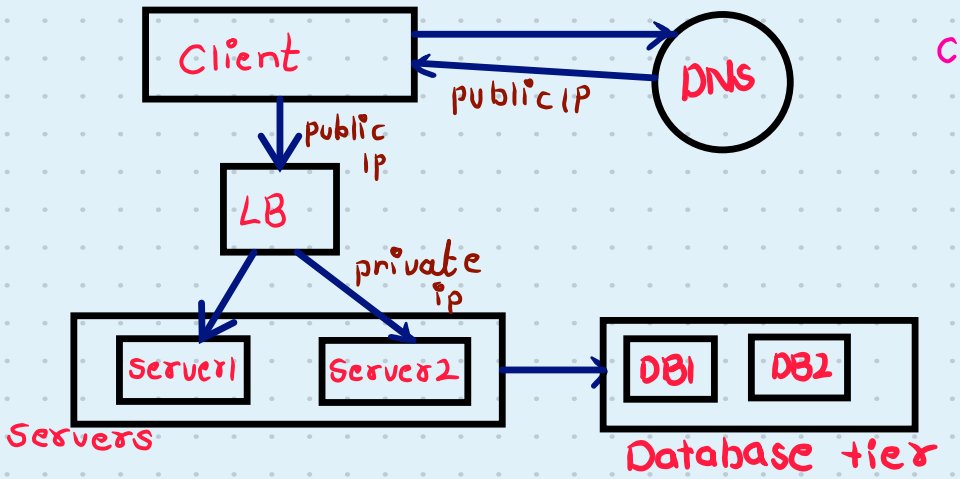
Lecture 4- Consistent Hashing || Load Balancers

#Load Balancer Recap



Flow
Client → DNS → Load Balancer → Servers

⊙Load Balancer ka kaam hota hai - incoming requests ko multiple servers mein distribute karna to reduce traffic on single server

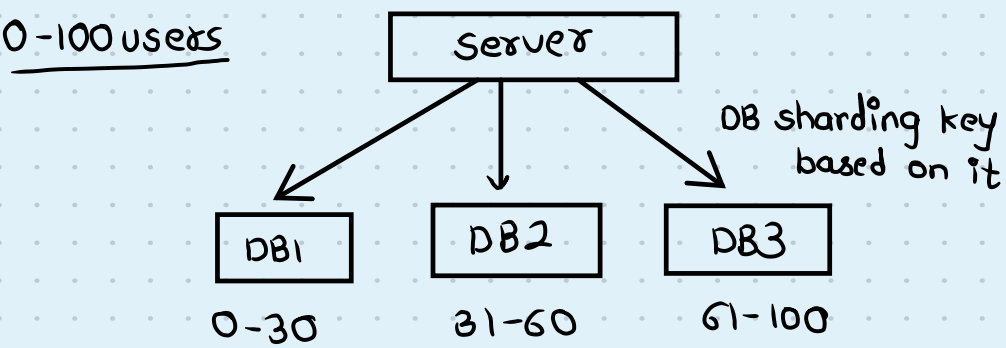
#Master - Slave Architecture

- Replication : multiple copies of DB
  ↳same copy of data from DB1 to DB2 , DB3 ,etc

- Why Replication ?
  ↳If main DB crash ho jaaye yaa respond naa kare toh -
  ① request koi aur replica handle karle
  ② High availability of our db

#Sharding

- Why Sharding
  → Ram kabhi na kabhi full ho sakti hai , isliye Sharding ka use hota hai taaki data ko chhote-chhote parts (shards) mein tod kar alag-alag machines yaa storage units par distribute kiya jaa sake. Isse system scalable , fast aur efficient ban jaata hai



◉ Sharding Layer :

Data ko range-based DBs mein divide karta hai

Database	user_id range
DB1	0-30
DB2	31-60
DB3	61-100

- DB ka load kaam hota hai
- Scalability improve hoti hai

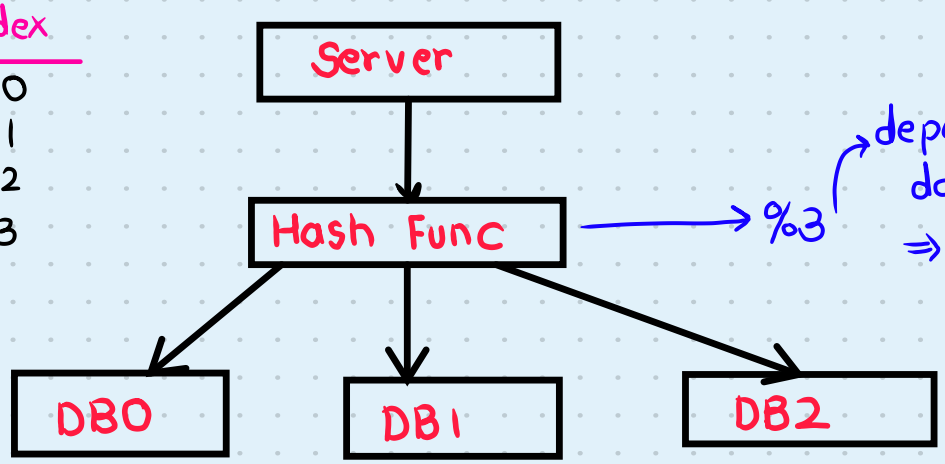
Ex Flow

user_id = 32 → request to DB2

user_id = 15 → request to DB1

Hashing Methods : Module Operator Algorithm

UserId	Index
key 0	0
key 1	1
key 2	2
key 3	3



Few Examples (No. of DB = 4)

user_id	Hash (%)	DB Index
40	40 % 4 = 0	DB0
3	3 % 4 = 3	DB3
12	12 % 4 = 0	DB0
43	43 % 4 = 0	DB3

No. of operation se user requests ko shard kiyaa jaata hai

Problems with Traditional Sharding

1) What if load increases ? or Any DB gets full ?
   → ek naya DB add karna padega
   → yaa koi server down ho jaaye toh usse remove karna padega
   Soln : Autoscaling support hona chahiye

2) If we increased no. of db
   → Hash function bhi badalta hai
   → Full db ko re-indexing karna padega + Migration karna padega
   → High cost + complexity
   → Hash func : %3 ⟶ %4
   → Full data redistribute karna padega
   → Re-hashing of all user IDs
   → Heavy DB Migration

3) If a DB goes down (eg : DB2)
   - firse keys ko adjust karna padega
   - Hash functions wapas %3 karna hoga
   - Mapping change karni padegi
   - complex + too much migrations

Conclusion

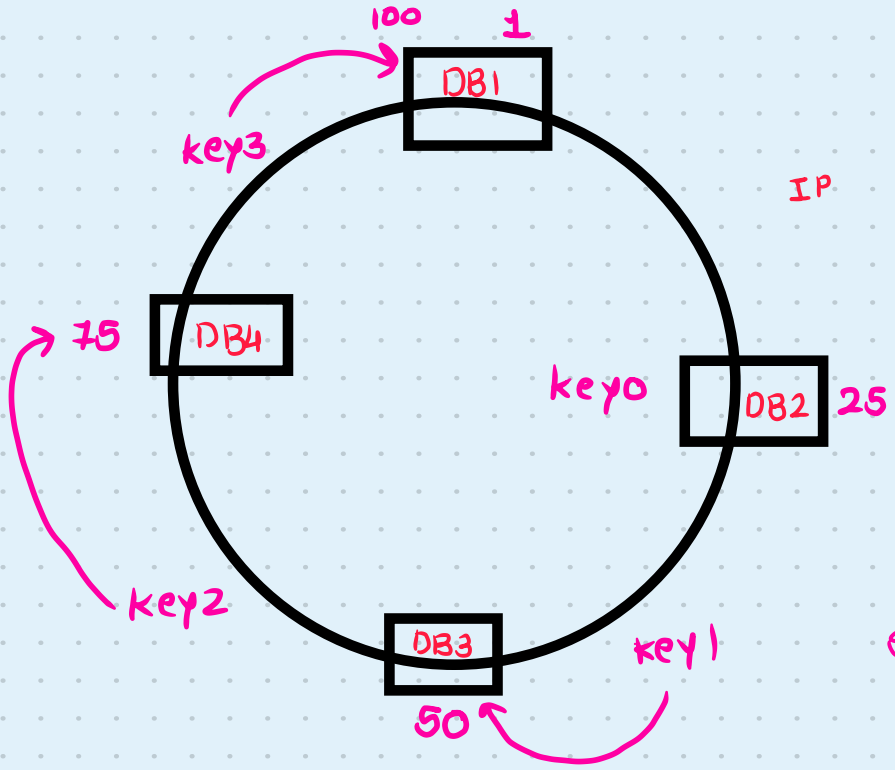
Traditional Hashing-based sharding mein scaling aur failure handling bahoot costly aur unstable ho jaata hai, isliye consistent Hashing ki need hai

Consistent Hashing
   - technique that allows minimum data migration ke saath easy scaling (add/remove DB nodes)

- What is Consistent Hashing ?
   ↳ Ismein ek hash ring (hash circle) ka use karte hai... jismein keys aur database nodes dono ko circle ke points par map kiya jaata hai
   ↳ Hash Ring ka range hota hai : 0-100

key0 : 0
key1 : 1
key2 : 2
key3 : 3

Hash-Ring (0-100)



hash(IP) = 1-100
hash Ring
Db hashing
keys hashing

hash(key0) = 11
hash(key1) = 41
hash(key2) = 65
hash(key3) = 80

Standard deviation should be minimised

⊙ Key-Mapping Logic

How to store key ?

- hash the key first ⟶ move in clockwise direction till you find your first DB Node ⟶ add the key in Node

⊙ Advantage -

1. Adding a New Node (eg DB5)
   - suppose hashed position of DB5 = 40
   - Ab agar - hash(key1) = 41 hai
     ↳ toh key1, joh pehle DB3 =50 ko point kar raha tha
     ↳ ab clockwise direction mein DB5 (40) ko point karega
   - ⊙ only key1 affected (minimum migration)
   - ⊙ baki keys apni jagah bani rehti hai

Limitations of Consistent Hashing

1. Uneven Distribution of keys
   - jab keys ke hash values mein kaam variation hota hai
     ∴ sab keys ek hi db mein map ho sakti hai
     Ex: all (hash value) < 25 ⟶ DB2
   - load imbalance hoga
   - ek hi db hotspot ban jaayega
   - remaining dbs (DB1, DB3, DB5) idle/underutilized rehte hai

2. Celebrity Problem / Hotspot Key Problem

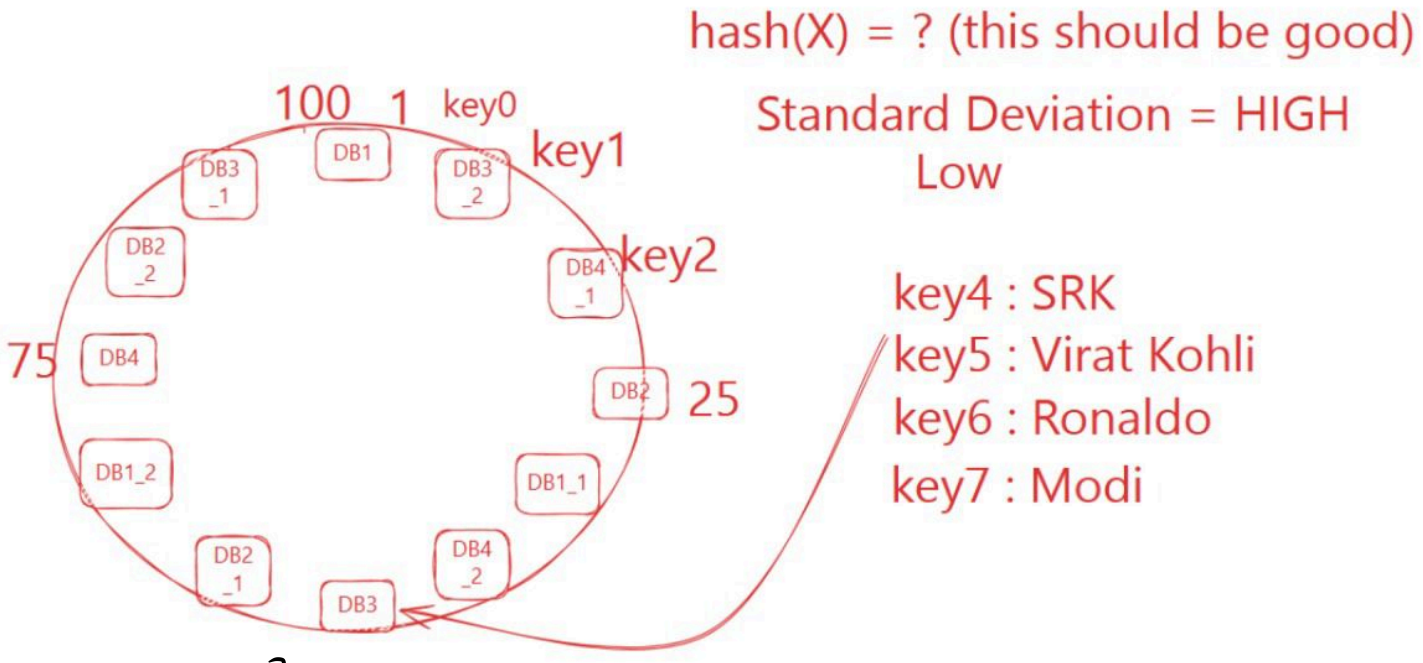
- koi ek key par jyada traffic aa raha hai
- even with consistent hashing, woh key ek hi node ko hit karega
- leading to :
  1. Single DB overload
  2. system-wide performance degrade

Solution ⇒ Virtual Nodes (VNodes)

What are Virtual Nodes ?

→ Har physical DB ko multiple virtual nodes/replicas ke form mein represent kiya jaata hai

Ex : DB1 ⇒ DB1.1, DB1.2
DB2 ⇒ DB2.1, DB2.2



How this work ?

- Yeh virtual nodes hash ring par alag-alag jagah par distribute hote hai
- isse database presence density badh jaati hai

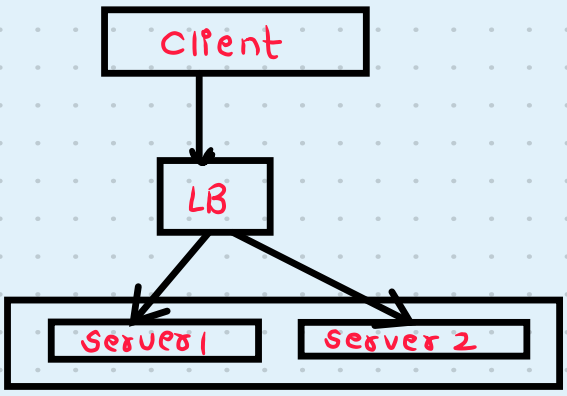
Why this solution ?

- keys ab jyada evenly distribute hote hai
- Ek hi region mein multiple dbs ke virtual node ho sakte hai
  1. isse load spread hota hai
  2. hotspot banne ka problem kaam hoti hai
  3. Low standard deviation issue solve hota hai

Load Balancing

- Load balancing helps in distribute incoming traffics into multiple backend servers
- LB targets to keep all server in balanced load so that no server gets overloaded

⦿ Flow



⦿ Functions of load Balancer

1. Which server is available
2. Which server can handle request
3. And then sends the request to server

- ensures no server is overload
- system efficiency ↑ses
- Availability of system ↑ses

Applications of load Balancing

1. Scalability
   - Autoscaling supported
   - depending on traffic, server from server pool scale up|down
   - efficient handling of fluctuating traffic

2. Availability
   - Incase, any server fails
     1. Load Balancer detects the server
     2. Stops sending request to that server
   - Ensures high availability and zero downtime

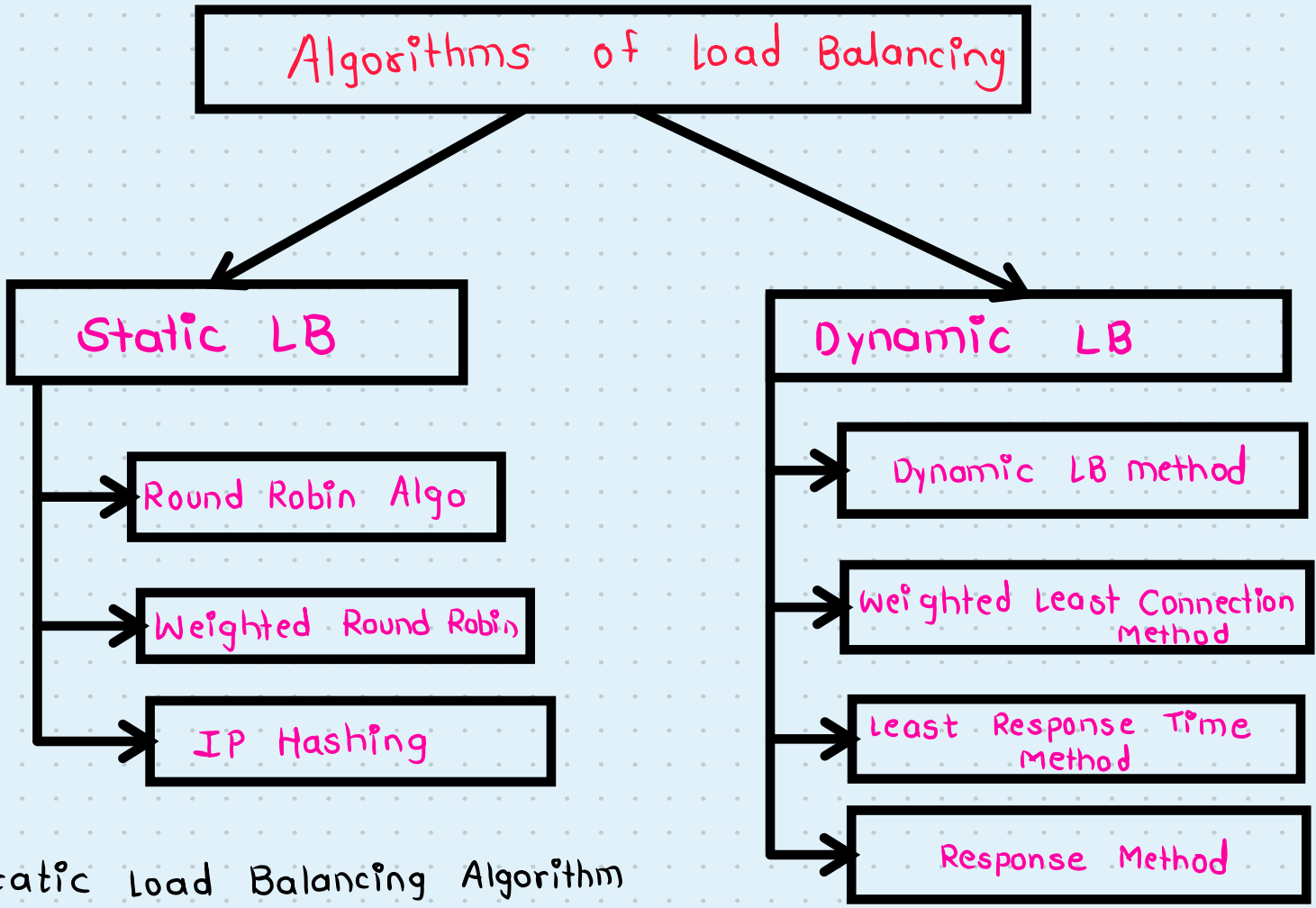
3. Security

- DDoS Attack Prevention
  1. Malicious traffic ko filter karta hai
  2. Backend servers tak nhi pahuchne dete
- Firewall Integration
  1. Bot, malware jaisi unwanted requests ko block karta hai

4. Responsiveness

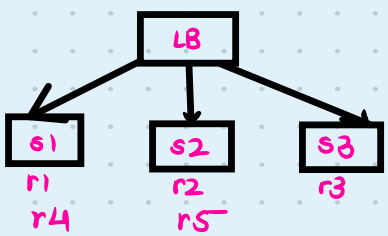
- Requests ko load multiple servers mein divide hota hai
- Isse servers fast respond karte hai
- Better user experience ensure hota hai

Algorithms of Load Balancing



Static Load Balancing Algorithm

1. Round Robin Algorithm



How it works?

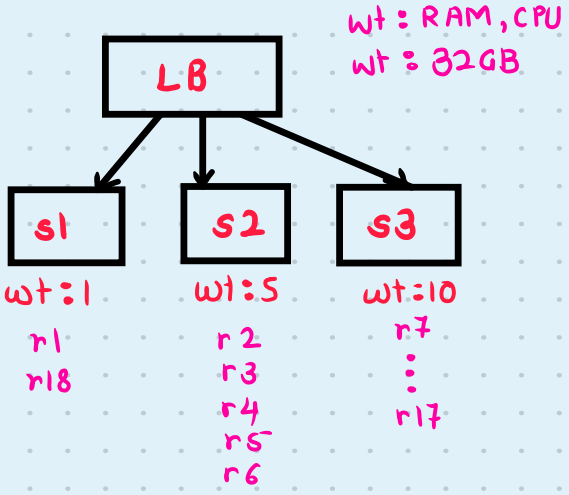
1. Sabse simple aur easy to implement algorithm
2. Requests ko sequentially servers ko assign krta hai -

Request 1 → Server 1
Request 2 → Server 2
⋮
Request 4 → Server 1

Limitation

1. Server ke current load ke dhyaan mein nhi rkhta
2. Sabhi servers ko equal request deta hai, chahe woh busy hoo yaa free
3. good for small applications

2. Weighted Round Robin Algorithm



⊙ How it works?

- Har server ko uski capacity (RAM, CPU) ke hisaab se ek weight diya jaata hai
- jyada weight wale server ko jyada requests milti hai

Example

Server	Weight	Request Distribution (Proportional)
S_1	3	3 req
S_2	5	5 req
S_3	10	10 req

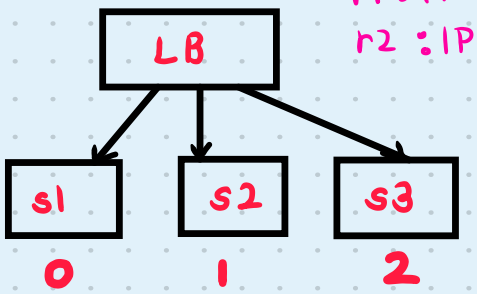
⊙ Total requests 18 ho toh distribution issi ratio mein hoga

Limitations

1. Server ke real-time load ke monitor nhi karta
2. Weight fixed rhta hai, dynamic load variation ko handle nhi karta

3. IP Hashing Algorithm

r1 : IP : hash(x) = ? (MOD)
r2 : IP : hash(x) = ?



⊙ How it works?

-client ko IP address ko hash krke determine karta hai ki usse konse server par bhejna hai

⊙ Formula

server_index = hash(client_IP) % no. of servers

Tightly couples (stateful architecture)
↳coz same IP ka hash same hi ayenga ⇒ so ek user ek hi server par jaayenga

⊙ Limitation

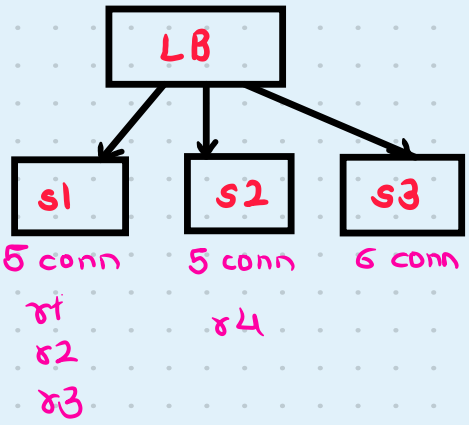
→ Hotspot Problem

- Agar koi clients same public IP share kar rahe hai (ex - office/college N/w)
- Toh sbki requests ek hi server ho jaayengi → woh server overload ho sakta hai
- baaki server underutilized rahenge

Dynamic Load Balancing

- Dynamic algorithms request distribution ka decision real-time server state ke aadhar par lete hai - jaise ki active connections, response time, yaa server health

1] Least Connection Method



⊙ How it Works :

- Requests uss server ko bheji jaati hai jiske paas sabse kaam connections hai

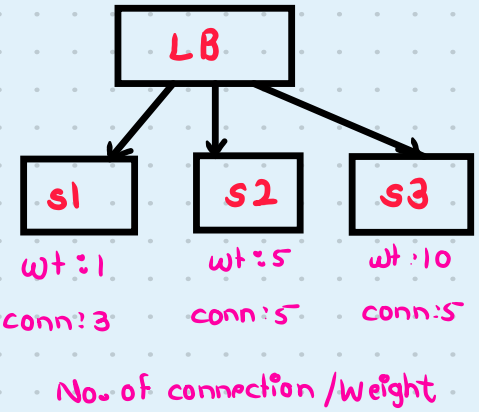
⊙ Best suited when :

1. sabhi connections ka size similar hoo
2. connections long-lived ho

Advantage

1. Uneven load distribution ko avoid karta hai
2. do not put much load on already busy servers

2] Weighted Least Connection Method



⊙ How it works ?

- Least connection ka enhanced version
- Har server ko ek weight assign kiya jaata hai (CPU / RAM ke hisaab se)

⊙ Load balancer calculate karta hai

$$\frac{\text{active\_connections}}{\text{weight}}$$

→ acc to this ratio ke anusar req ko assign krta hai

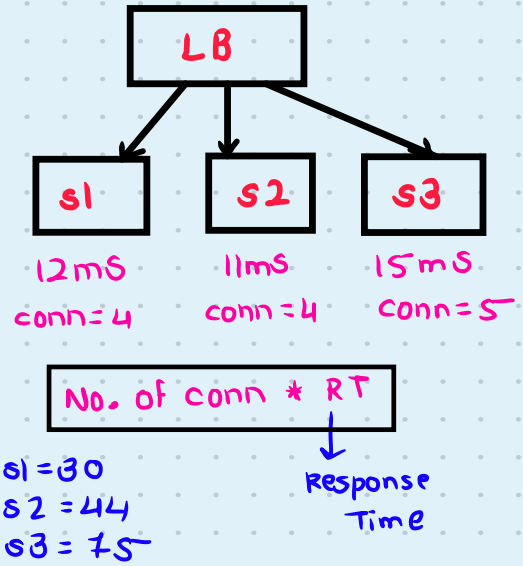
Example

Agar S1 ka weight 5 hai aur S2 ka 10, aur dono par 10 connections hai, toh S2 ko next request milegi coz -

S1 : 10/5 = 2.0

S2 : 10/10 = 1.0 → lower ratio

3] Least Response Time Method



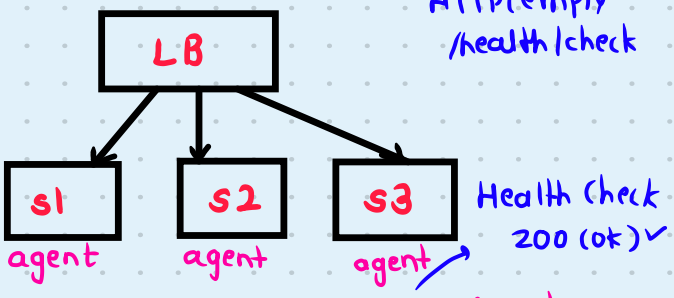
⊙ How it works?

- Requests ko server par bheja jaata hai, jiska response time sbse kaam hai
- Response time ko health check ke madat se measure kiya jaata hai

⊙ Useful for -

1. High traffic websites
2. Applications where low latency is critical

4] Resource Method



Capacity / load

1. no. of active conn
2. response time
3. latency
4. health check

Agent

1. checks if server is up
2. At a particular moment what is capacity or load in that server

⊙ How it Works?

1. Load balancer har server ke live resources ko monitor karta hai
   ↳ CPU, RAM, Active connections, response time, latency
2. Joh server ke paas sbse kaam load hota hai, request wahi jaata hai

Agents Involved

- Har server par ek agent hota hai
- Yeh agent real-time metrics collect karta hai aur load balancer ko bhejta hai

⊙ Health Checks

- Determine karta hai ki koi server "up" hai yaa "down"

How it Works

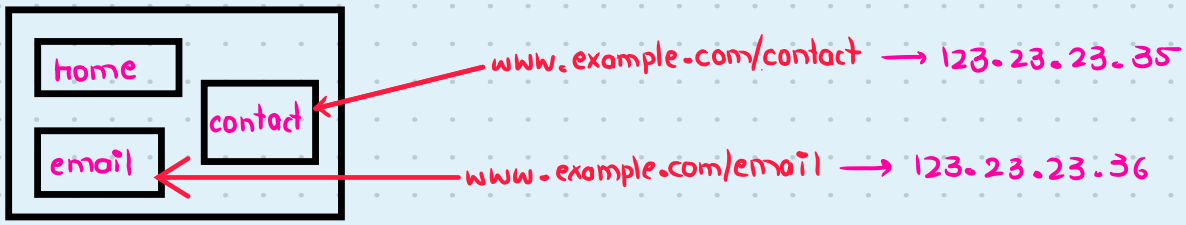
1. Load balancer server ko HTTP request bhejta hai (Ex: /health)
2. Agar response : HTTP 200 ok aata hai → server healthy mana jaata hai
3. Agar health check fail ho jaata hai → load balancer uss server ko request bhejna band kar deta hai

⊙ Why Important?

- Faulty yaa dead servers avoid karne se -
  1. System ki availability bani rahti hai
  2. better experience for user

Types of load Balancer : Based on layers

1. Application level DB (HTTP / HTTPS)
2. Network layer DB (IP Address)
3. Global Server load balancing (data center : GEO_DNS)
4. DNS load balancer



Types of LB