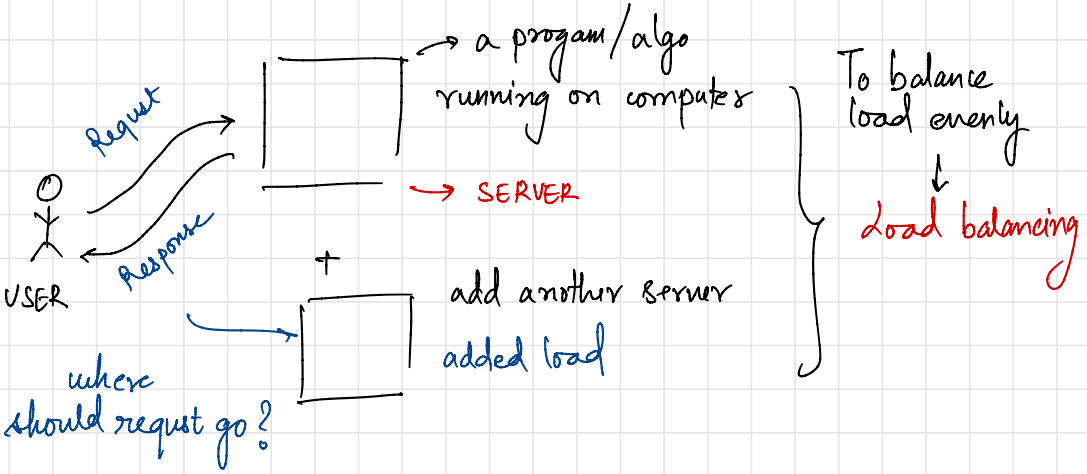
LOAD BALANCING ?



Concept of consistent Hashing for load Balancing

Request ID for each request (ideal random IDs)

0 — (M-1)

→ h (r_1) → m_1 % n

n = no of servers

hash function

h(20) = 15%4 = 3

h(35) = 12%4 = 0

- server 0
- server 1
- server 2
- server 3

∴ uniform load on servers

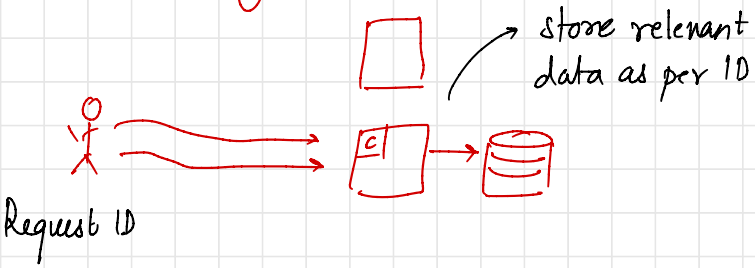
In actual scenario IDs are not random

one ID repeat

↓

sent to same server again & again

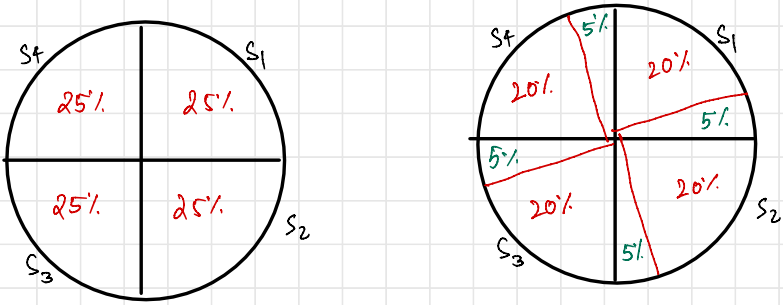
Using a cache memory at server (local data)



Instead of increasing more servers → great change in system

small changes in complete system

more efficient



CONSISTENT HASHING

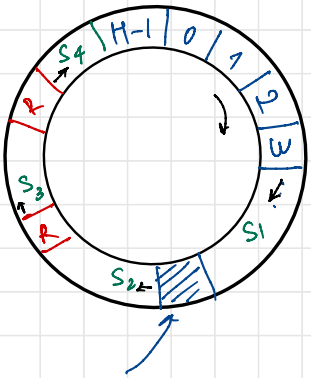
Request ID $\longrightarrow h(r_1)$

S_0 S_1 S_2 S_3 S_4

hashing server IDs

$h(0) \% M$

M = Search Space



request point to server

Going clockwise finding nearest server

server next in clockwise direction will serve request

$$\text{Expected Load factor} = \frac{1}{\text{no. of server}} \quad \text{(ideal)}$$

Why use this architecture over simple hash?

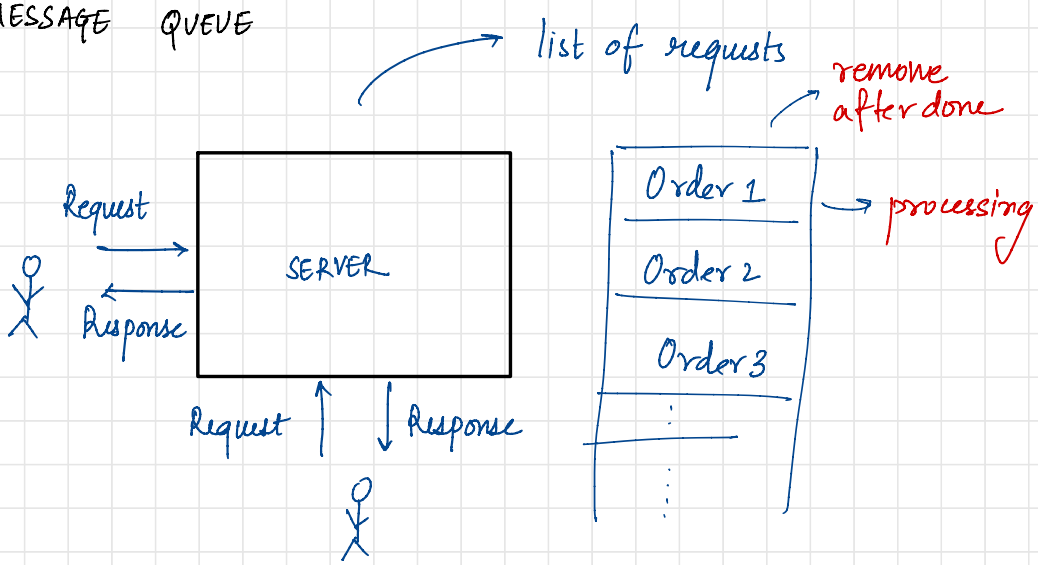
* Adding a new server will not change system load drastically
* No issues for adding new servers.

Disadvantage?

- Skewed load balancing in practical

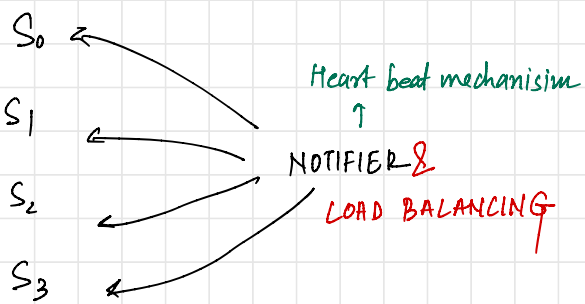
Solution; Making Virtual Servers
  - ↳ using multiple hash functions
  - ↳ k hash functions

MESSAGE QUEUE



ASYNCHRONOUS PROCESSING

* In case of fault other servers taking requests



ID	TYPE REQUEST	DONE ?
1	P1	Y
2	P2	N
3	P3	N

IF FAULT with server it will stop responding to notifier

↓

DIVIDE NOT DONE REQUESTS → USING LOAD BALANCER to ensure no DUPLICACY

MESSAGE / TASK QUEUE

Take request → assign to server → If server takes long (assuming fault)
↓
send to next server

Examples of Task Queues (Encapsulating complexity on serverside)

- Rabbit MQ
- Zero MQ
- Java Messaging Service