update नहीं हुआ है।
→ S2 down हो चुका है।

S0 ने notice किया कि
S1 Bahut der se update nhi hua



Handling Failures ⇒

@ Temporary Failures →

↳ Strict Quorum → Read/write blocked.

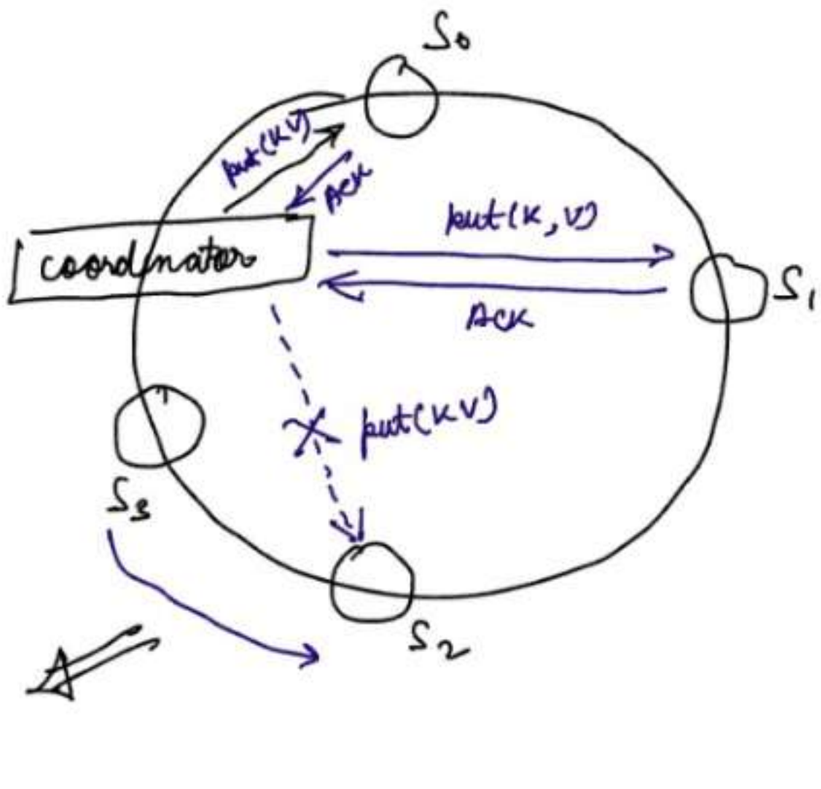
↳ Sloppy Quorum → Used to improve availability (थोड़ी सी छूट है)

Hash Ring ↳ First W servers to write → पहले W server में write कर दो

↳ First R servers to Read. → पहले R server से Read allow कर दो

When server is down, some other server will process request temporaly.

Hinted Handoff } Once it is up, changes will be pushed back to achive consistency.



since S_2 is down
S_3 data push_back करेगा
S_2 को जब S_2 up होगा

Hinted Handoff

---

(33) Design Key-Value Store (Part 8) →

Handling Permanent Failure | Anti-Entropy | Merkle Tree ⇒

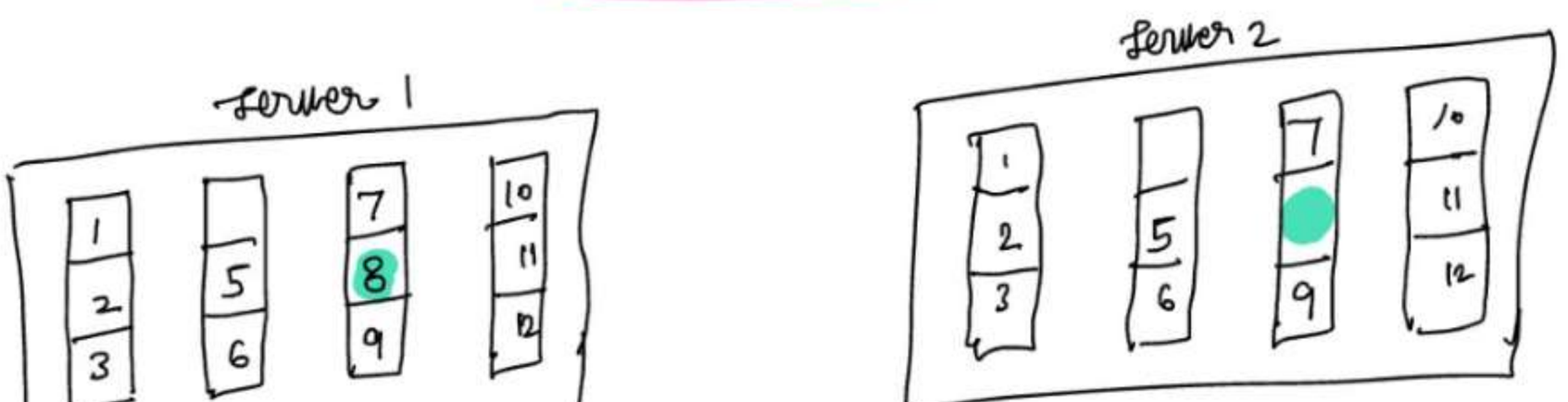
* Permant Failure ⇒

→ we implement "Anti Entropy" protocol to keep replica in sync.

→ Anti-Entropy →

↳ compare each piece of data on replicas and upding them.

↳ We use "MERKLE TREE"



सारे Keys को Hash कर देते हैं।

Server 1

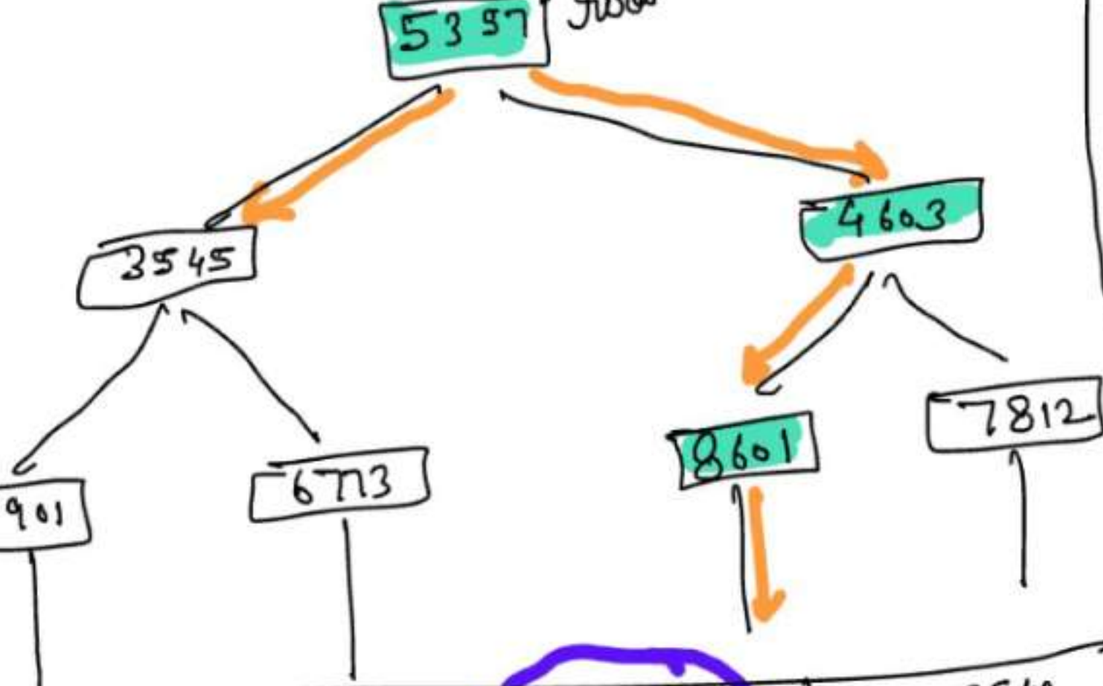
1 → 2343		7 → 9654	10 → 3542
2 → 1456	5 → 2145	8 → 1356	11 → 8705
3 → 9865	6 → 7456	9 → 4358	12 → 3697

Server 2

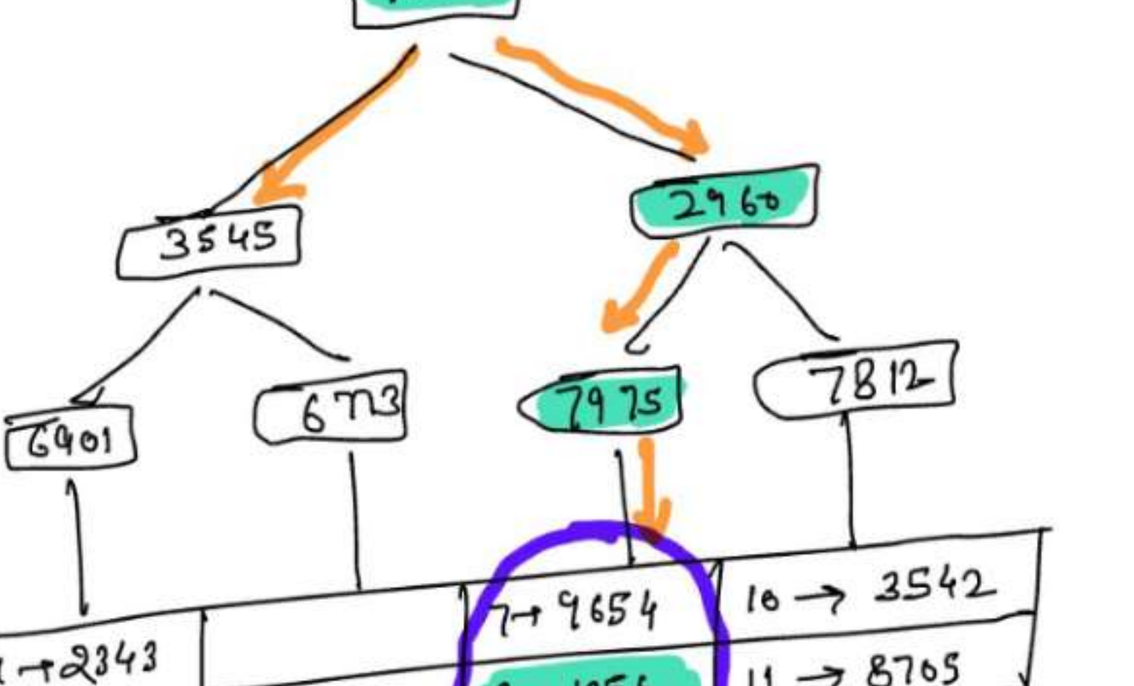
1 → 2343		7 → 9654	10 → 3542
2 → 1456	5 → 2145		11 → 8705
3 → 9865	6 → 7456	9 → 4358	12 → 3697

हर Bucket के लिये single Hash Node होगा

Server 1

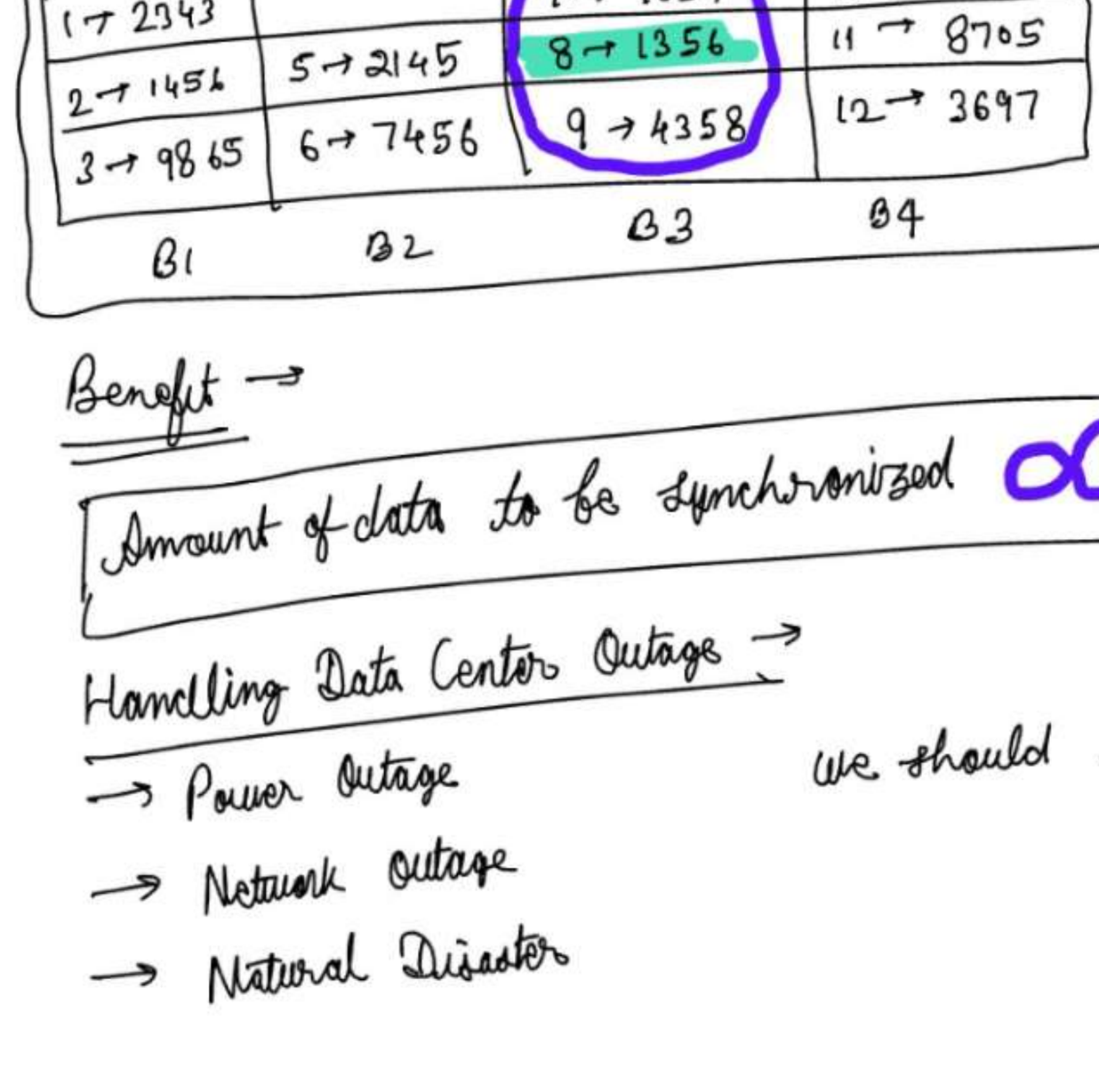


Server 2



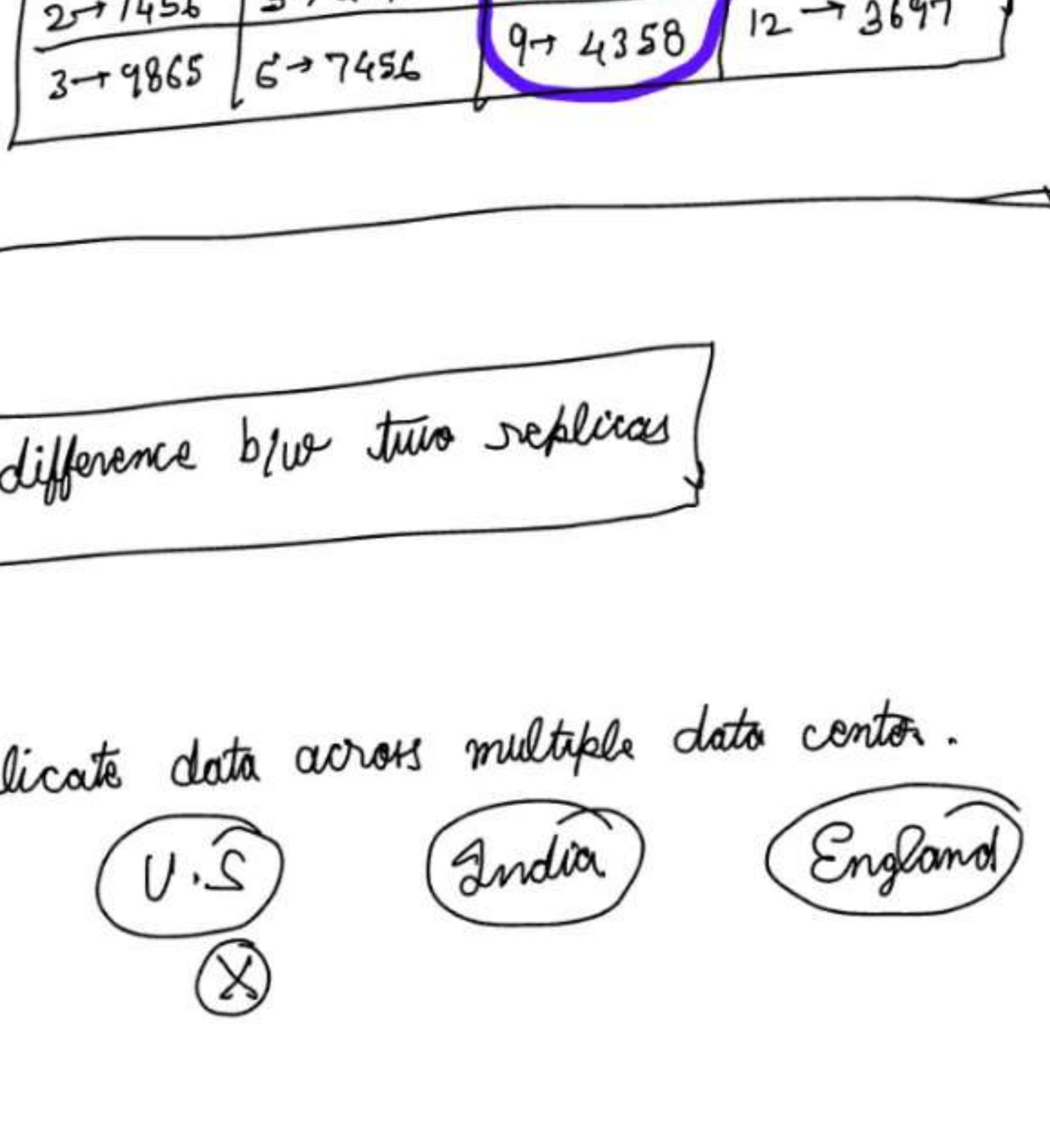
हम इसे Tree बनाते है।

Server 1



1 → 2343		7 → 9654	10 → 3542
2 → 1456	5 → 2145	8 → 1356	11 → 8705
3 → 9865	6 → 7456	9 → 4358	12 → 3697

Server 2



1 → 2343		7 → 9654	10 → 3542
2 → 1456	5 → 2145	8 → 1356	11 → 8705
3 → 9865	6 → 7456	9 → 4358	12 → 3697

Benefit →

Amount of data to be synchronized $\propto$ difference b/w two replicas

Handling Data Center Outage →

→ Power Outage
→ Network Outage
→ Natural Disaster

we should replicate data across multiple data center.

U.S (X) India England

(34) Design Key Value Store (Part 9) →

System Architecture Diagram →

→ Client ⇄ (get(key) / put(key, value)) Key Value Store

→ co-ordinator node : proxy b/w client & Key-Value store.

→ Nodes distributed on a ring → Consistent Hashing.

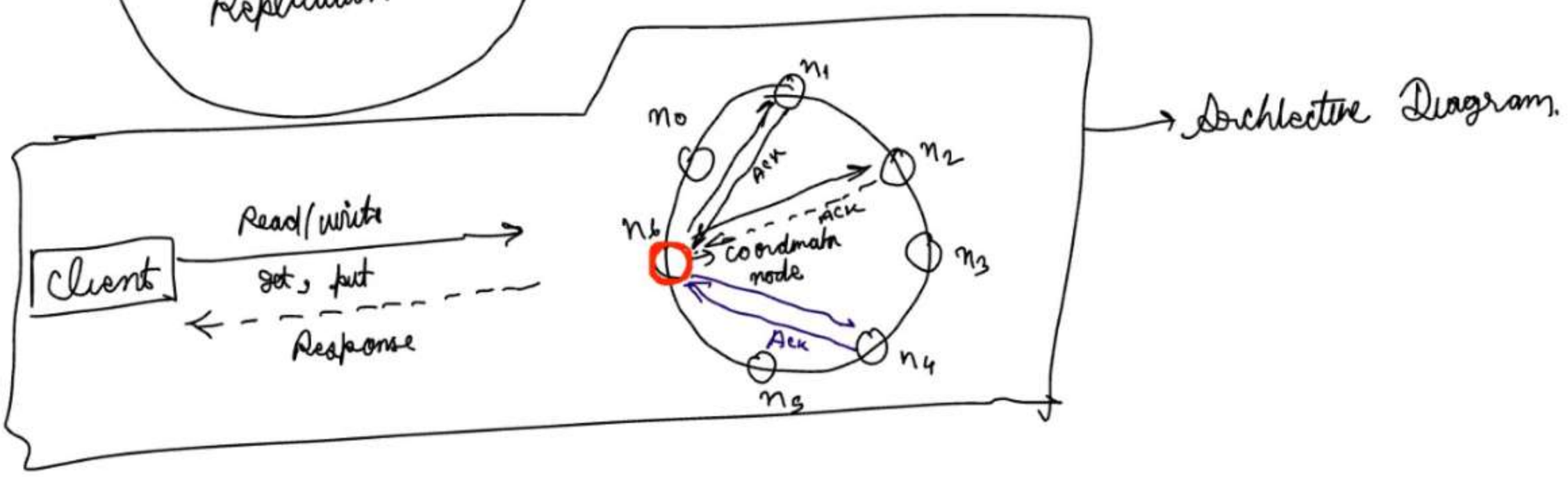
→ Add / Remove Node → Automatic.

→ Data Replicated in Replicas

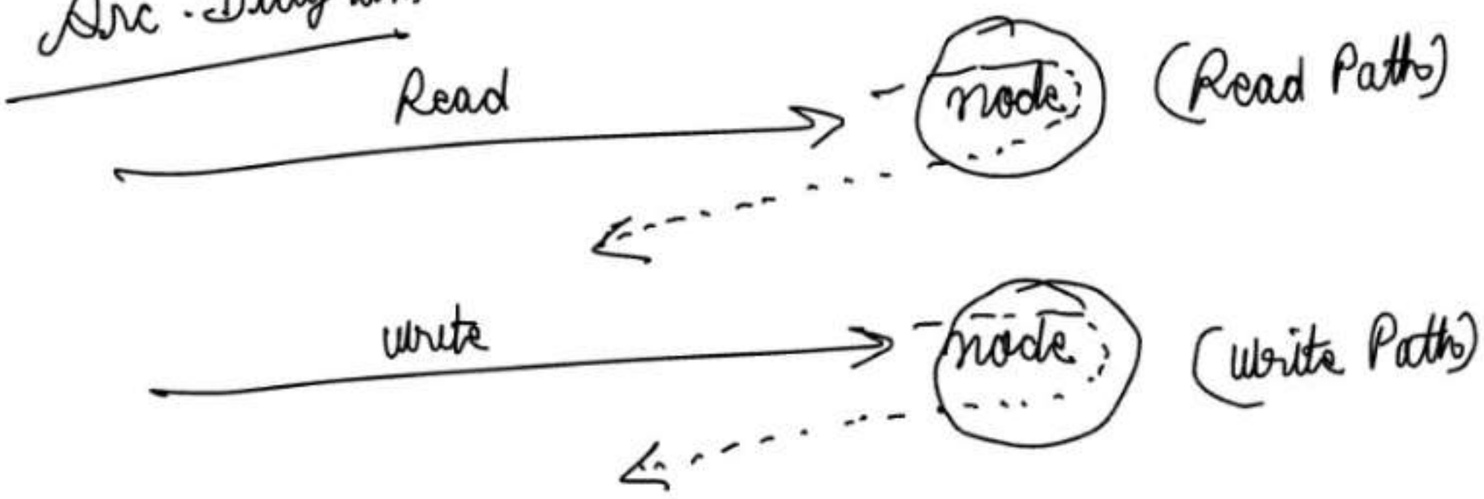
→ No single point of failure — Multiple Nodes

Node & Responsibilities

- Client API
- Failure detection
- Failure Handling
- Conflict Resolution
- Replication



Arc. Diagram →

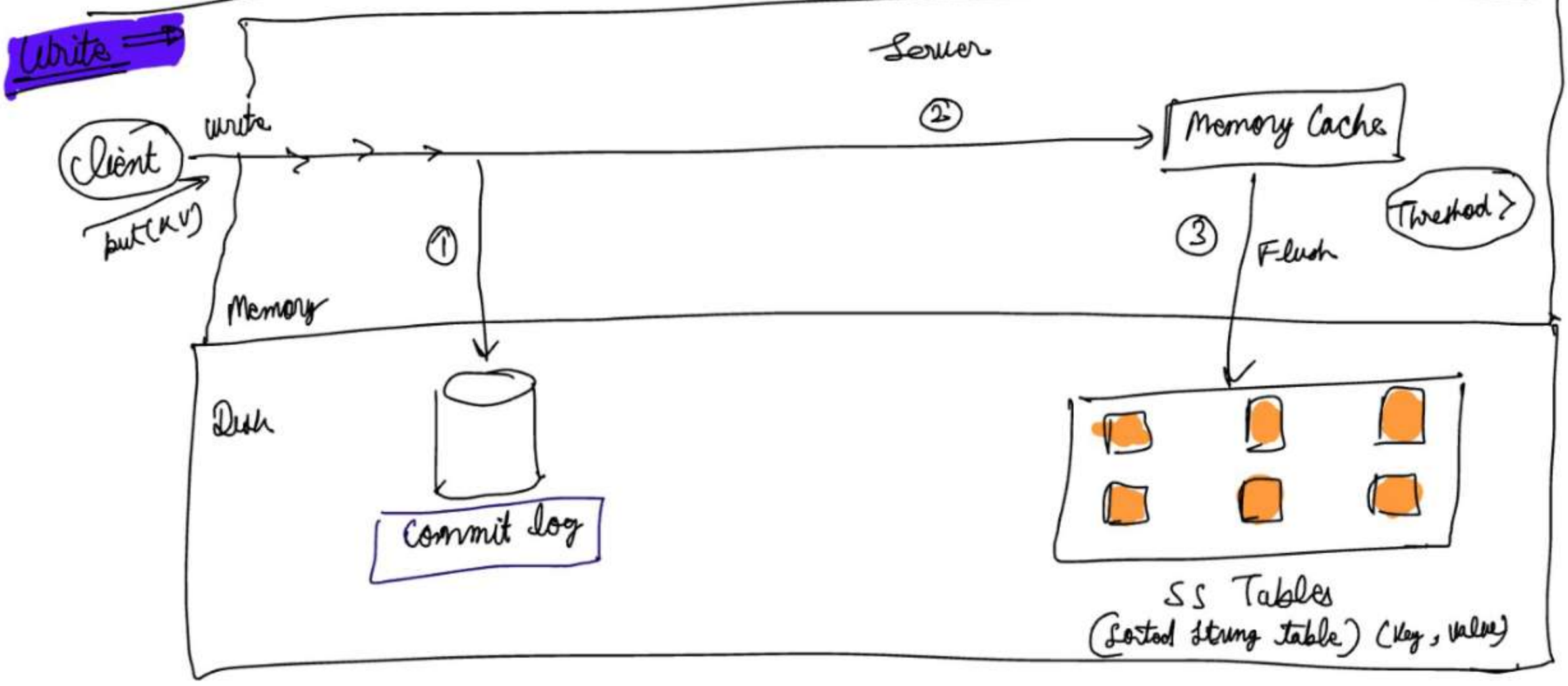


Real life Example → Cassandra.

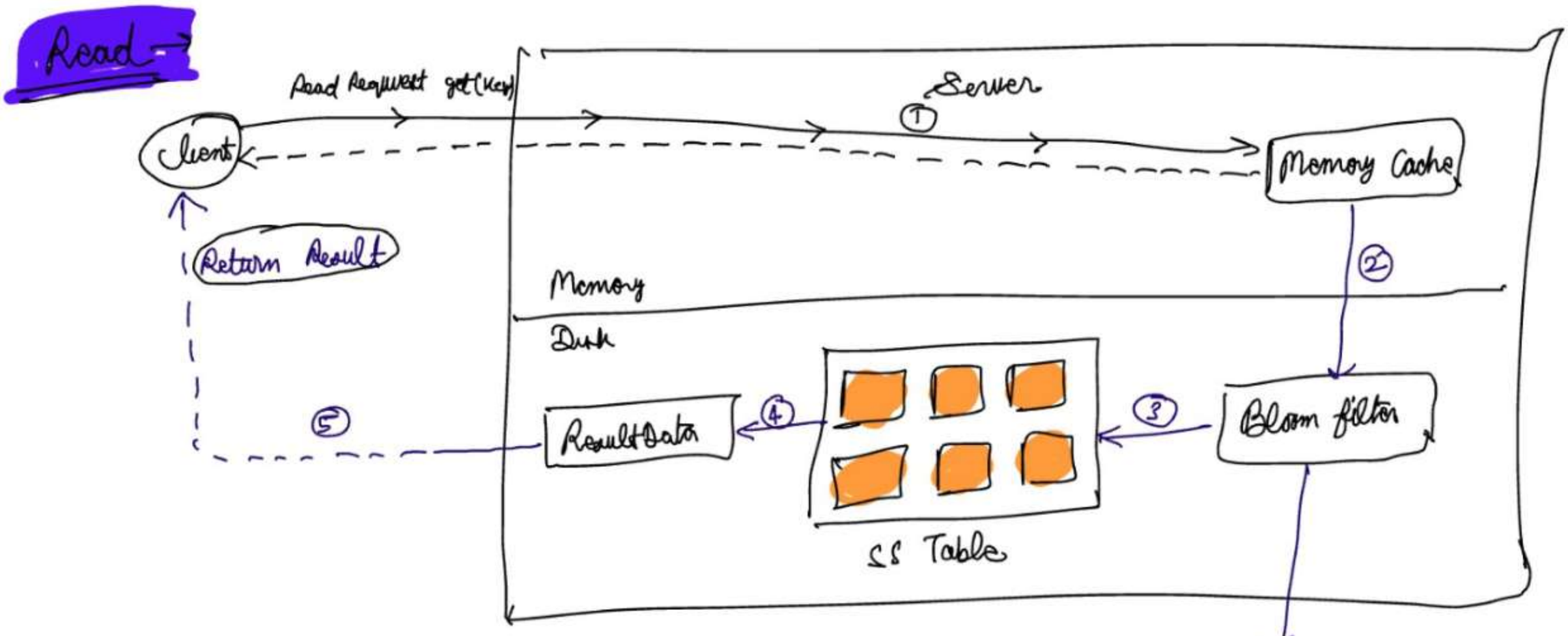
35 Design Key Value Store (Part 10) →

Read Path Write Path → Similar to Cassandra.

Write



Read



Goal/problems	Technique
Ability to store big data	Use consistent Hashing to spread the load across servers
High availability reads	Data replication Multi-data centre setup
High available writes	Versioning and conflict resolution with vector clocks
Data set partition	Consistent Hashing
Incremental Scalability	Consistent Hashing
Heterogeneity	Consistent Hashing
Tunable consistency	quorum consensus
Handling temporary failures	Sloppy quorum & hinted handoff
Handling permanent failures	merkle tree
Handling data center outage	Cross-data centre replication

(36) Design a unique ID Generator in distributed System (Part 1) →

Understand the problem →

First Thought → DB → Table → Primary key with "auto increment" attribute

→ But इसमें problem है

→ Single database is not large enough

→ multiple database में unique_id generate करना challenging है।

example →

user_id
1 3 2 3 4 8 6 2 9 1
1 4 3 9 2 1 8 6 3 2
1 5 0 0 1 9 2 0 3 4

Step ① Understand the problem and Establish design scope

Candidate → what are the characteristics of Unique_IDs?

Interviewer → IDs must be unique and sortable.

Candidate → For each record, does ID increment by 1 ?

Interviewer → Increment तो करेगा time के साथ but not necessarily by 1.

(Evening ID) > (morning_ID)

Candidate → Do IDs contains only numerical values?

Interviewer → Yes

Candidate → ID के length की क्या requirement है?

Interviewer → IDs should be fit in 64-bit.

Candidate → What is the scale of the system ?

Interviewer → System should be able to generate 10 000 IDs per second

Requirement →

→ IDs must be unique
→ Numerical values only
→ 64 bit
→ ordered by date.
→ 10 000 unique IDs / sec

(37) Design a unique ID generator in Distributed Systems (Part 2) Multi Master Replication →

Step (2) Propose High level design and Get Buy In →

Multiple options can be used to generate unique IDs in Distributed Systems

→ (i) Multi Master Replication.
(ii) UUID (Universally unique IDentifier)
(iii) Ticket Server
(iv) Twitter snowflake Approach (popular)

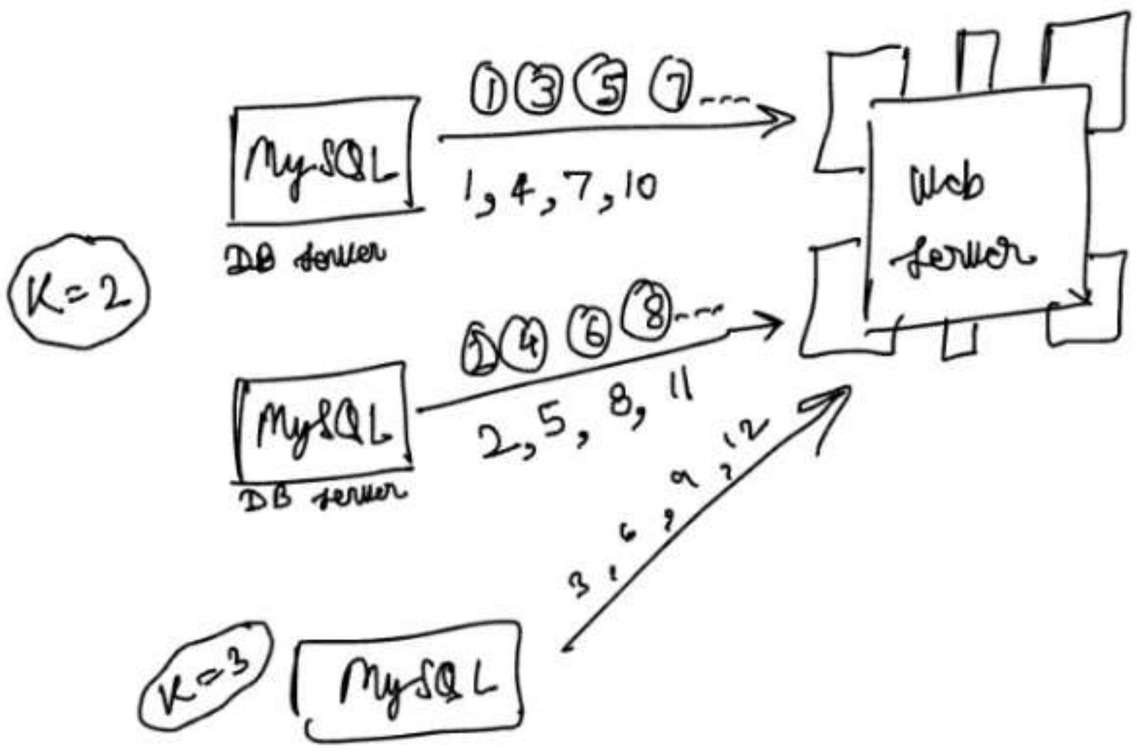
(i) Multi Master Replication →

→ It uses DataBase (auto-increment) feature.

→ ID को 1 से नहीं increment करेंगे, it increments by 'K'.

K = no of DB servers in use.

→ Lets see a diagram with K=2.

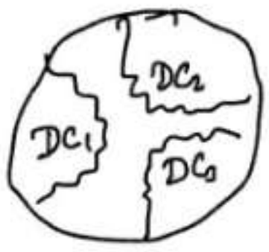


This solves some scalability issue कैसे ?

Major Drawbacks

→ How to scale with multiple Data center

→ When a server is added OR Removed
क्या करोगे कैसे करोगे ?
(will not scale well)



multiple data centre mein scaling difficult ho jata hai

(38) Design a Unique ID Generator in Distributed System (Part 3) : UUID →

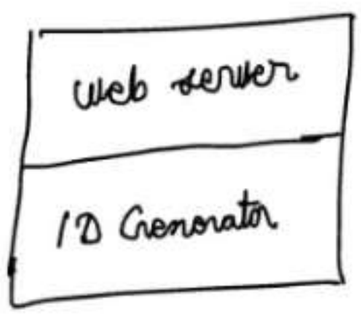
UUID →

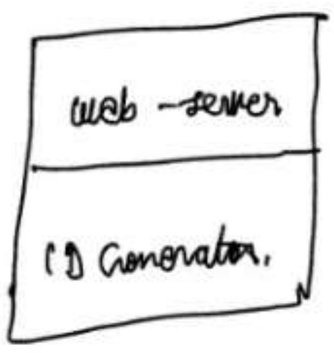
→ Easy way to obtain Unique IDs.
→ 128-bit number.
→ Very low probability of getting collisions.
Fact : 1 billion UUIDs/second for 100 years probability of single collision will be 50%
→ Can be generated independently without co-ordinating with servers.

eg →

123e4567-e89b-12d3-a456-426614174000 → numeric भी है non-numeric भी है।

Diagram →





Pros :-

→ Generation is simple.
→ No co-ordination within servers required.
→ So, no synchronization issues.
→ Easy to scale.

Cons :-

→ 128 bits long. Our requirement was 64-bits.
→ IDs could be non-numeric.

---

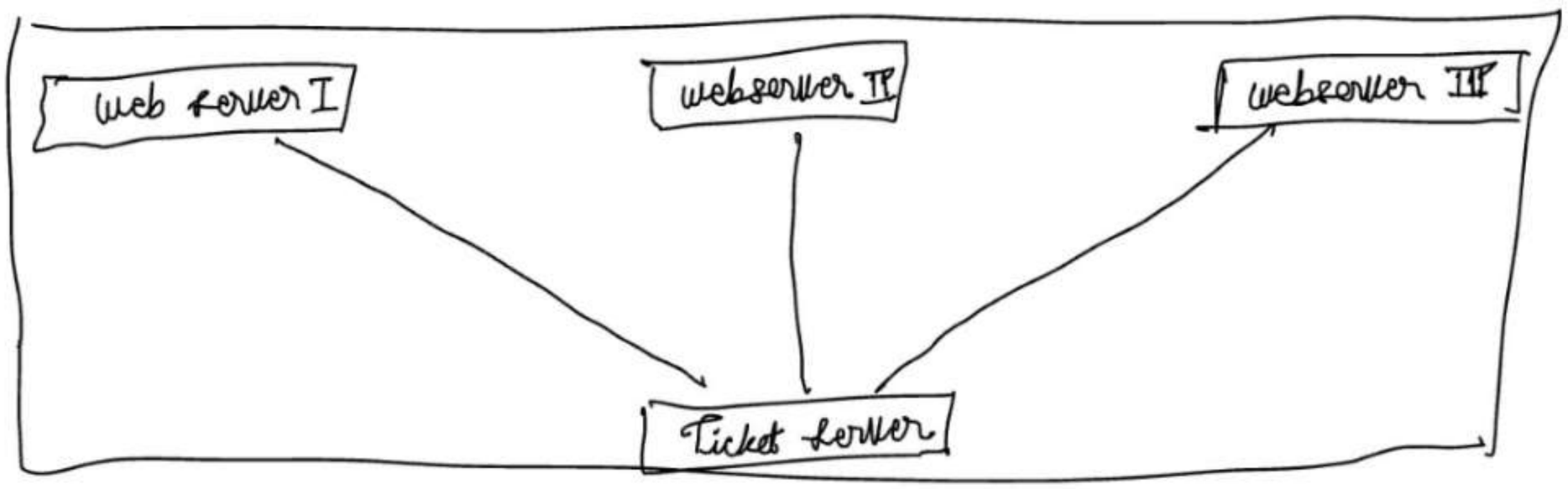
(39) Design a Unique ID Generator in Distributed System (part 6) →

Ticket Server →

☆ Ticket Server →

→ Developed by "FLICKER"
→ To generate Distributed Primary keys.
→ कैसे क्या है इसका Idea?
→ Use a centralized auto increment feature in a single database server (Ticket Server)

Lets see this with a Diagram →



Pros →

→ Numeric IDs

→ Easy to Implement

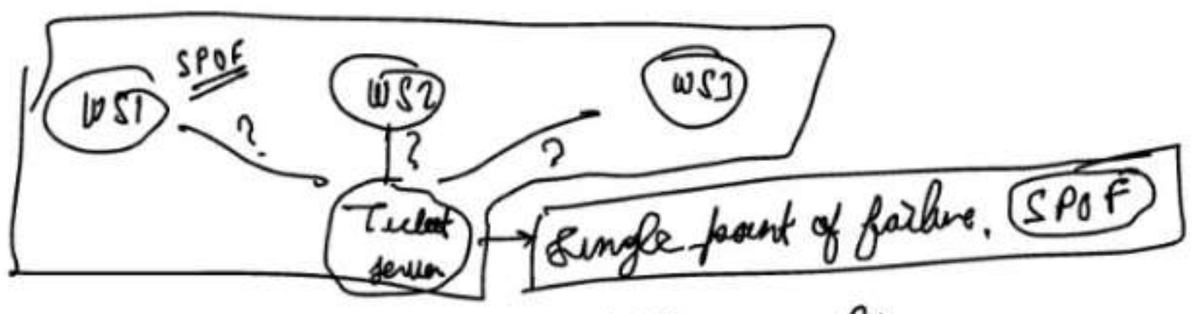
→ works for small to medium scale applications.

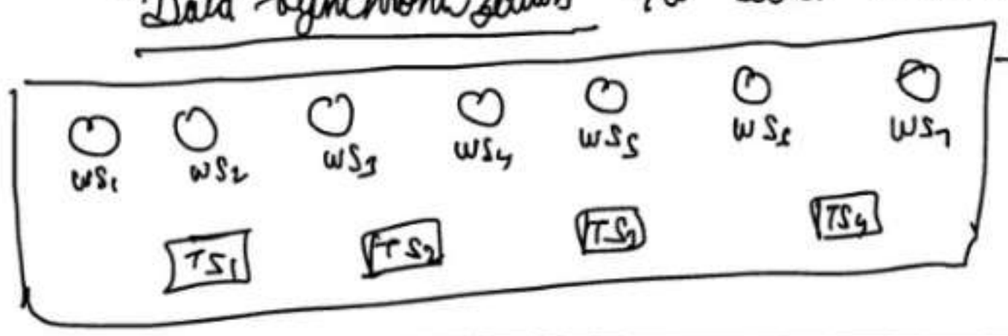
Cons :-

→ Single point of failure - Ticket Server.

→ To avoid SPOF (single point of failure)

→ we can setup multiple ticket server. But this will introduce challenges like 'Data Synchronization' b/w Ticket Servers.





→ yahaan par SPOF se bachne ke liye ek se zyada ticket server laga lia hai lekin iss wale approach me Data Synchronization ka issue aayega.

---

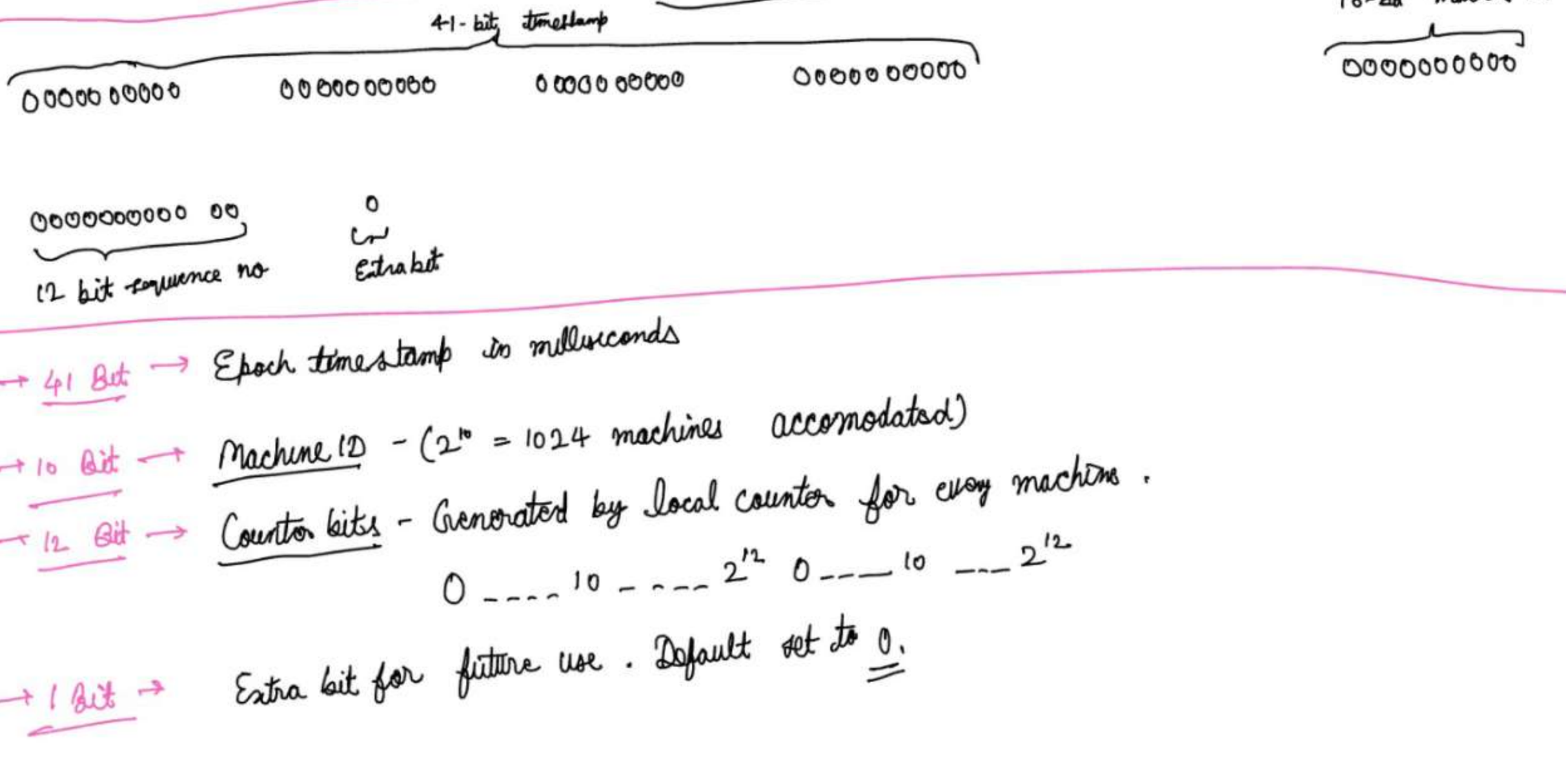
(40) Design a Unique ID Generator In Distributed System →

Part ⑤: Twitter snow-flake Approach →

* Twitter Snowflake Approach →

→ Developed by Twitter

→ Optimized concept of UUID → Keeping IDs sortable; ID size 64 bit.



→ 41 Bit → Epoch timestamp in milliseconds

→ 10 Bit → Machine ID - (2^{10} = 1024 machines accomodated)

→ 12 Bit → Counter bits - Generated by local counter for every machine.

$0 \_\_\_\_ 10 \_\_\_\_ 2^{12}$ $0 \_\_\_ 10 \_\_\_ 2^{12}$

→ 1 Bit → Extra bit for future use. Default set to 0.

* Advantages of Twitter flake ⇒

→ Sortable by time → They have a timestamp in the beginning.

→ 64-bit → Less storage size requirement.

→ Scalable → 1024 machine

→ Highly available → 1024 machine & each machine generate 2^{12} (4096) unique IDs each millisecond.

* Disadvantages of Twitter Snowflake →

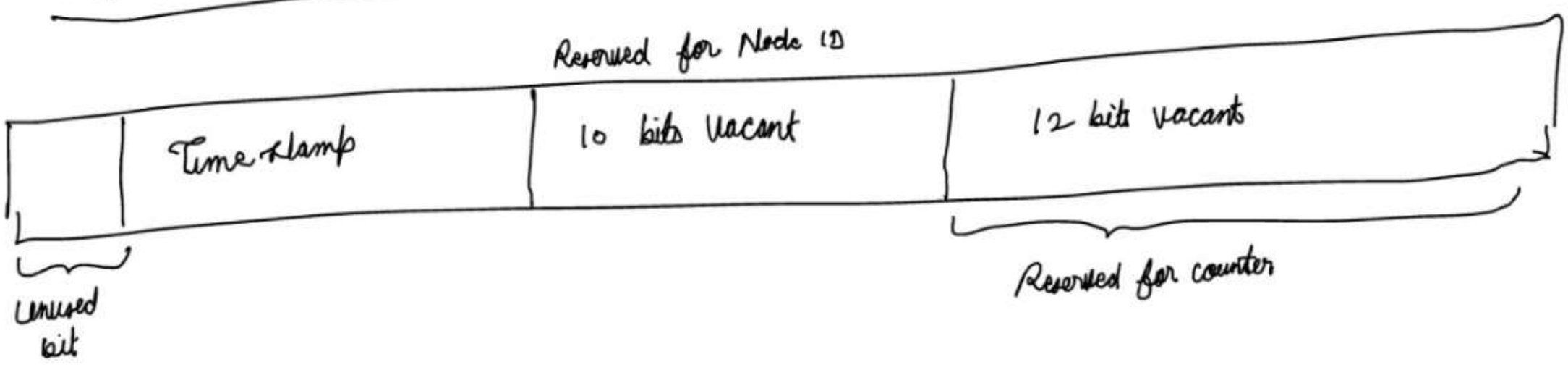
→ We have to maintain the machine IDs

→ Generated IDs are not random like UU IDs

↳ Predictable → Security issues.

(Q1) Design a Unique ID Generator in Distributed System →

Part 6: Design Deep Dills →



* Node ID → Choosen at the startup time.
(10 Bits) Generally fixed when the system is running. Use MAC Address — unique for every machine.

* Timestamp (41 - bits)

↳ As timestamp grow with time, IDs are sortable by time.

↳ time stamp (binary) w.r.t epoch.

↓

Decimal

↓

to millisecond.

↓

to UTC time

eg→ Aug 09 2022 17:34:31 UTC.

* Maximum timestamp with 41 bits →

$2^{41} - 1 = 2199023255551$ (ms)

$\approx$ 69 years

So after 69 years we will need new epoch time OR some other technique to migrate IDs

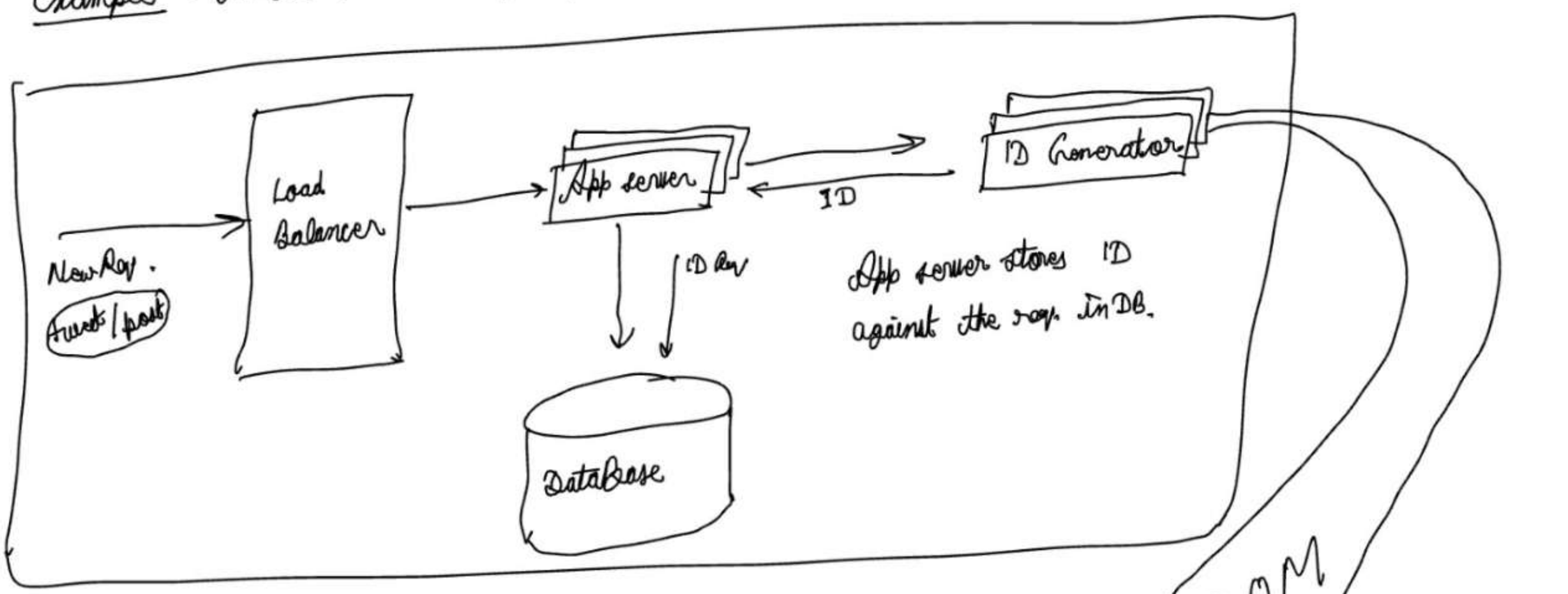
* Counter (12 Bits) → 12 Bits $= 2^{12} = 4096$ combination

$= 4096$ IDs per millisecond

---

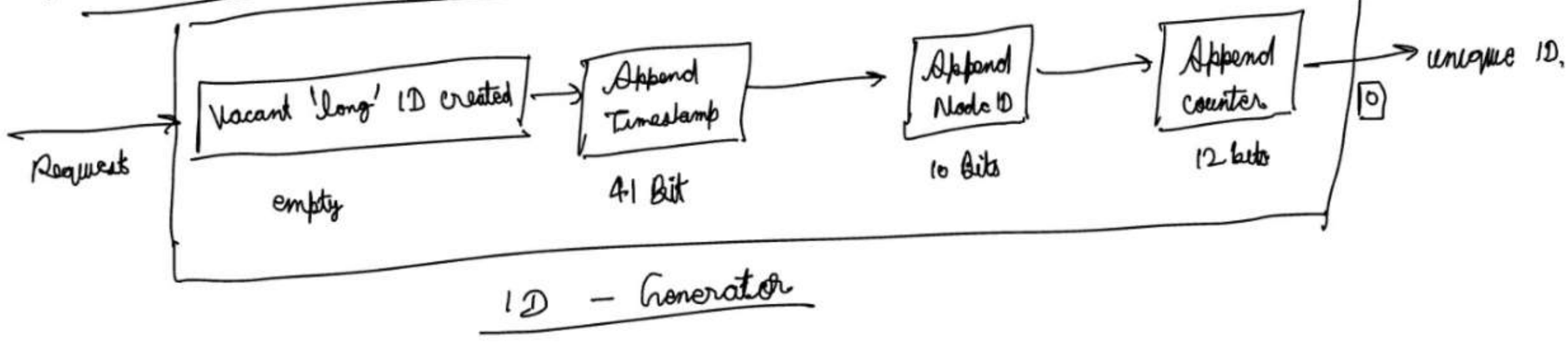
* System Diagram →

Example → A social networking application calls ID generator to assign unique IDs to each post.



ID Generator को Zoom करके देखते है ⇒

* ID Generator →



ID - Generator

(42) Design a Unique ID Generator in Distributed System (Part 2) (Wrap up) →

What we discussed so far →

→ Multi Master Replication

→ UUID

→ Ticket Server

→ Twitter Snowflake → settled for this

↳ Supports all our usecases

↳ Scalable in a distributed Environment

→ If you have extra time you can discuss few additional points :-

① Clock Synchronization → We assume ID generation servers has the same clock. Might not be true when a server is running on multiple cores.

"Network time protocol"

② Fewer Sequence numbers but more timestamp bits →

→ sounds like a good idea for long-term applications.

③ High availability → ID generator is a mission-critical system, so it must be highly available.

---

(43) Design a URL Shortner ⇒ (Uber, Microsoft)

Part ① : (Understand the problem & Establish Design Scope)

Step ① → Understand the problem & Establish Design Scope →

Candidate → Can you give an example how URL shortner works?

Interviewer → Assume URL.

https://www.interview_ds_algo.com/v=course&c=loggedin&V=V3&l=long → original URL.

your service should create an alias with shorter length.

Example → https://tinyURL.com/y7KeocWxyz → shortened URL

↓

If you click this it should redirect to original URL.

Candidate → What is the traffic volume?

Interviewer → 100 millions URL's are generated / day.

Candidate → How long is the shortened URL?

Interviewer → As short as possible

Candidate → what characters are allowed in shortened URL?

Interviewer → Can be a combination of
number: (0-9)
Character: (a-z) (A-Z)

Candidate → Can shortened URLs be updated or deleted?

Interviewer → For simplicity, assume that they cannot be updated or deleted.

Summary and Basic use Cases →

→ URL shortening → Long URL → Short URL

→ URL redirecting → ShortURL → Redirects to Long URL address

→ High availability → Scalable → fault Tolerance

Next: Back of the envelope estimations →

---

(44) Design a URL shortner →

Part (2): Back of the envelope estimation →

→ Write operation → 100 million URLs/day. generate hoga.

→ Write operation per second: 100 million/24/3600 = 1160 URLs (write)

→ Read operation → Assume that Users read 10 times the write operations.

Read	write
10	1

So, Read operations/second = 1160 * 10 = 11600 Read

→ कब तक चलेगी मेरी service ⇒

→ Assume → 10 years

→ So we must support

= 100 million * 365 * 10 = 365 billions record.

100 million: per day; 365: days in a year; 10: years

Assume average URL length = 100

Let, 1 byte for each character

So, total = 100 * 1 = 100 bytes per URL

100: URL length

→ Storage requirement for 10 years →

We have to support 365 billion URLs over 10 years

→ for URL = 100 bytes.

→ Total storage → (365 billion)*(100 bytes) = 365 TB.

10 years ≃ 365 TB

---

(45) Design a URL shortner (Part 3) →

Step (1) → Proposed a High level Design and Get Buy in !! ! →

① API end points →

② URL Redirection →

③ URL shortening flows →

① API end points → our service needs 2 API end points :-

(i) → URL को short करने के लिये

POST : api/v1/data/shorten

↳ Request Param :

{ long URL : <long URL &string> }

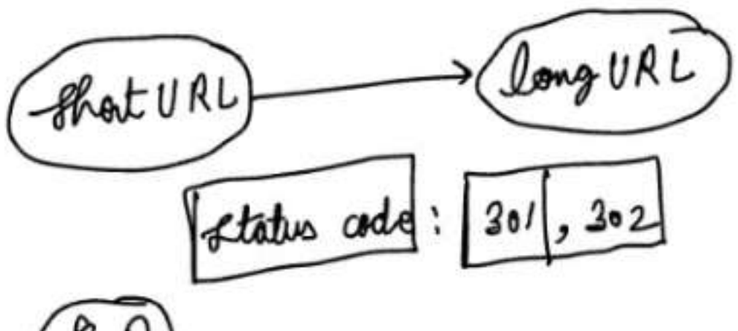
↳ Returns <short URL>

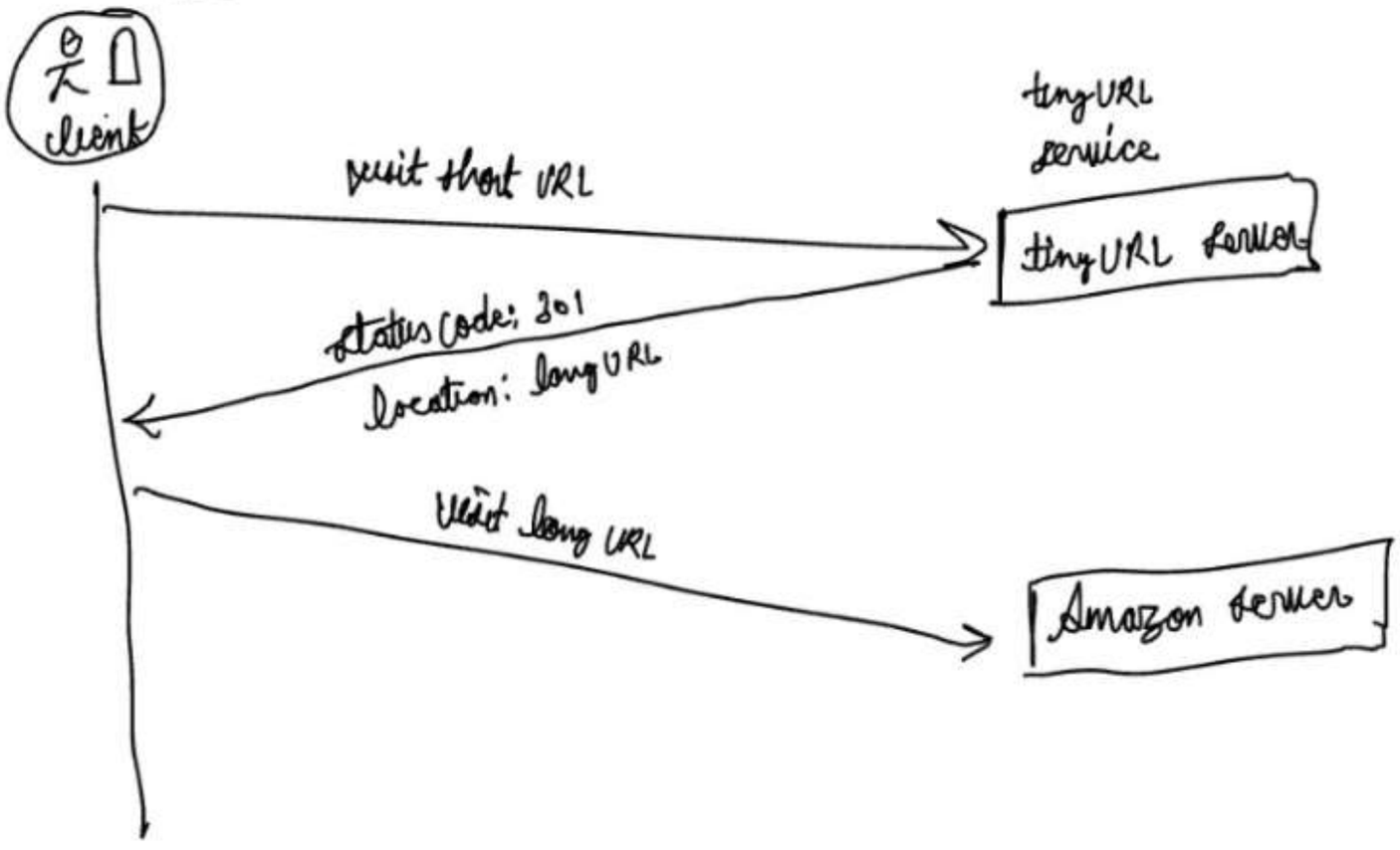
(ii) short URL को long URL में redirect करने के लिये ⇒

GET : api/v1/shortURL

↳ Returns long URL for redirection →

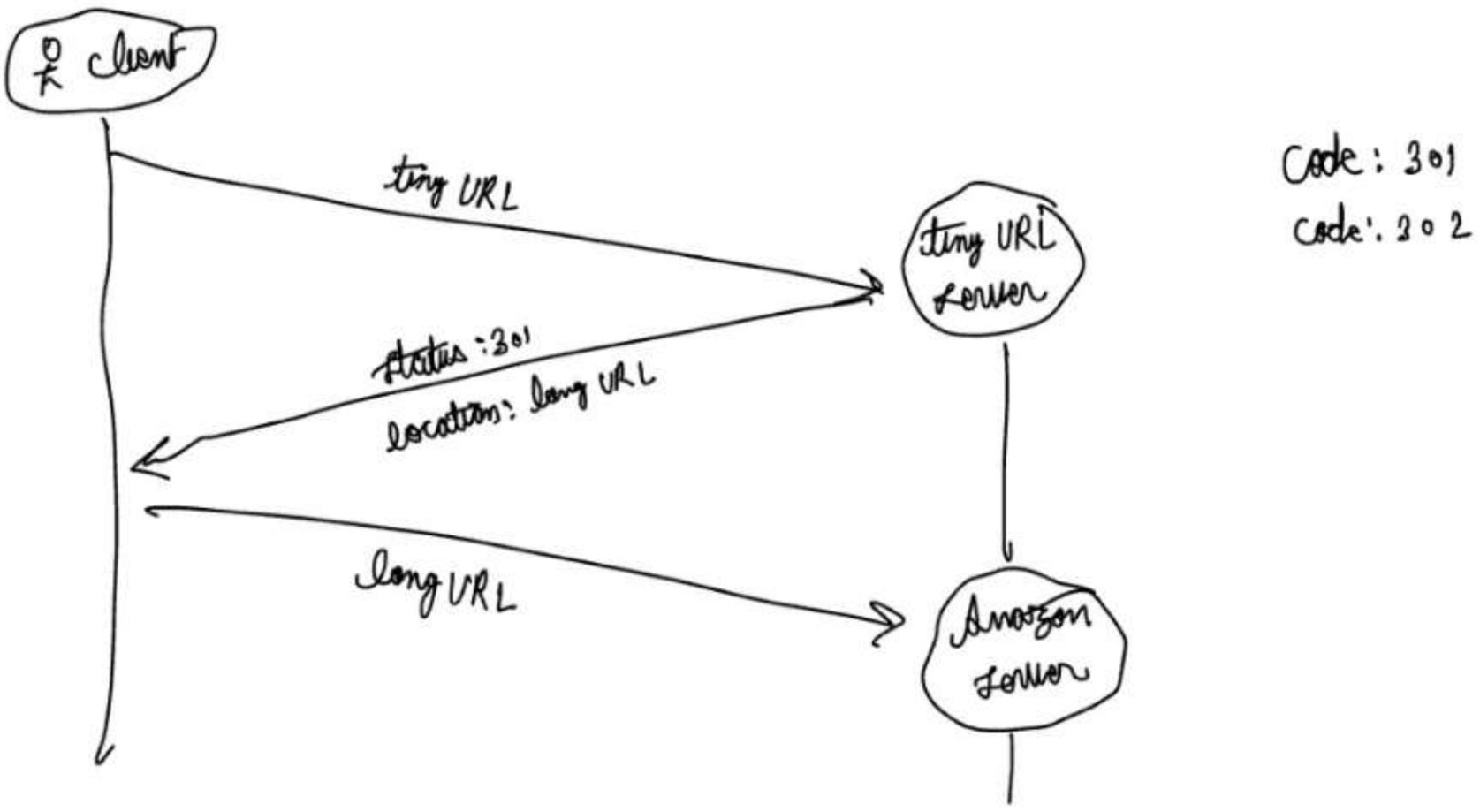
② URL Redirection ⇒ कैसे काम करता है Browser में ?





(46) Design a URL shortner →

Part ④ : ~~Step~~ ① 301 Redirect Vs 302 Redirect Status code ⇒



Code: 301
Code: 302

* 301 Redirect →
  - ↳ Requested URL (tiny URL) is permanently moved to long URL.
  - ↳ Browser caches the response
  - ↳ फिर से click करेगा अगर tinyURL में, तो (tinyURL service) के पास request नहीं जायेगा अब! अब direct long URL में ही redirect होगा cache से

* 302 Redirect →
  - ↳ Requested URL (tiny URL) is "Temporarily" moved to long URL.
  - ↳ फिर से click करेगा अगर tiny URL में तो फिर से tinyURL service के पास request जायेगी.

Pros and Cons of (301/302) redirect ⇒ कब कौन सा use करेंगे? ⇒

301 → cache
302 → no cache

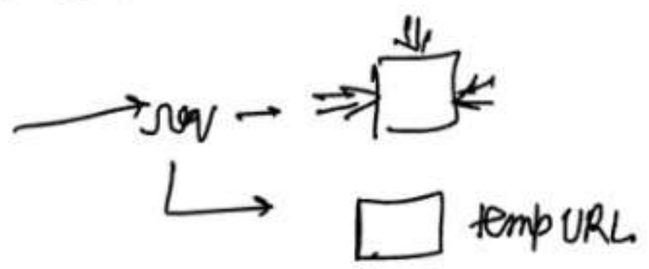
tinyURL server
↓
Load कम हो
↓
तुम चाहोगे की उसपर request कम आये
↓
so 301 sounds good

---

* temporarily तुम्हे किसी और page में redirect कराना है
↓
(302) sounds good

→ also analytics के लिये अच्छा है।
  - ↳ track click Rate
  - ↳ Source of click.

{ suppose your website is undermaintance, then u want to redirect other URL }



(47) Design a URL shortner →

Part ⑤ : Step ② : URL Shortening (Hash function) ⇒

③ URL Shortening Flows ⇒

Before we start point ③ एक बात बताओ → point ① 'URL redirection' के बारे में :-

tinyURL ——redirection——→ long URL

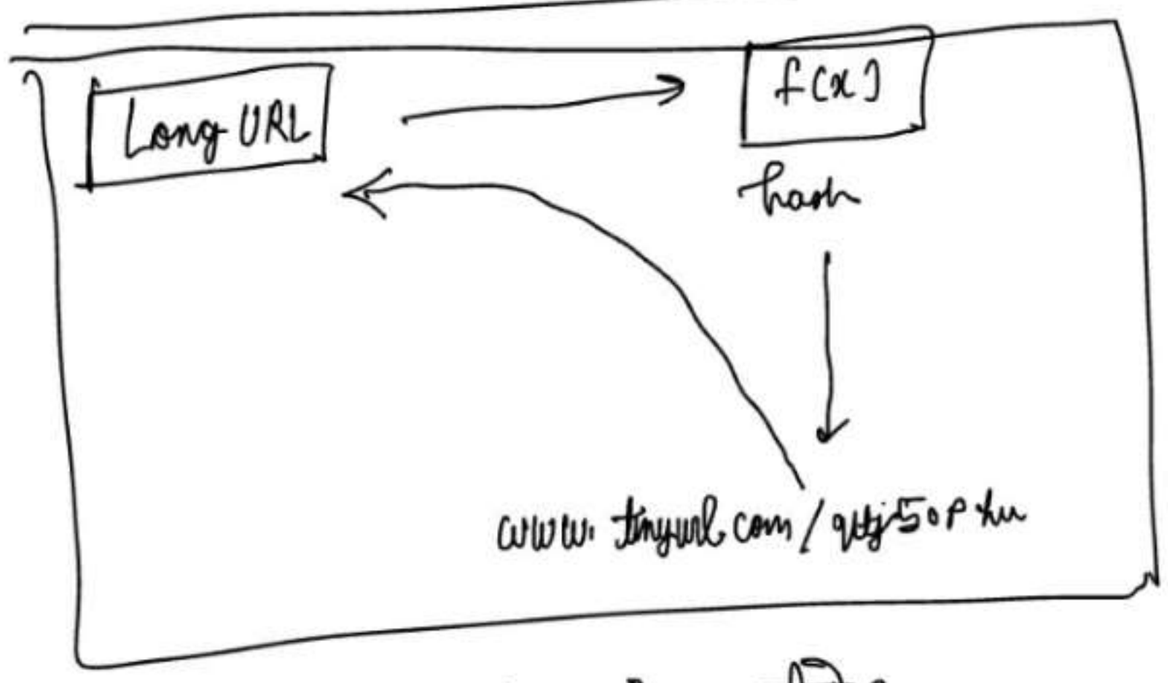
Basic तरीका → map < string , string > map;

shortURL long URL.

③ URL shortening flow →

example :- www.tinyURL.com/{hash_value}

We need a hash function f(x) such that :-



→ Hash function कैसा होना चाहिये ?

→ Each long URL must be hashed to one hash value.

→ Each hashvalue can be mapped back to long URL.

(48) Design a URL shortner (Part 6) →

Step ③ Design Deep Dive (Data Model) →

→ URL shortening

→ URL redirection

Now, we will deep dive inside →

→ Data Model → DataBase कैसा रखेंगे ?

→ Hash function →

→ URL shortening →

→ URL Redirection →

Data Model → Hash Table जैसा होता था <Key, Value>

फिर database क्यों चाहिये था?

<Key, Value>

↳ Hash Table → not feasible for real world systems. → क्यों??? → यार Memory resources limited होता Hash table me

So better option is →

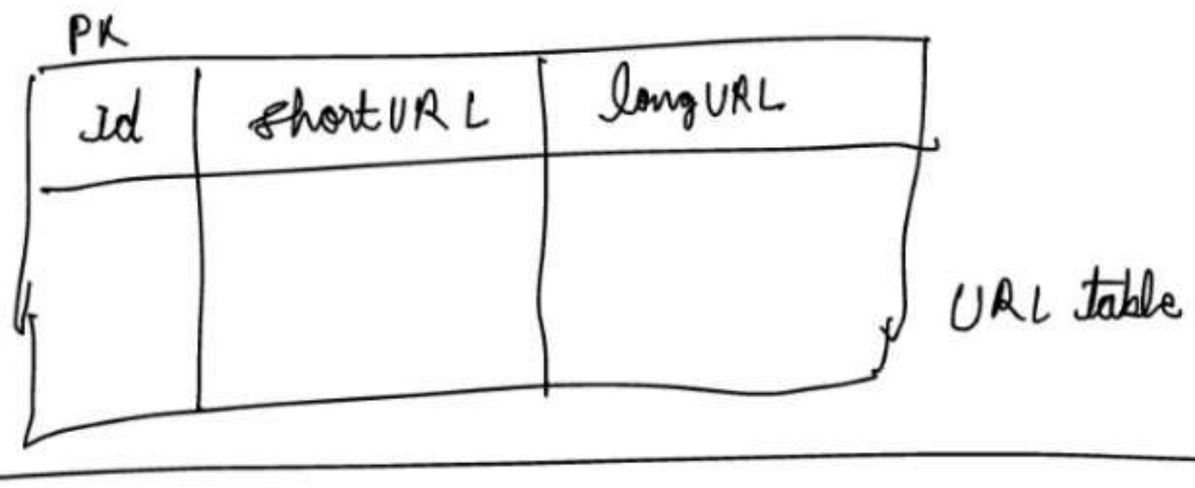
<Short URL, Long URL> → 

D.B.

Interviewer → Table Design कैसा रखोगे?

Candidate → 3 columns रखूँगा

id = primary key and auto-increment



URL table

---

(49) Design a URL shortner (Part 7) →

Design Deep Dive (Hash function) ⇒

* Hash function →

Interviewer → Hash value की length क्या रखोगे?

Candidate → Hash Value = [0-9 a-z A-Z]

→ Total possibilities = 10 + 26 + 26 = (62) possibilities

Hash Value length = n

combination = 62^n

62 62 62 62 62

URL support करना है = 365 billion. ⇒ $62^n \geq 365$ billion n = ??

N	Maximal no of URLs
1	$62^1 = 62$
2	$62^2 = 3844$
3	$62^3 = 238,328$
4	$62^4 = 14,776,336$
5	$62^5 = 916,132,832$
6	$62^6 = 56,800,235,584$
7	$62^7 = 3,521,614,606,208 = \approx 3.5$ Trillion

So $\boxed{n=7}$ is more than enough to store 365 billion URLs

Interviewer → What hash function you will be using ?

Candidate → ① "Hash + Collision Resolution"

② "Base 62 Conversion"

---

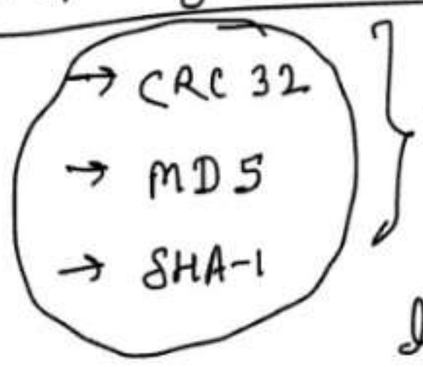
(50) Design a URL Shortner (Part 8) →

~~Step~~ (3) Design Deep Dive (Hash + Collision Resolution) ⇒

* Hash + Collision Resolution →

→ We want Hash function →

→ Straight forward solution → Use well known hash functions like →



These hash function available in market.

long URL :- (https://en.wikipedia.org/wiki/system_design)

Hash function	Hash Value
CRC 32	5cb54054
MD 5	5a6250 9a 84 df9cc 03 fe 1230 b9 df8 b84e
SHA 1	0 eeae 7916 c06 85 39 01 d 9cc befb fcaf4 de 57 ed 856

But सबसे छोटा (CRC 32) का भी length > 7 है ।

What to do ??

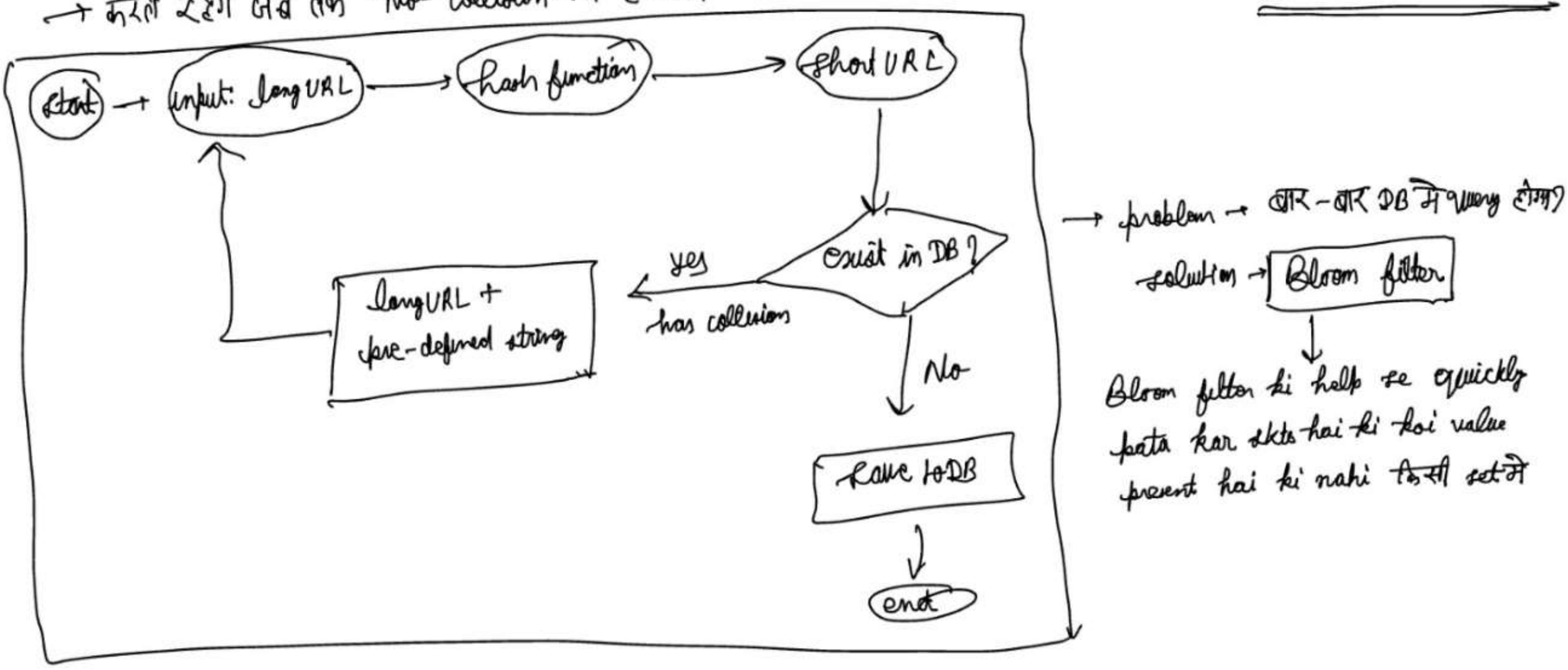
→ starting का 7 character उठा लो (X)

e.g → 5cb54054 } different है but दूसरे के 7 character same है।
5cb54059 } → Hash collision.

अब क्या करेंगे → ??

→ अगर collision हुआ तो long URL में एक 'Pre-defined string' append करके फिर से try करेंगे।
(longURL + predfined string)

→ करते रहेंगे जब तक "No-collision" ना हो जाये



→ problem → बार-बार DB में query होगी।

solution → Bloom filter

Bloom filter की help से quickly पता कर सकते हैं कि कोई value present है कि नहीं किसी set में

---

(51) Design a URL shortner

Part(9): Step(3) Design Deep Dive (Base 62 conversion) →

* Base 62 Conversion →

Hashed Value

0-9	→ 10
a-z	→ 26
A-Z	→ 26
	62

Take Unique Number → example → auto-incremented id of a MYSQL table.

e.g. → 11157 → convert this to Base 62. (0 1 2 ... 9 a b c ... z A B C ... Z)
(0 1 2 ... 9 10 11 12 ... 35 36 37 38 ... 61)
62 characters.

62	11157	59	X
62	179	55	T
62	2	2	2
	0		

https.tinyurl.com/2TX → shortURL

10	134	4
10	13	3
1·	1	1
	0	

$(10^2 \times 1) + (10^1 \times 3) + (10^0 * 4)$ → Decimal

$(62^2 * 2) + (62^1 * T(55)) + (62^0 * X(59)) = (11157)$

why → www.tinyurl.com/2TX

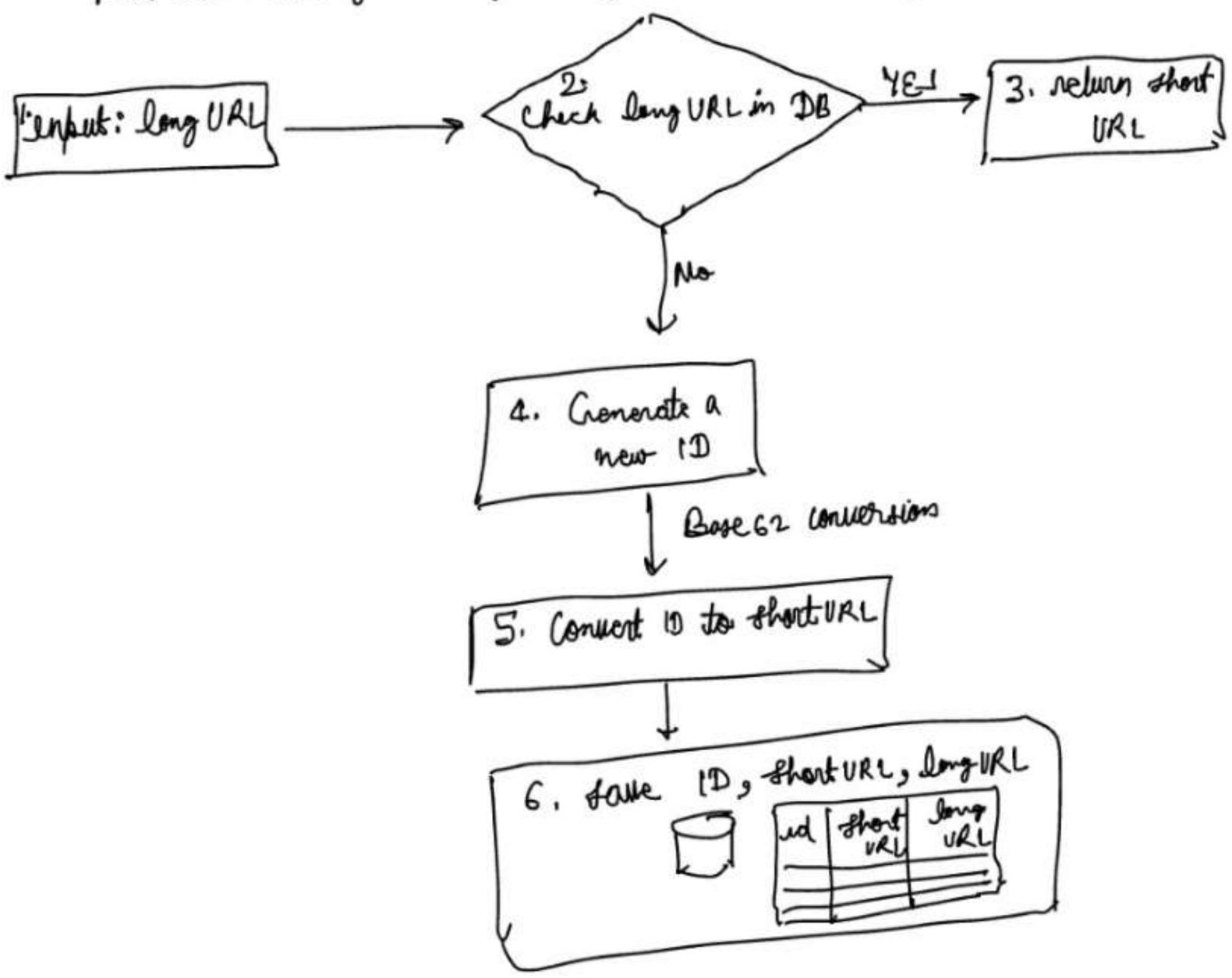
why not? → www.tinyurl.com/11157 → (11157 > 2TX → jitna small ; utna achcha)

Hash + Collision Resolution	Base 62 Conversion.
Fixed short URL length	The short URL length is not fixed - It goes up with ID.
It does not need a unique ID generator	This option depends on a unique ID Generator.
Collision is possible & must be resolved	Collision is impossible because ID is unique.
It is impossible to figure out the next available short URL because it does not depend on ID.	It is easy to figure out the next available short URL if ID increments by 1 for a new entry. This can be security concern.

(52) Design a URL shortner (Part 10) ⇒

Step (3) Design Deep Dive (URL shortening flow) →

Lets Draw the flow diagram for URL shortening flow →



example →

longurl :- https://en.wikipedia.org/wiki/System_design

unique ID generator returns ID: 20138653

Convert ID to short URL using: "m1en2f3"

Base 62 conversions

ShortURL : www.tinyurl.com/m5en253

id	ShortURL	long URL
2013853	m5en2f3	https://en.wikipedia.org/wiki/system_design

---

(53) Design a URL shortner ⇒

Part (11) Step 3&4 : Design Deep Dive and WRAP UP

(URL Redirecting Deep dive) →

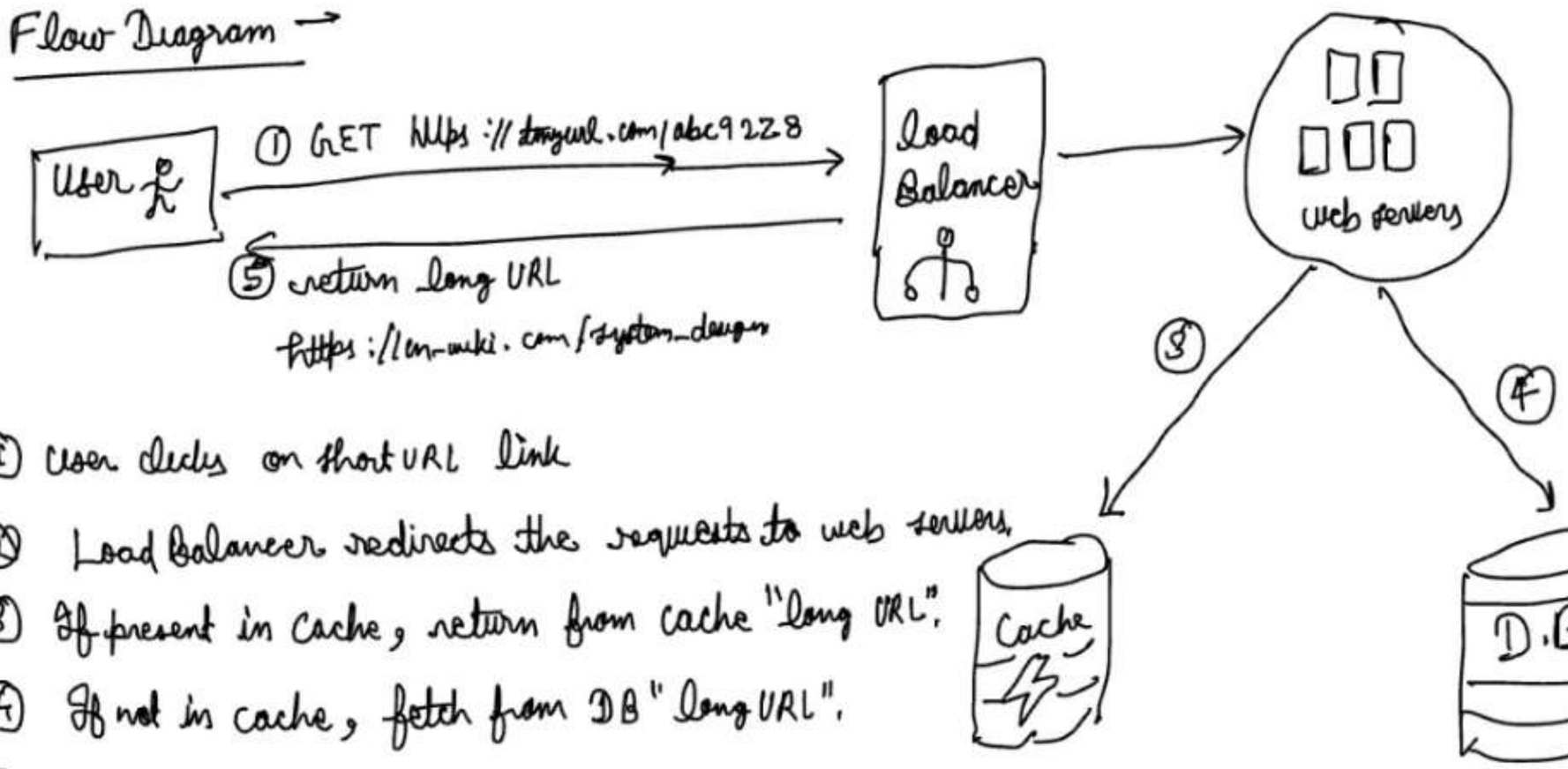
* Generally → More Reads then writes

(No of Read Requests) > (No of write Request)

< Short URL , Long URL > → Store in Cache Memory.

↳ Improved performance.

Flow Diagram →



① User clicks on short URL link

② Load Balancer redirects the requests to web servers.

③ If present in cache, return from cache "long URL".

④ If not in cache, fetch from DB "long URL".

⑤ Return long URL to user.

Step ④ WRAP UP ⇒

If there is extra time in the end you can discuss →

→ Rate limiter → malicious user → Bahut saare req bhej rha hai.

→ Web server scaling → Easy to scale up ↑ and scale down ↓

→ Data Base scaling. → Data Base Replication / Sharding.

(54) Design a Web Crawler ⇒ (Google Amazon Meta) →

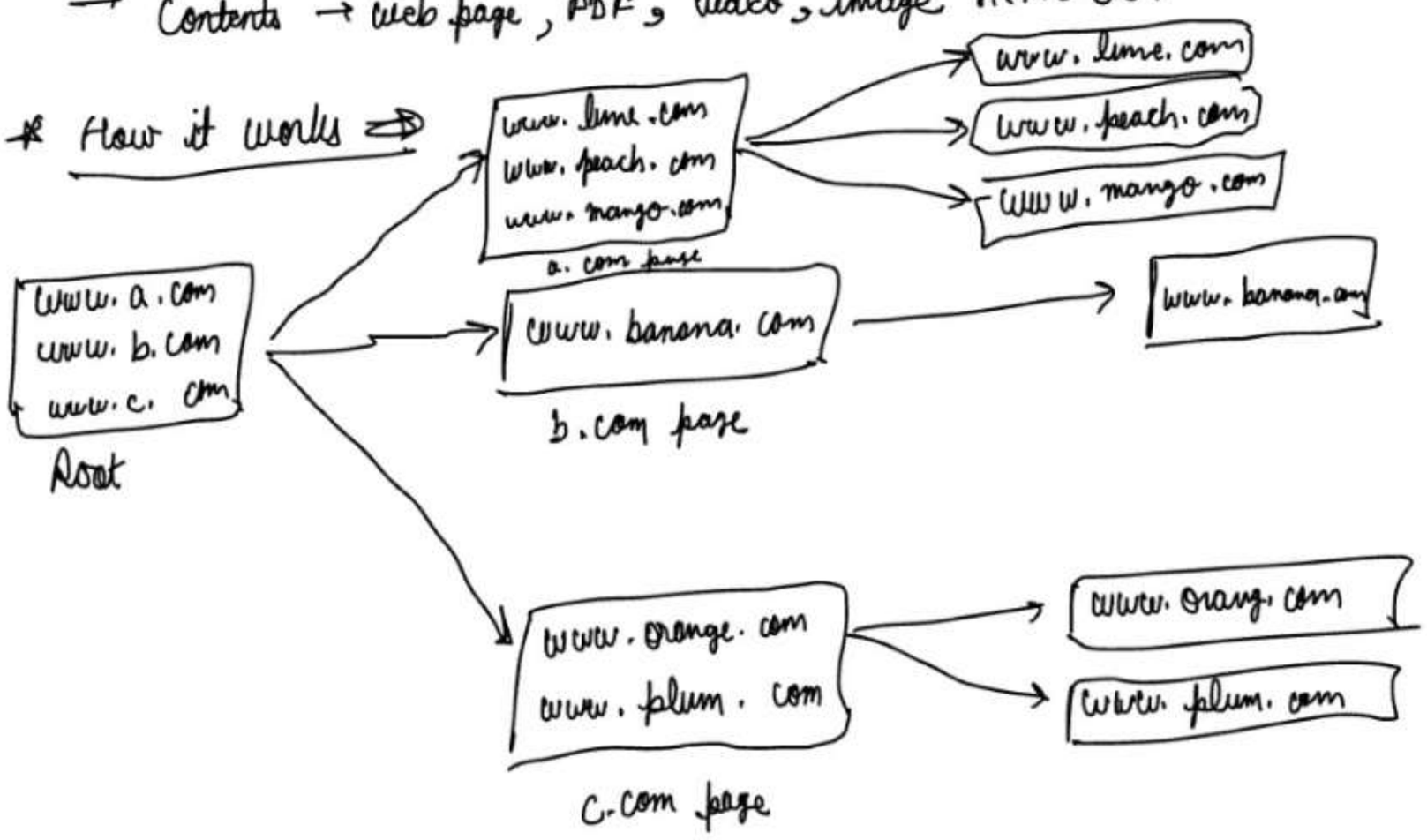
Web Crawler ⇒

→ Robot OR Spider

→ Used by search engines to find contents on Web.

→ Contents → web page, PDF, video, image HTML etc.

* How it works ⇒



Step ① Understand the problem and Establish Design Scope →

The Basic algorithm →

② Set of URLs → Download all web pages addressed by URLs.

③ Extract URLs → from these web pages.

④ Add new URLs to the list of URLs then, Repeat from step ②

सबसे simple है क्या? ⇒ (NO).

Lets start asking Questions →

Candidate → what is the main purpose of web crawler?

→ search engine indexing?

→ Data mining?

→ OR something else?

Interviewer → Lets go for 'Search Engine Indexing'

Candidate → How many web pages does the web crawler collect/month?

Interviewer → 1 Billion page.

Candidate → What content types are included? HTML/PDF/images etc.

Interviewer → HTML

Candidate → Do we also need to store the Downloaded web pages →

Interviewer → Yes store up to 5 years.

Candidate → What if we get web pages which are duplicate?

Interviewer → Ignore the Duplicate web pages.

Besides functionalities you should note down Characteristics of a

→ Good Web Crawler ⇒

→ Scalability →

→ web is huge

→ Billions of web pages

extremly efficient रखना पड़ेगा

using parallelization

→ Robustness →

→ BAD HTML

→ Unresponsive server

→ Malicius links etc.

→ Handle all these.

→ Politness →

→ Crawler should not make too many requests within a short time interval.

→ Extensibility →

→ system flexible होना चाहिये

→ New feature add करने के लिये Minimal changes करने पड़े।

→ example → Crawler image files in future.

---

(55) Design Web Crawler →

Part (2) : Step (1) : Back of the Envelop Estimation →

☆ Back of the envelope Estimation →

→ Assume → 1 Billion web pages download / month.

→ Query per second (QPS) → 1 000 000 000 / 30 / 24 / 3600

≈ 400 pages / second download करेगा हमारा web crawler.

→ Peak Query per second (Peak QPS) = 2 * QPS = 2*400

≈ 800 pages / second.

→ Assume → web pages average size = 500 KB

→ (1 000 000 000) * (500 KB) = 500 TB. storage / month.

→ Assume data will be stored for 5 years

→ Total storage Requirement = 500 TB * 12 * 5

= 3000 TB = (30 PB). storage will be needed for 5 years.

---

(56) Design a Web Crawler (Part 3) →

Step (2): propose a high level design & Get Buy in →

→ Starting में URLs चाहिये होगा ना to get started with seed URLs.

Choose Good seed URLs → Shopping URLs
→ Sports URLs
→ Health Care URLs etc.

→ जिन URLs को download करना है वो कहीं पर store करेंगे ना!!! → URL Frontier

→ URL frontier से URL मिलेगा तो download करेगा फिर web pages - HTML downloader

→ web page को download करने के लिये

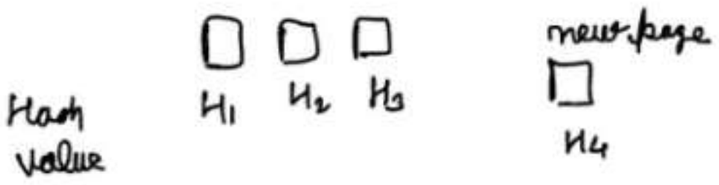
URL → (translate to) → IP address

→ DNS Resolver

→ Download होने के बाद parse, validate

- Content Parser

→ content seen? भी तो check करोगे??



$H_1 == H_4$ → content seen
$H_2 == H_4$ → content seen
$H_3 == H_4$ → content seen

→ content को store करने की भी बात हुई थी

Content Storage
↳ Disk (BC3, too many data)
↳ Memory (Quick access)

→ HTML pages से भी तो URL निकालना है।

URL Extractor

→ बेकार Broken links को filter भी तो करोगे

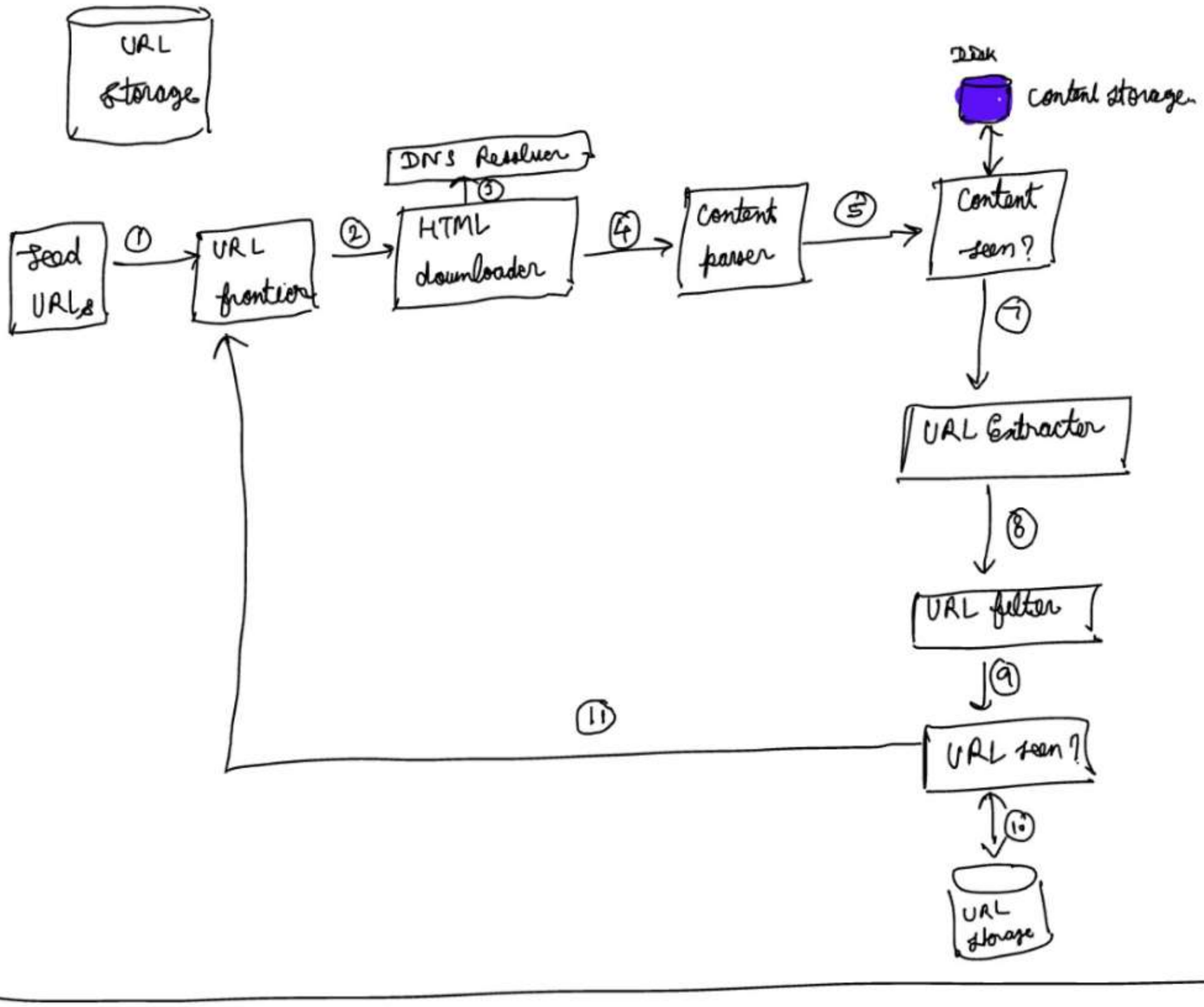
URL Filter

→ जो URL पहले देख चुके हो उसको दोबारा थोड़े ना देखोगे URL seen? ⟶ URL frontier में डाल देंगे

↳ Hash Table

↳ Bloom filter

→ जो URLs visit होगये उसको कहीं Store कर लेंगे।



---

(57) Design Web Crawler Part ④

Step ③ Design Deep Dive (BFS V/s DFS) →

→ DFS V/s BFS.

→ URL frontier

→ HTML Downloader

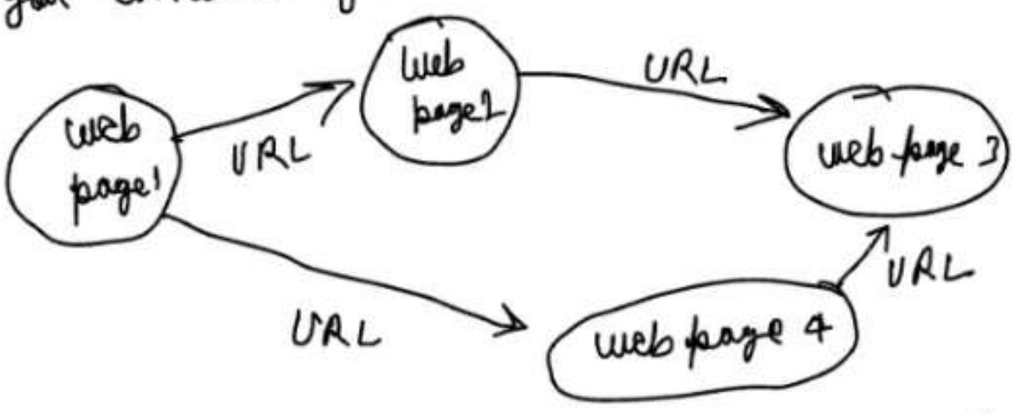
→ Robustness

→ Extensibility

→ Detect and avoid problematic Content.

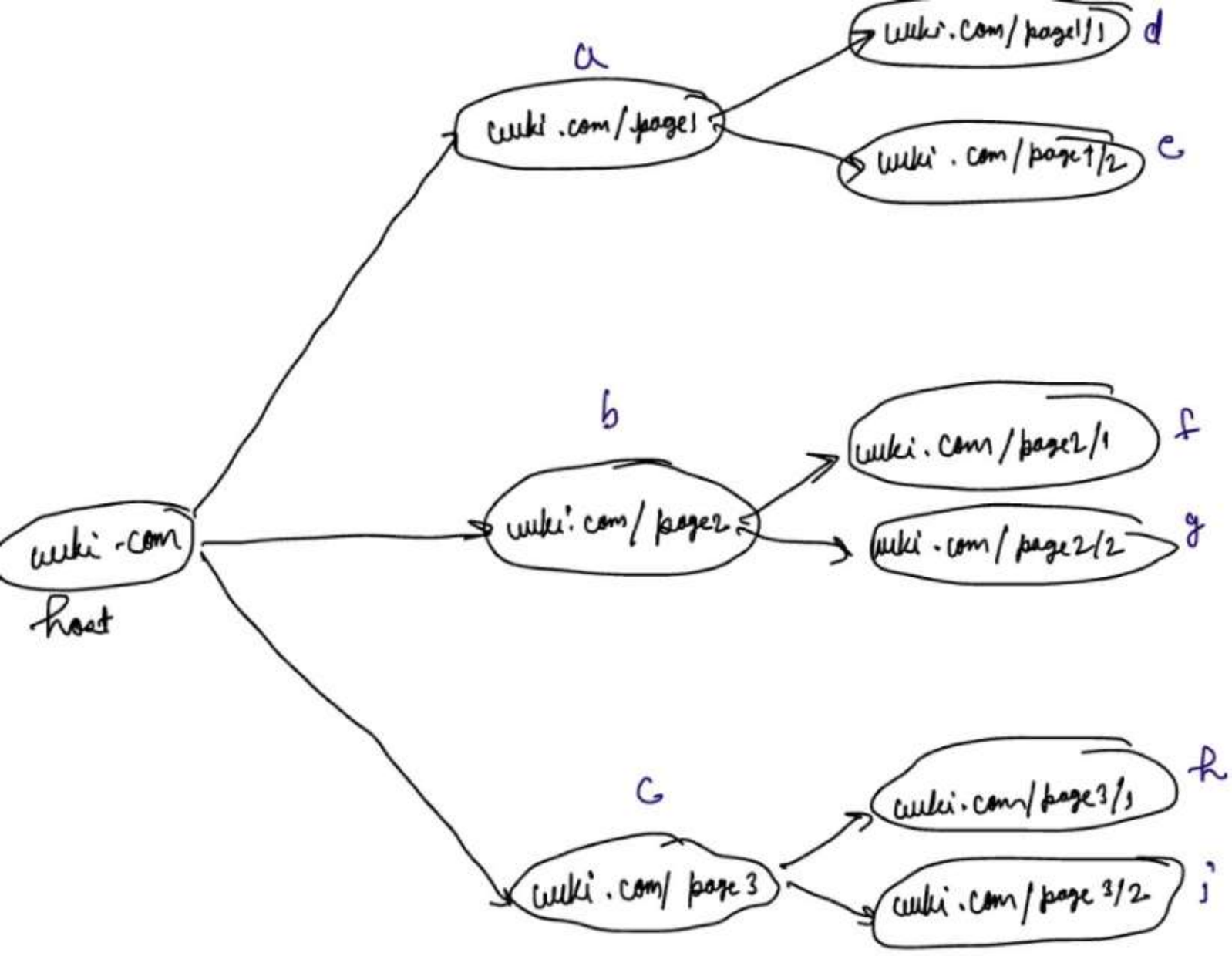
DFS V/s BFS →

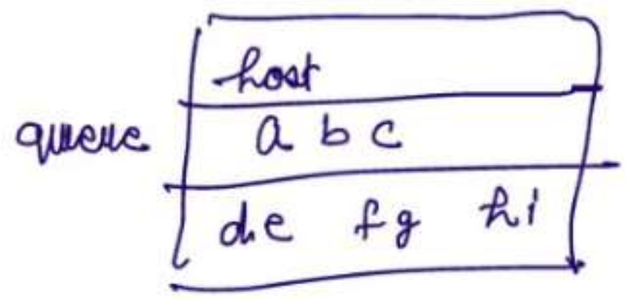
→ You can think of web as → Directed Graph.



CRAWLING → "Traversing a directed graph from one web page to other".

BFS & DFS →





Impolite Nature

↳ इसमें ही Host को use किए जा रहे

↓

wiki.com

BFS → जिसका Rank या priority ज्यादा है उसको पहले queue में insert करेंगे।

	Rank
webpage 1	1
webpage 2	2
webpage 3	3

(58) Design a Web CRAWLER part(5) →

step(3) Design DeepDive (URL Frontier) →

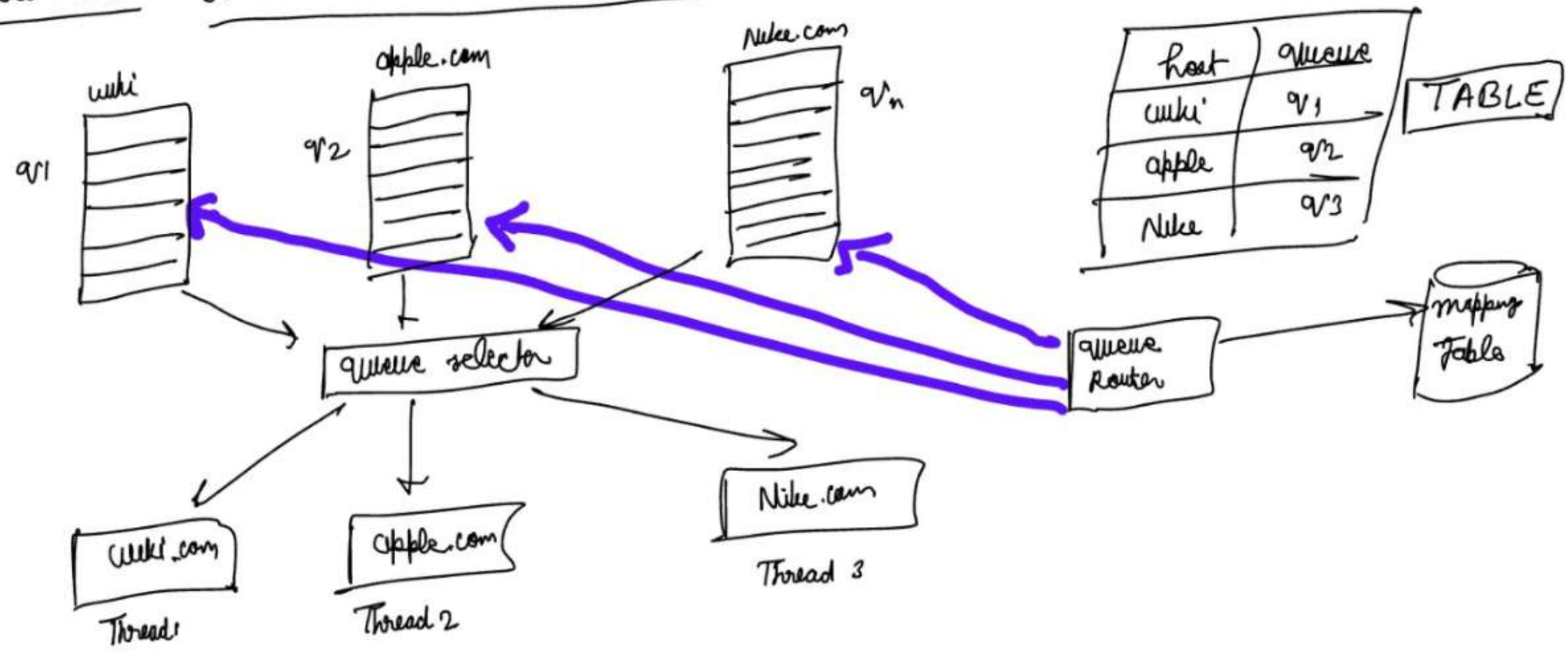
↳ BFS wali problem को solve करती है।

↳ Ensure politeness

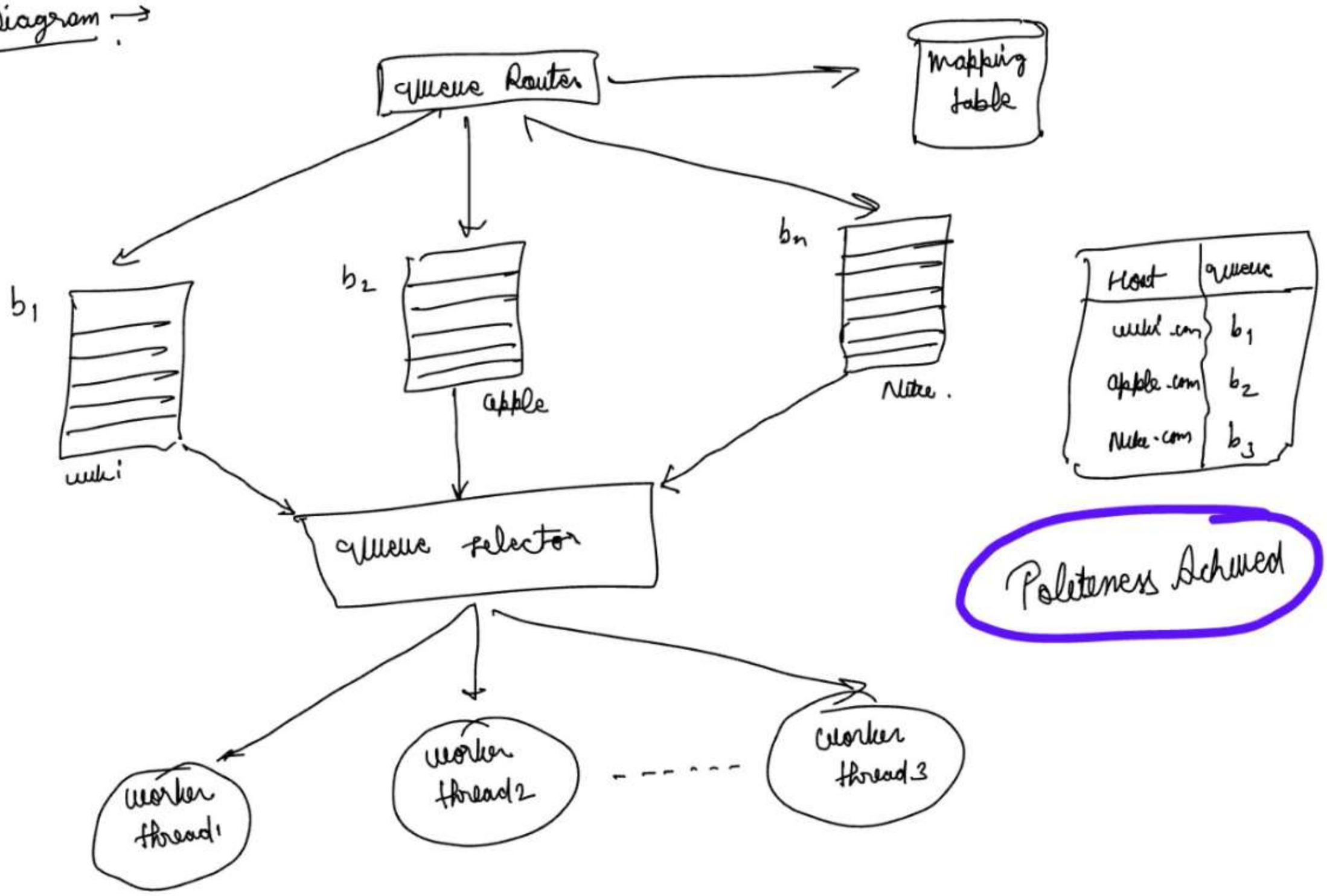
↳ URL prioritization

↳ Freshness.

Politeness → Avoid sending to many request to same host within a short period of time.



Nice Diagram →



Priority →

Random Discussion Forum

---"Apple"----

Apple Home Page

---"Apple"----

← सबसे important

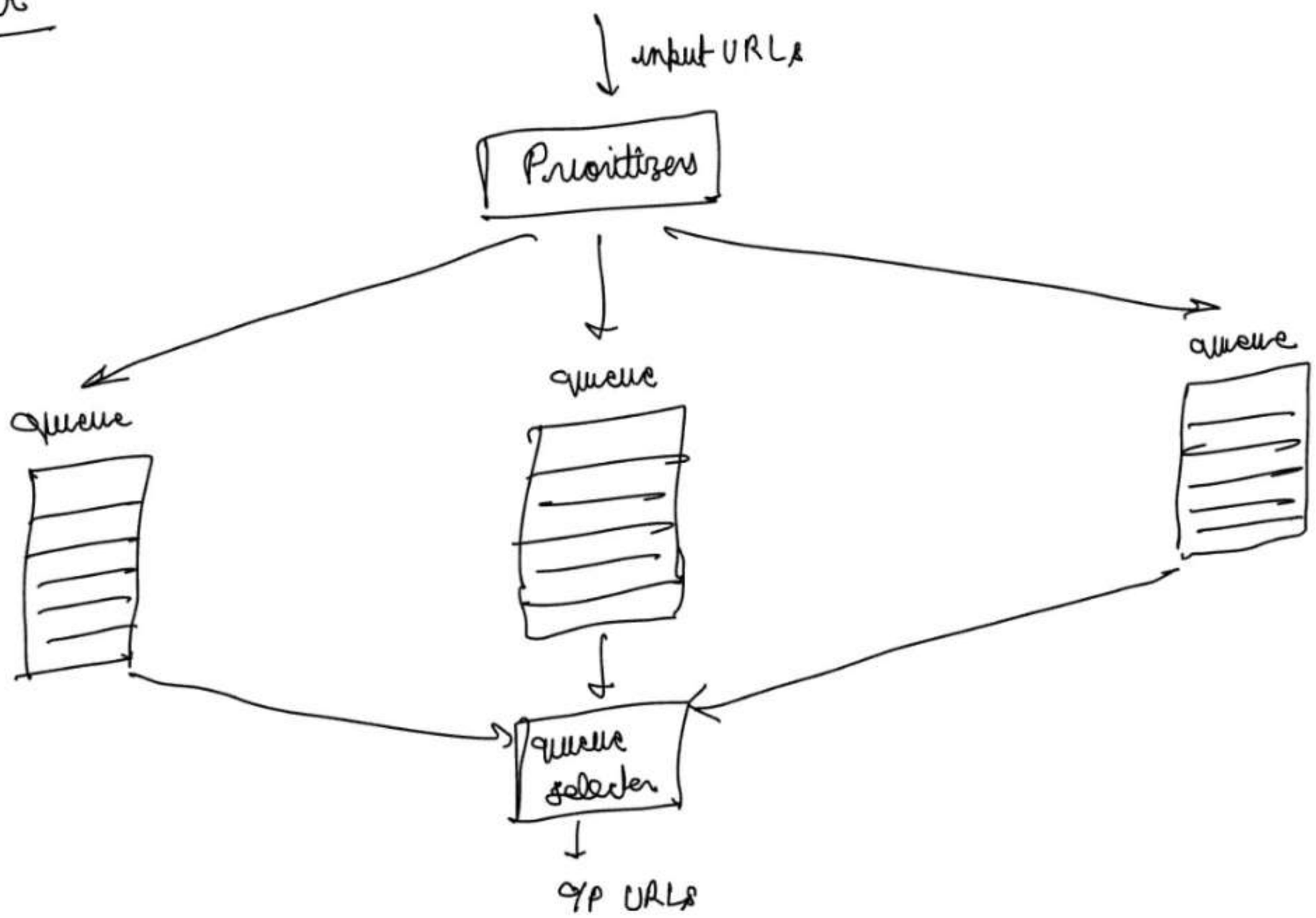
सबसे Priority.

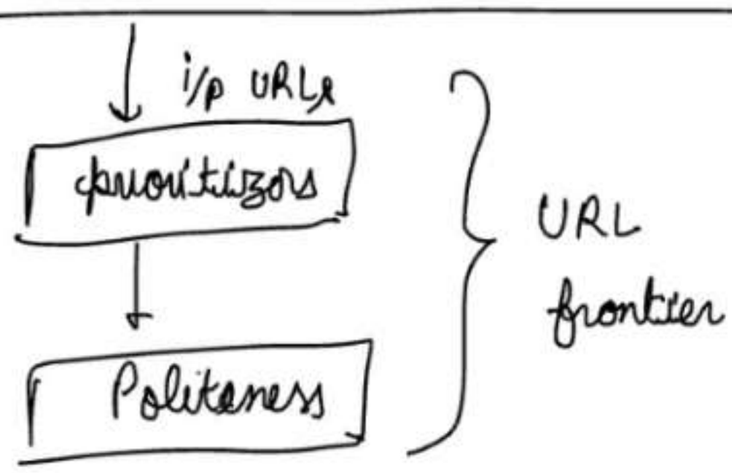
Usefulness → prioritize.

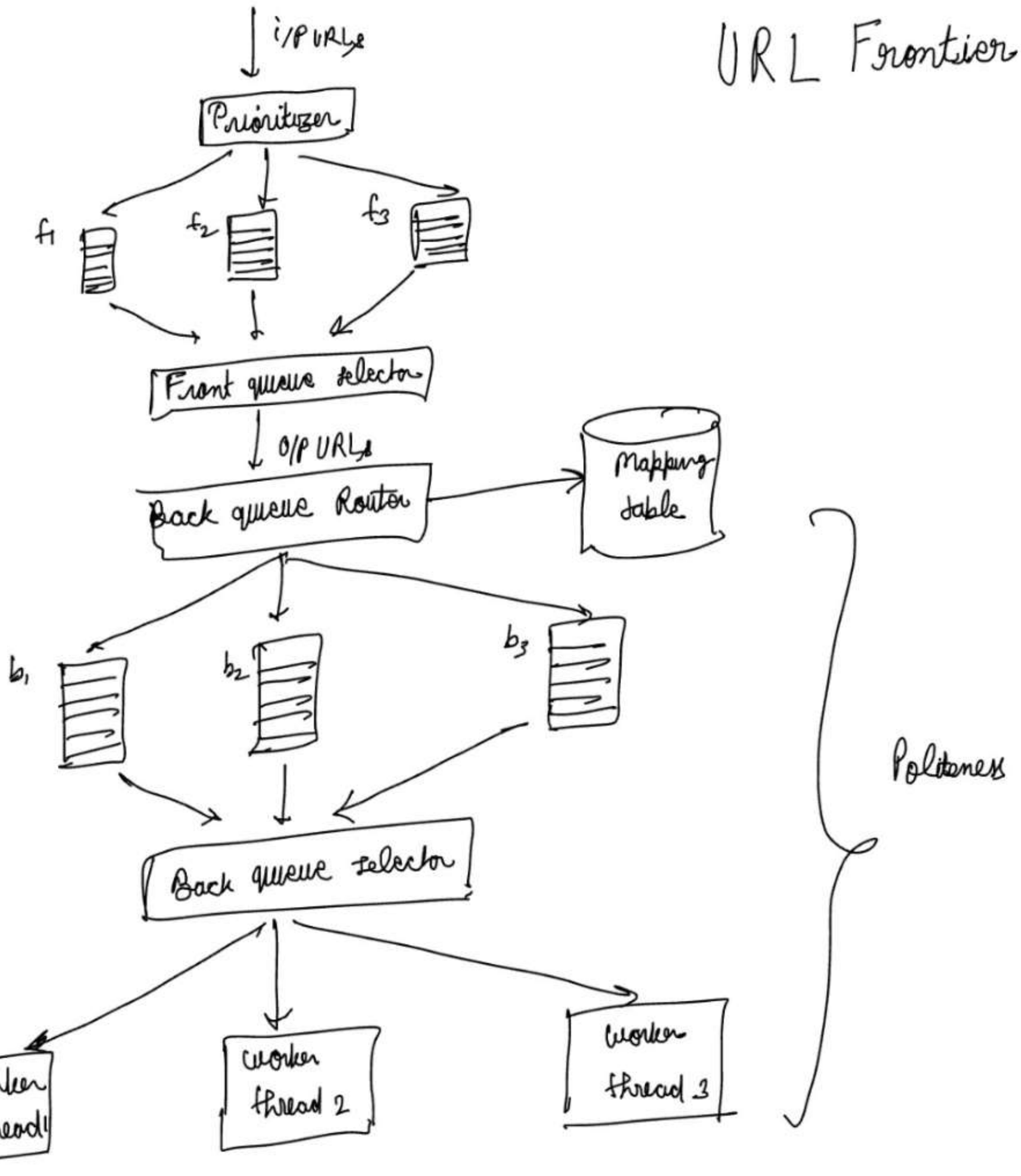
↳ More web traffic

↳ Update frequency

Prioritizer →







URL Frontier

Freshness →

Web Pages →

- Edited
- Deleted
- Added.

→ Last update History.

→ Prioritize. → web pages.

Storage for URL Frontier →

→ Real World → Millions of URLs

→ Putting everything on Memory

↳ Neither Durable

↳ Nor Scalable.

→ Disk में भी सब कुछ नहीं रख सकते

Because Disk is slow.

→ so we use Hybrid.

→ Disk → Majority URLs

→ Memory → Remaining URLs

To avoid read & write load on Disk.

---

(59) Design Web Crawler (Part-6) ⇒

Step ③ Design deep dive Complete & WRAPUP →

* HTML Downloader → Download web pages from internet.

Interviewer → Can web crawler download everything from web page? (Because its not safe)

Candidate → No, Crawler को कुछ Rule follow करने पड़ेंगे जो web page ने सेट किया है।

Amazon.com

Robots.txt

"Robot Exclusion Protocol"

Cache में store कर लो

amazon.com → Robot.txt } periodically update hota rahega "robots.txt"

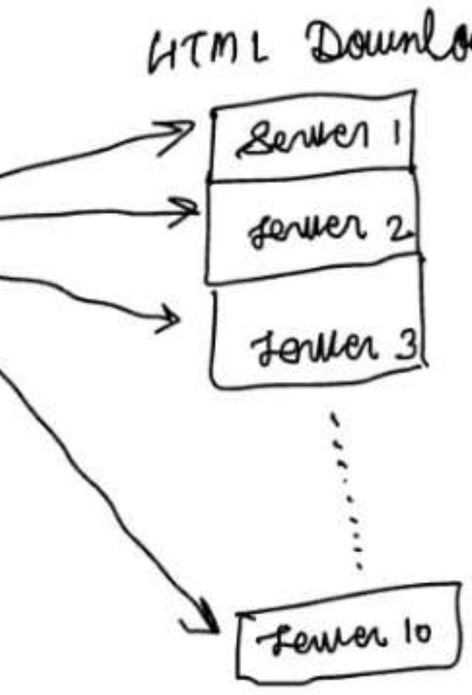
Nike.com → Robot.txt }

www.amazon.com/robots.txt

HTML Downloader →

Distributed Crawling →

URL Frontier → URLs



URL: amazon.com

Server इसको नहीं समझेगा

10.21.99.... ~

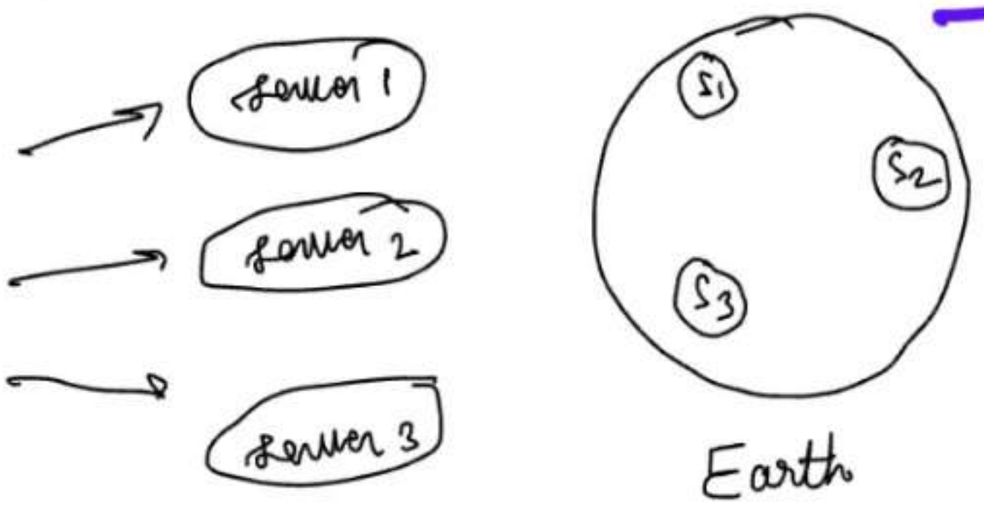
Server इसको समझेगा

→ Cache में DNS result को store कर लेंगे।

Amazon.com → 10.21.99.---

Nike.com → 10.32.87.---

HTML Downloader →

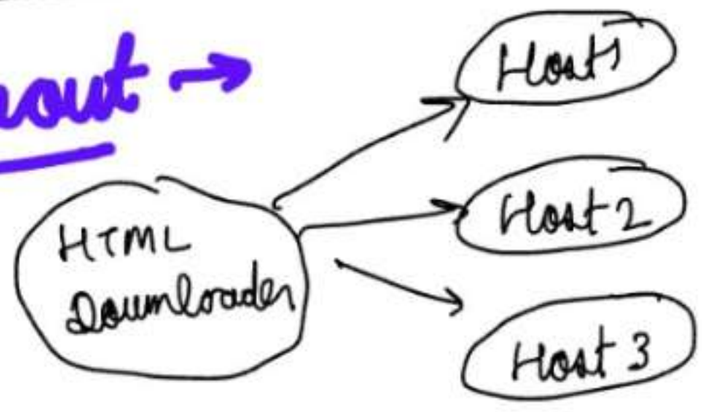


Locality :-

Geographically distributed for large

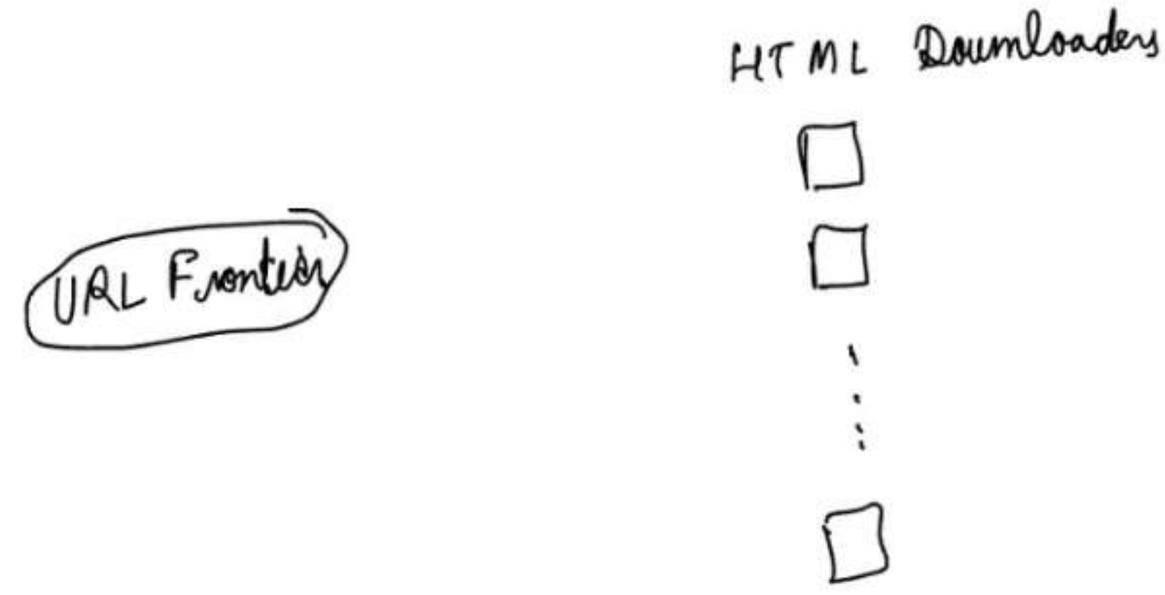
HTML Downloader →

Set Timeout →



Robustness →

→ किसी गड़बड़ को handle करने की ability.



→ We will use consistent Hashing.

माना लिजिये S_1 crash कर गया

→ Consistent Hashing

↳ Add/ Remove Server.

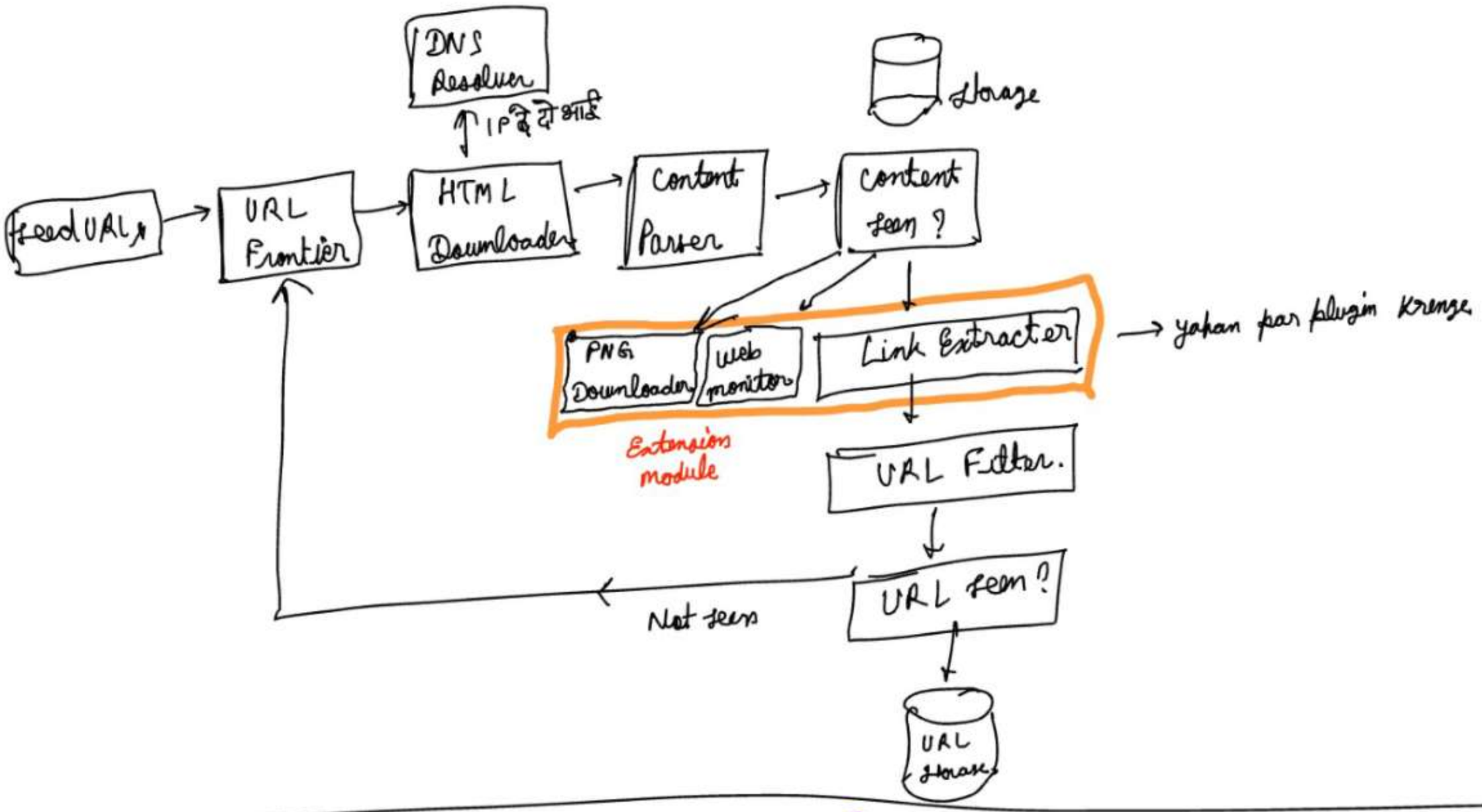
↳ we will save state & data of CRAWL

↳ Exception Handling (Handle error, crash etc gracefully)

☆ Extensibility →

Interviewer → What if in future, I want to download PNG files also? where in the design you will put it.

Candidate → New Modules plug कर सकते है।
Lets recall our high level Diagram.



Detect & Avoid Problematic Content →

→ Redundant (Duplicate) Data 30% duplicate data
→ Harmful Data (Spider Traps) → www.spidertrap example.com/foo/bar/foo/bar/foo/bar.....
→ Meaningless Data. (Data Noise)

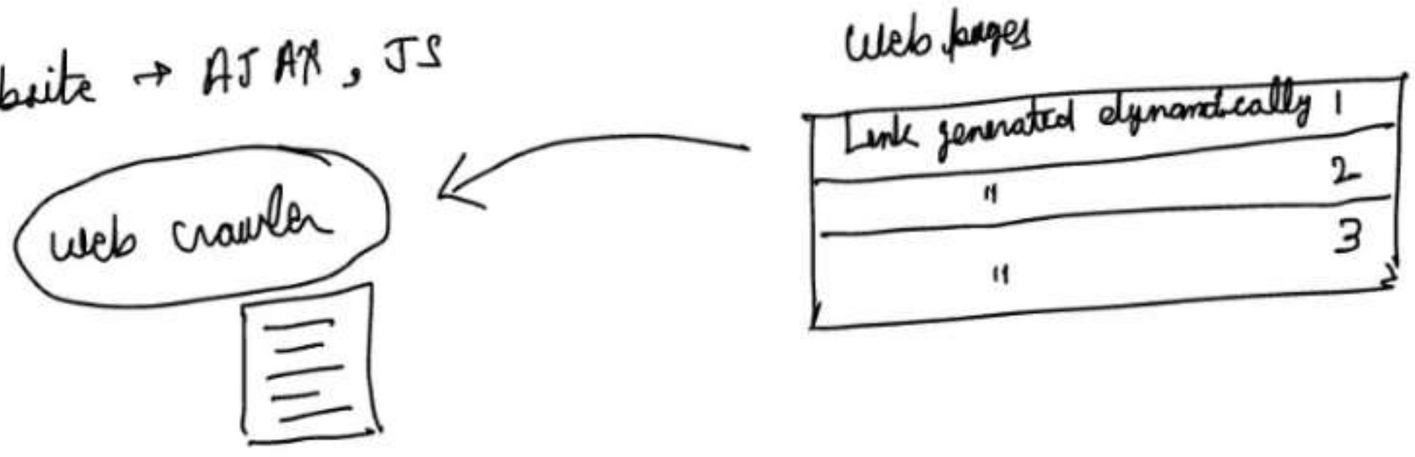
eg → advertisement
code snippet
spam URLs

WRAP UP →

Good CRAWLER →

① Scalability
② Politness
③ Extensibility
④ Robustness

Website → AJAX, JS



Server side Rendering →

→ unwanted Pages → filter out.

→ Horizontal Scaling

→ Availability consistency & Reliability.