SYSTEMS DESIGN

SYSTEMS DESIGN CONCEPTS

i. Introduction
ii. Client – Server Model
iii. Network Protocols
iv. Storage
v. Latency and Throughput
vi. Availability
vii. Caching
viii. Proxies
ix. Load Balancers
x. Hashing
xi. Relational Databases
xii. Key – Value Stores
xiii. Replication and Sharding
xiv. Leader Election
xv. Peer to Peer Networks
xvi. Polling and Streaming
xvii. Configuration
xviii. Rate Limiting
xix. Logging and Monitoring
xx. Publish/Subscribe Pattern
xxi. MapReduce

SYSTEMS DESIGN MORE CONCEPTS

i. Hoizontal and Vertical Scaling

SYSTEMS DESIGN BASIC QUESTIONS

i. Design AlgoExpert
ii. Design a Code-Deployment System
iii. Design a Stockbroker
iv. Design Facebook News Feed
v. Design Google Drive
vi. Design the Reddit API
vii. Design Netflix
viii. Design the Uber API
ix. Designing WhatsApp
x. Designing Tinder
xi. Designing Instagram News Feed
xii. Google Maps : Location based database

SYS DESIGN FUNDAS

4 categories → intertwined

1] Underlyin, foundaᵗᵃˡ knowledge → eg: Client-server model, network protocols. Gotta atleast understand all these to have a knowledge of sys. des.

2] Key characteristics of sys → Thing you might want a sys to have, things that you might be tradin off while making design decisions.
eg:- availability, ~~latency~~ wait & see, throughput, redundancy, Consistency

3] Actual sys. components → Tangible things that you hv/ implement in a sys. Bread & butter of sys.
eg:- load balancer, proxy, cache, rate limiting, leader elecⁿ

4] Tech → Real, existin products or services that you can use in sys either as actual components or to achieve a certain charac. in a sys. These are real tools
eg:- Zookeeper, XCP, Engine X, Reddits, Amazon S3, Google Cloud storage

Date:
M T W T F S S

DATA STRUCTURES & ALGORITHMS

I SYSTEMS EXPERT

Design fundamentals → foundational knowledge reqd. for systems design interviews

Data structures → foundational knowledge reqd for coding interviews

eg of design fundamentals: SQL, server, cache, polling, load balancer, HTTP, database, hashing, replicaⁿ, client, proxies, Nginx, availability, leader elecⁿ, P2P

IA Client-Server Model/Architecture/Paradigm

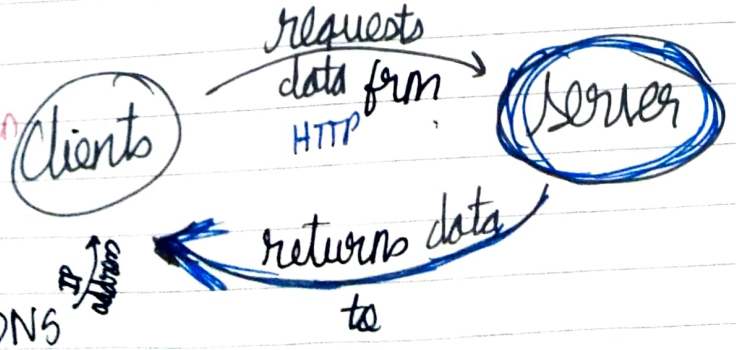
Foundaⁿ of modern internet, helps us understand how computers speak to one another. Paradigm consists of clients requestin service, data frm servers and servers providin it all

What we do? ⇒ Type URL in browser and hit 'Enter'

What happens?

↳ single machine, or software can act as both client, server. eg: Node and Postman on PC

(i) Client: is something, a machine that speaks to the server.
(ii) Server: something else, a machine that listens to the client, listens for clients to speak and then speaks back to the clients



interact thru following network protocols ↓ DNS IP address

speech is done by sending of data, demand (demand a form type of data request for data back)

(iii) Browser : Client
Website : Server
(ones which have server)

⇒ NOTE: A website (same) may have diff. servers in diff. loca^{n}s. eg:- Netflix India has a diff. server from Netflix USA.

(iv) Browser doesn't really know what the server is. All that it knows is that it can communicate with server. Doesn't really know what the server represents. It just requests info from it and does stuff based on info recd. from server.

(v) To start communica^n with a server, a browser 1st sends a DNS (Domain Name Server) query to find out IP address of server.

DNS query → special request going to a predetermined set of servers asking for IP address of a server

IP address → unique identifier for a machine. All comps. connected to internet have ways to find out these IP addresses or discover routes to those addresses
They can send packets of data/info in form of bytes to IP address.

Analogy :- IP address is like a mailbox that some entity has granted to a machine
eg:- Algo Expert's IP address has been granted to it by its cloud provider - Google Cloud Platform. It reserved an IP address for Algo Expert.

eg:- Thus, browser makes DNS query for AlgoExpert.io receives back an IP address and thus starts communica with the server.

(vi) HTTP: way to send info that comp. can understand.

exercise: type "dig algoexpert.io" → does a DNS query and returns IP address of algoexpert.io

→ when browser sends HTTP req. to ~~ge~~ ge server, it basically it sends a bunch of bytes / char that are gonna get packed into some special format and sent to servers. This request also contains IP address of your PC (a.k.a. source address)
When server receives source address, it knows that ~~to~~ on which IP address it needs to send a response to.

(vii) Ports - Servers usually listen for requests on specific ports. Any machine having a distinct IP address has 16000 ports that progs. on the machine can listen to.
On communicating with machine, you (client) gotta specify what port you wanna communicate on

Analogy → IP address: Mailbox to an apartment complex
Ports: actual apartment no. that the mail (client req.) arriving at mailbox has to route to.

Most clients know the port that they should use

depending on the protocol that they're trying to speak to server with.

Protocol	Port used by client
HTTP	80
HTTPS	443

exercise: netcat: allows you to read from/write to network connec. using certain protocols

Terminal Window 1	Terminal Window 2
nc -l 8081	nc 127.0.0.1 8081
↑ says that listen to stuff happening on this port	special IP address that always pts to your local machine (local machine's IP address
READING DATA	WRITING DATA
↑ Anything that you type in terminal window 2 appears.	↑ this terminal is entering a communi. channel with the machine at this IP address at port 8081

(viii) Thus, once server receives req., it is able to read it cuz it understands the HTTP format
Server understands that when you go to ae.io, you're trying to see the HTML of "ae" and so it returns a response viz. the HTML of 'ae' to browser which receives response and renders the HTML on the page for you

↑ turns HTML code into text, imgs, multimedia, interactivity effects, etc.

Addi"al info:

IP address: IPv4 addresses consist of 4 nos. separated by dots a.b.c.d where all 4 nos $\in [0, 255]$

i) 127.0.0.1 ⇒ Your own local machine a.k.a localhost

ii) 192.168.x.y ⇒ Your private network.

Eg:- Your machine and all machines on your n/t. wifi network have the 192.168 prefix

NETWORK PROTOCOL

msgs sent over wire or network ≡ msgs sent over the internet from 1 machine to another machine

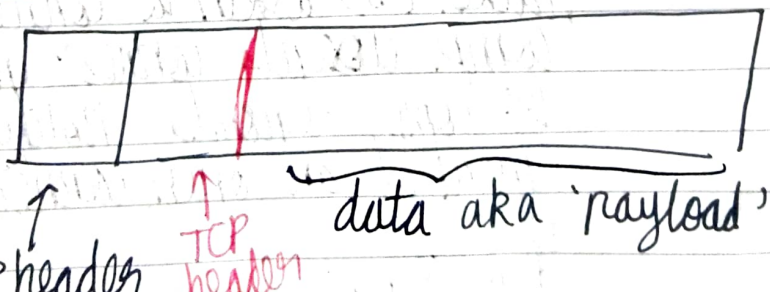
Network protocol consists of:
- types of msgs
- struc.
- order
- response to a msg, what shud it look like if present
- rules on when msgs can be sent to 1 another

3 main protocols:

1) IP → Internet Protocol

i) Modern internet effectively runs on IP. When 1 machine (client) interacts with another (server) and sends data to it the data is sent in form of IP packet (fundamental unit bldg blocks of data sent b/w machines i.e. communication)

ii) IP packet → made up of bytes



⇒ IP packet

iii) IP header - At beginning of packet. Contains imp info → source IP address, destination IP address (machine sending data) (machine data will go to)

destinaⁿ ≡ dstn

total packet size: version of IP that packet is operatin by (current used → IPv4, IPv6). Based on IP version, packet might looks diff
header size : 20 to 60 bytes

iv) Size of IP packet ≈ 2^{16} B ≈ 65 KB
∴ info sent as multi-IP packets across machines. Order in; assurance that packets will be sent is kinda fked.
That's why, TCP is req-d.

2) TCP – Transmission Control Protocol
Implemented in the kernel, which exposes sockets to apps, that they can use to stream data thru open connecⁿ
i) Built on top of IP. Solves order, assurance issues of IP packets in an error-free, uncorrupted way
ii) Used in web apps mostly
iii) TCP header → present in data por" of packet
Contains TCP info like order of packets
eg:- browser wants to connect to a dstn server such as Google. Browser is first gonna create a TCP connecⁿ with dstn comp. / server thru a handshake
iv) Handshake : Special TCP interacⁿ where 1 comp communicates with other by sending packet(s) askin to connect. Receiving comp. accepts connecⁿ invite.
1st comp responds notifyin establishment of an open connecⁿ b/w 2 of em.
Now machines can exchange data.
v) If 1 of the machines doesn't send data in a given time period, connecⁿ can be timed out

devpr ≡ developer

vi) A machine can end comm. by sendin a msg notifyin its intenⁿ and then TCP connecⁿ is done

vii) Thus, TCP is a more powerful, funcⁿal wrapper around IP.
It lacks robust framework that developers use to define comm. channels b/w client-server in a sys. as its just data that fits into IP packets underlyin.

3) HTTP - Hypertext Transfer Protocol

i) Built on top of TCP
ii) Introduces higher lvl abstracⁿ (req-respons paradigm) above TCP
iii) 1 machine req. 1 machine responds. Makes it easy for devprs to create easy to use, robust syss
iv) Req. : Machine that wants to interact with other machine sends this. Hv lotta props. defined by HTTP
v) Resp : Destn machine's reply to a requ
vi) Req., Resp → can be thought to be similar to objs. with imp fields. props. that describe em.

Can be visualized (not exactly like this in reality) as

```
const httpReq = {
  host: 'localhost',
  port: 8080,
  method: 'POST', //GET, PUT, DELETE, OPTIONS, PATC
  path: '/payments',
  headers: {
    'content-type': 'application/json',
    'content-length': 51
  },
  body: '{"data": "This is a low JSON format data piece"}'
```

describe destn server

describe purpose of req. Hvjst guidelines only to your server as how we em.

provides data → server
retrieve data
delete data

Client sends data to server which has various paths which dictate what kinda logic and execution req eg: auth

more used to POST payments i.e. pay

```
const httpResponse = {
    statusCode: 200,
    headers:
        {
        'access-control-allow-origin': 'https://www.goog
        'content-type': 'application/json',
        },
    body: '{}'
}
```

describes type of resp. status codes are also like guidelines that you or alter as per reqmt
eg: - 404 status code = requested data piece not found

Path $\xrightarrow{\text{contains}}$ logic $\xrightarrow{\text{gets executed}}$ as per path provided in req

Headers → collection of k-v pairs containing imp. meta data or info abt req.

eg: 403 status code → data you're req. in is forbidden

NOTE status code → describes type of ~~resp~~ response status codes are like guidelines that you can alter as per requirement

eg:- 404 status code = requested piece of data not found.

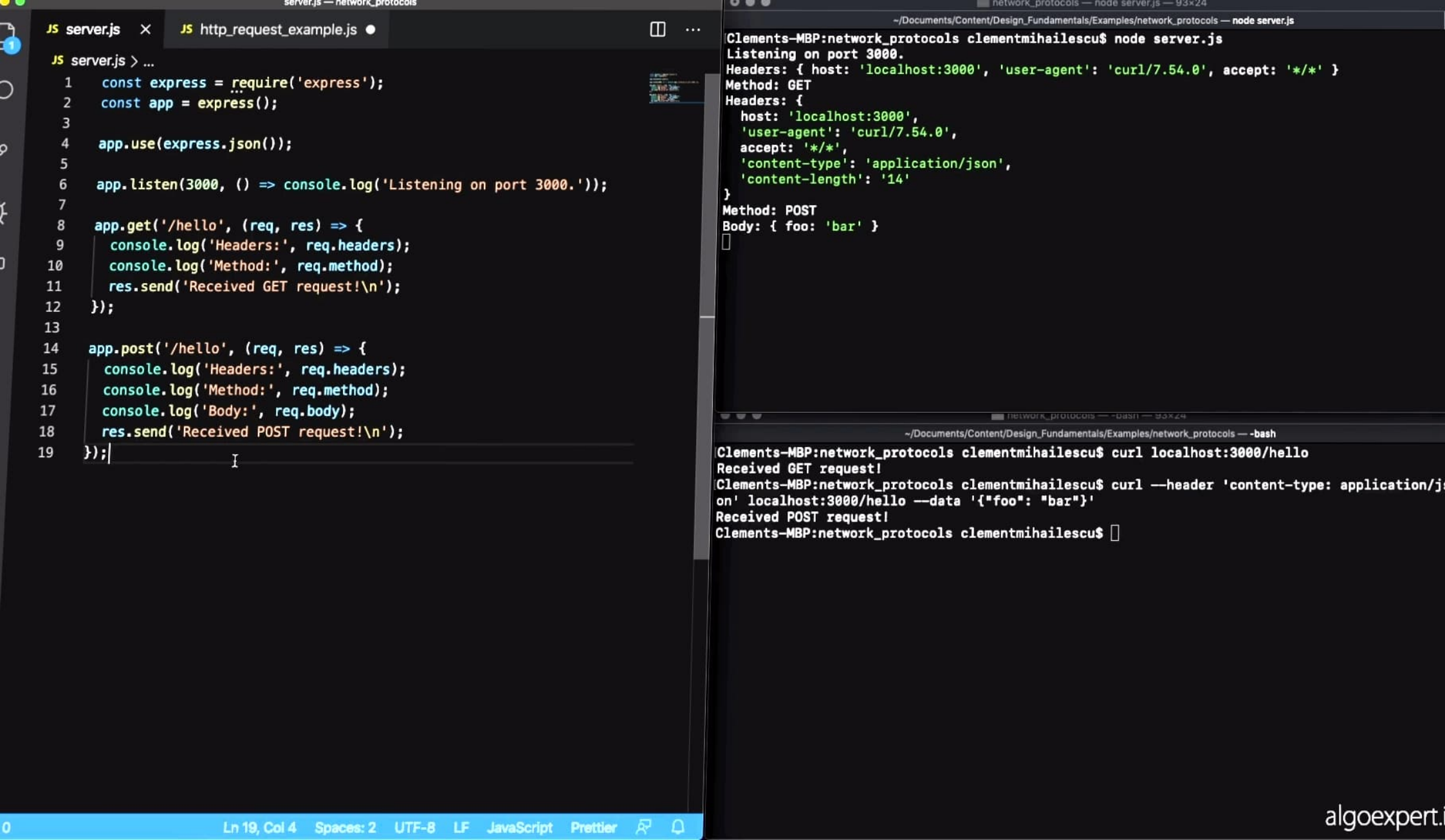
Typical HTTP req. schema

host : string (eg : facebook.com)
port : integer (eg :- 3000, 80, 43)
method : string (eg :- GET, PUT, POST, DELETE, OPTIONS, PATCH)
headers : pair-list (eg :- "Content-Type" ⇒ application/json")
body : opaque sequence of bytes

Typical HTTP resp. schema

error ↑ ↗ all ok

status code : integer (eg :- 404, 200)
headers : pair list (eg :- "Content-Length" ⇒ 1238)
body : opaque sequence of bytes


server.js — network_protocols
server.js
http_request_example.js
server.js > ...
1 const express = require('express');
2 const app = express();
3
4 app.use(express.json());
5
6 app.listen(3000, () => console.log('Listening on port 3000.'));
7
8 app.get('/hello', (req, res) => {
9 console.log('Headers:', req.headers);
10 console.log('Method:', req.method);
11 res.send('Received GET request!\n');
12 });
13
14 app.post('/hello', (req, res) => {
15 console.log('Headers:', req.headers);
16 console.log('Method:', req.method);
17 console.log('Body:', req.body);
18 res.send('Received POST request!\n');
19 });
Ln 19, Col 4
Spaces: 2
UTF-8
LF
JavaScript
Prettier
network_protocols — node server.js — 93×24
~/Documents/Content/Design_Fundamentals/Examples/network_protocols — node server.js
Clements-MBP:network_protocols clementmihailescu$ node server.js
Listening on port 3000.
Headers: { host: 'localhost:3000', 'user-agent': 'curl/7.54.0', accept: '*/*' }
Method: GET
Headers: {
host: 'localhost:3000',
'user-agent': 'curl/7.54.0',
accept: '*/*',
'content-type': 'application/json',
'content-length': '14'
}
Method: POST
Body: { foo: 'bar' }
network_protocols — -bash — 93×24
~/Documents/Content/Design_Fundamentals/Examples/network_protocols — -bash
Clements-MBP:network_protocols clementmihailescu$ curl localhost:3000/hello
Received GET request!
Clements-MBP:network_protocols clementmihailescu$ curl --header 'content-type: application/j
on' localhost:3000/hello --data '{"foo": "bar"}'
Received POST request!
Clements-MBP:network_protocols clementmihailescu$
algoexpert.

Database → servers with long life that interact with the rest of your app thru network calls, with protocols on top of TCP or even HTTP

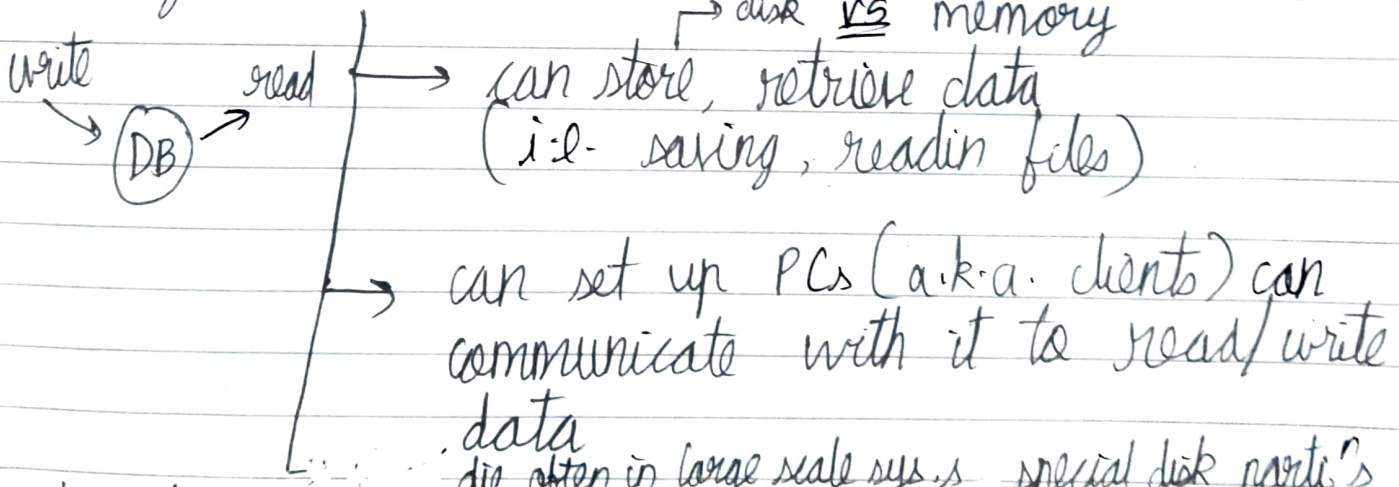
STORAGE

→ 1) Every system requires storage.
eg:- for storin user info, metrics

→ 2) Database = a machine that helps storing and retrieving data

synonymous word pairs w.r.t. data:
- setting ≡ writing ≡ recording ≡ storing
- getting ≡ reading ≡ querying ≡ retrieving

In most cases database (DB) is actually just a server
eg:- even a PC can be made to act as a DB



a.k.a volumes are used by DB to access volumes of data
die often in large scale sys.s special disk partn's even if machine dies (crashes permanently)

→ 3) Persistance of data in a DB
↳ non-volatile storage → can retrieve info after being power cycled

Disk - writin data to disk persists even if the DB itself crashes / faces some outage

eg:- excludin extreme issues, any file you save on PC, persists even after PC shuts down or crashes.

↳ needs const. power to retain data

↳ RAM - volatile storage

Memory - Data stored in this does not persist if DB goes down

eg:- any data stored in variable in your server's DB does not persist if DB server goes down

eg:- In a mobile, say 64GB storage, 8GB RAM

↳ 64GB → disk

↳ 8GB → memory

NOTE Reading data from memory is faster than reading data from disc.

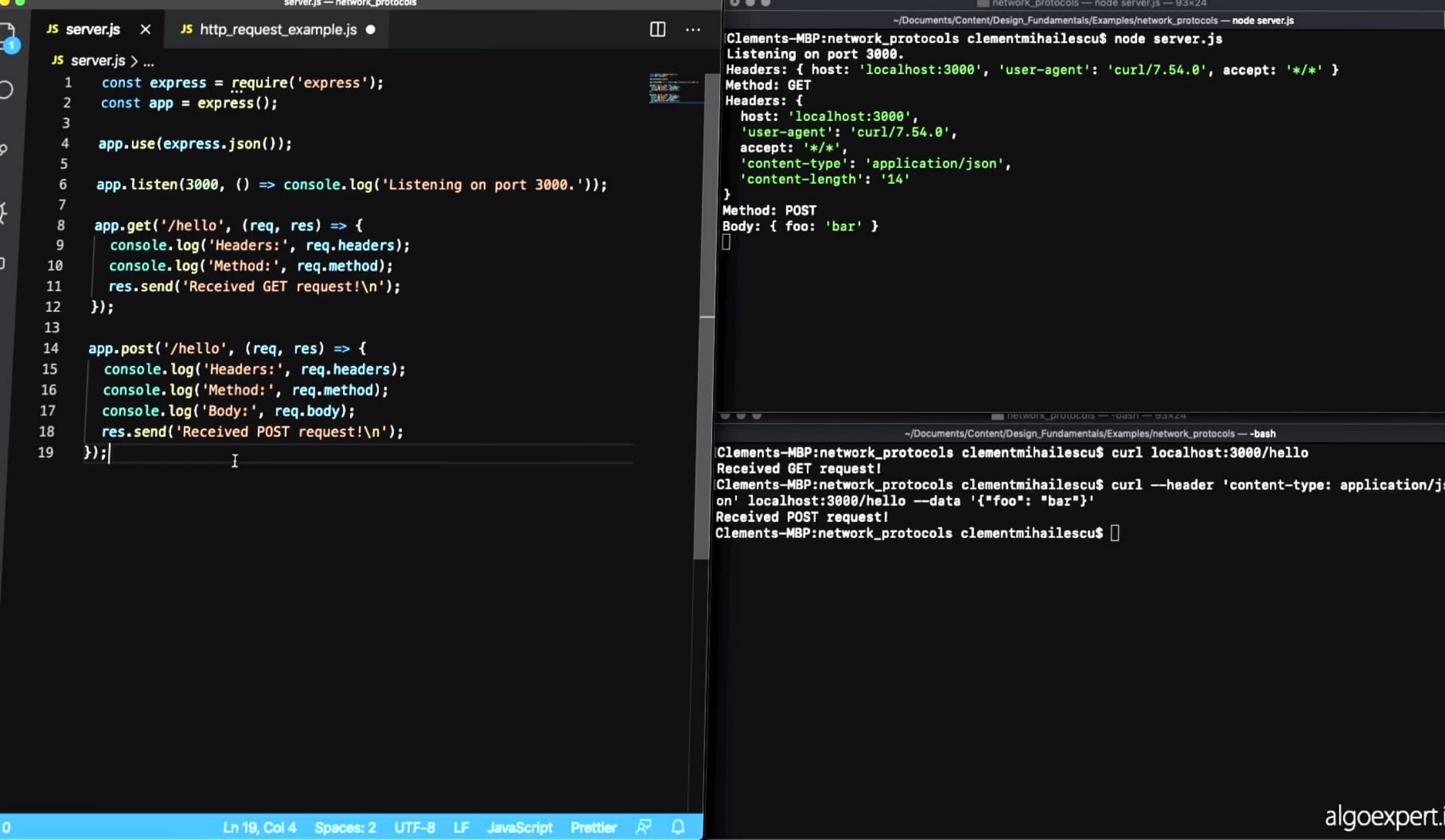
There are a lotta DB offerings to give options based on performance, data security

When DB goes down, ~~because~~ depending on how critical a part of the entire sys the DB is, does the system crash or persist through

Distributed storage → storing data on multiple machines.

↳ M-1 → split data up

↳ M-2 → replicate data on machines

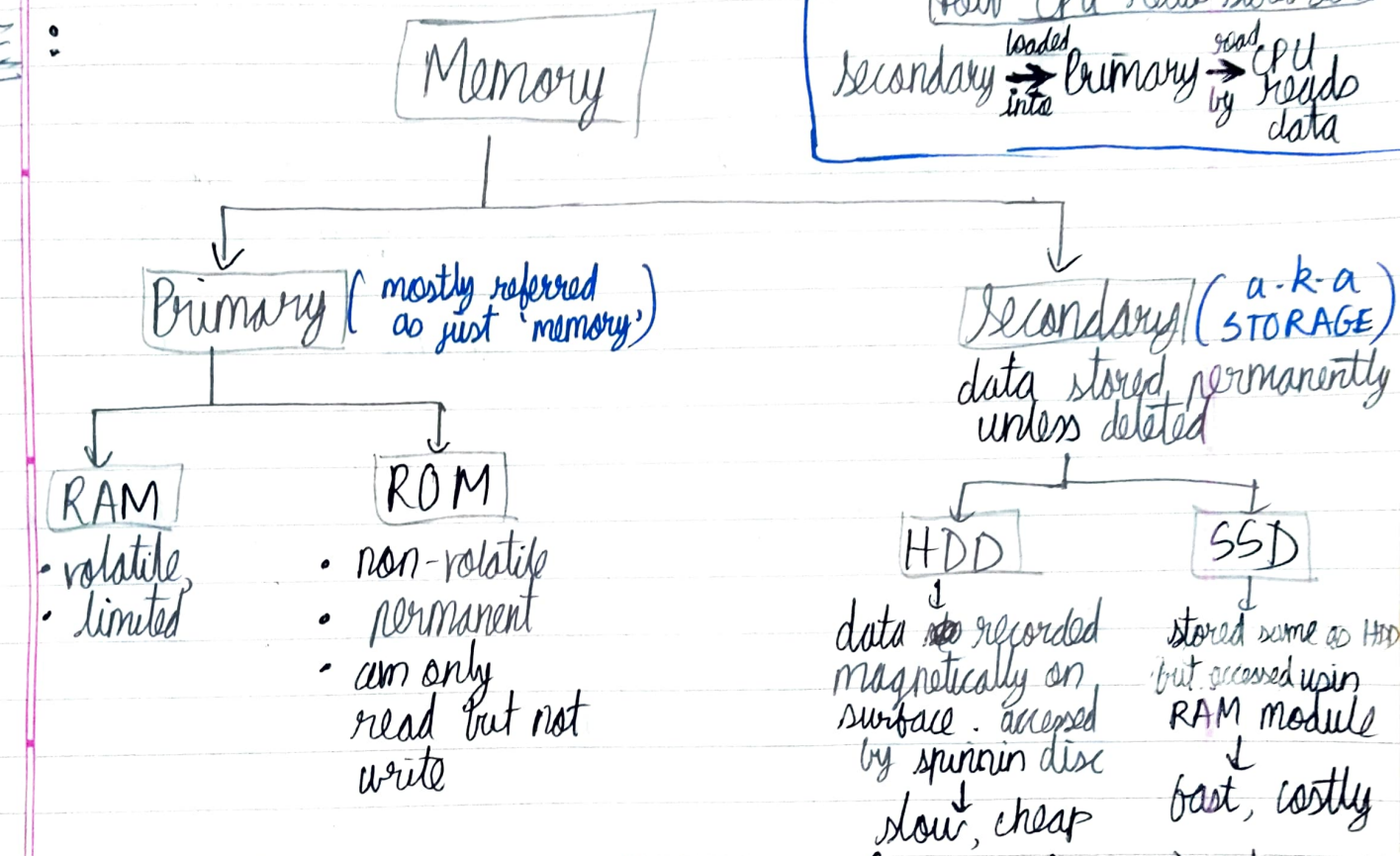

server.js — network_protocols
server.js
http_request_example.js
server.js > ...
1 const express = require('express');
2 const app = express();
3
4 app.use(express.json());
5
6 app.listen(3000, () => console.log('Listening on port 3000.'));
7
8 app.get('/hello', (req, res) => {
9 console.log('Headers:', req.headers);
10 console.log('Method:', req.method);
11 res.send('Received GET request!\n');
12 });
13
14 app.post('/hello', (req, res) => {
15 console.log('Headers:', req.headers);
16 console.log('Method:', req.method);
17 console.log('Body:', req.body);
18 res.send('Received POST request!\n');
19 });
Ln 19, Col 4
Spaces: 2
UTF-8
LF
JavaScript
Prettier
network_protocols — node server.js — 93×24
~/Documents/Content/Design_Fundamentals/Examples/network_protocols — node server.js
Clements-MBP:network_protocols clementmihailescu$ node server.js
Listening on port 3000.
Headers: { host: 'localhost:3000', 'user-agent': 'curl/7.54.0', accept: '*/*' }
Method: GET
Headers: {
host: 'localhost:3000',
'user-agent': 'curl/7.54.0',
accept: '*/*',
'content-type': 'application/json',
'content-length': '14'
}
Method: POST
Body: { foo: 'bar' }
network_protocols — -bash — 93×24
~/Documents/Content/Design_Fundamentals/Examples/network_protocols — -bash
Clements-MBP:network_protocols clementmihailescu$ curl localhost:3000/hello
Received GET request!
Clements-MBP:network_protocols clementmihailescu$ curl --header 'content-type: application/j
on' localhost:3000/hello --data '{"foo": "bar"}'
Received POST request!
Clements-MBP:network_protocols clementmihailescu$
algoexpert.

Consistency - A concept in storage referring to the staleness or up-to-dateness of data

eg:- on accessing data from a DB, how fresh/up-to-date is the data that you get

How CPU reads stored data:
Secondary → (loaded into) Primary → (read by) CPU reads data

NOTE :



- ROM contains a prog. called BIOS (Basic I/O sys) which microprocessor (i.e. computer CPU) to load OS from HDD into RAM whenever PC is turned on. Newer motherboards use UEFI (Unified Extensible Firmware Interface)

- Analogy :- (i) MEMORY → a work desk
  (ii) storage → Secondary memory → desk drawers where files are stored
  (iii) memory → Primary memory → table top of desk
  (iv) Prog in memory → tools, things on desk (easy, fast to access)
  (v) Prog in storage → files in desk drawers that you gotta put on table top to access (slow to access)
  (vi) Loading a prog. from secondary memory → open a drawer, remove required file and put it on table top to use it

Latency & Throughput

A] Latency and Throughput → 2 most imp measures of the performance of system

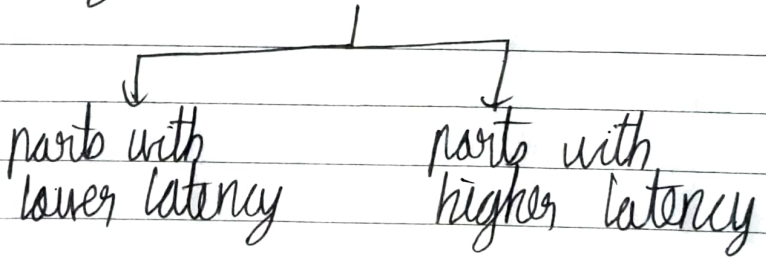
LATENCY:

i) Latency is basically how long it takes for data to traverse a sys. i.e. get frm 1 pt. in sys to another pt. in sys.

eg:- latency of network req : how long it takes for 1 req. to go frm client to a server and then the processed response to go from server to client

eg 2:- time reqd. by a server to read data frm disk

ii) Diff parts of a sys hv diff. latencies
∴ while designin a sys we'll face a trade-off while optimizin it cuz we'll have



iii) Some comparisons

Part of sys + its fⁿ	Latency
Readin 1MB from memory (RAM)	250 μs
Readin 1MB from SSD	1000 μs
Sendin 1MB over 1Gbps network	10^4 μs → (sendin to comp. wtin next to the i.e. dist. is not considered)
Readin 1MB from HDD	2×10^4 μs
Sendin packet (≈ 1055 B) over network from California to Netherlands and then back to California	15×10^4 μs

Date:
M T W T F S S

eg: of sendin data over a network → API network req.

eg of readin data frm memory → reading a variable in code

Takeaways:

i) Dependin on network and your PC hardware, sometimes sendin, receivin data over network is faster than readin data frm HDD

ii) Sendin data around the world takes a lot longer than any other meth.
Reason: req. gonna ~~get fucked up~~, get converted into small packs (into binary data) → converted into freq. modulated radiowaves → sent to cell towers thru cables → bounced to satellite → passed around the world thru satellite comm. → passed to destina" all towers → passed to destina" and reconverted back to original form

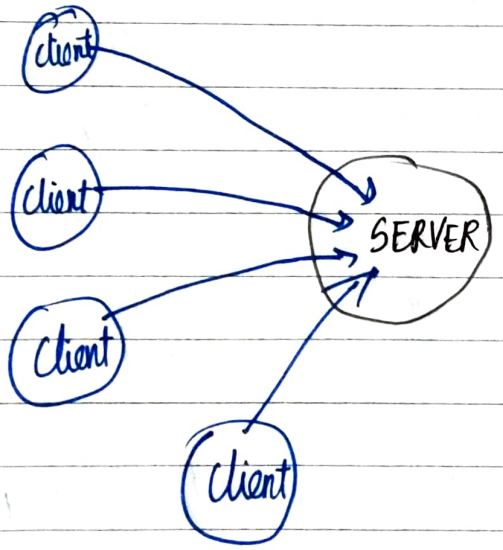
iii) Optimizin a sys → ↓ its overall latency

eg:- Video games gotta have to have really low latency and Lag → delay in ac" passed from 1 user → server → receiving user. These ac"s are passed as network req.s to 's server so if you're pretty far away frm server, your PC will take more time to make network req/receive responses frm server

eg:- Websites → It's peace if they have low latency because in most cases, their priority is getting the info displayed to be accurate, uptime to be cont. 24x7

THROUGHPUT (TP)

Throughput is basically how much work a machine can perform in a given amt. of time → how much data can be transferred frm 1 pt. in sys to another pt, in given time
unit :- bytes/sec
eg:- 1Gbps network → network can support 1Gb per sec.



TP is how many req.s (each hvin some data) can this server handle in given time → how much data it can let thru per sec.

To optimize sys. → pay to ↑ TP (case(i))

↳ (case(ii)) in case just ↑ TP doesn't solve prob. as you might have a server hvin 10^3 or 10^6 req. issued to it per sec so no matter how much we ↑ TP, we'll still hv a bottleneck (only some data is let thru server) at server

Case (ii) soln : Have diff. servers for req.s so they don't clog at bottleneck

IMP : Latency and throughput are not correlated

eg:- you might have parts of sys with reqd low latency (fast data transfer) → but then if you hv a part with v. less TP, then our advan. due to low latency in other parts gets cancelled out as req.s gotta wait (clog) at this part cuz of low TP

∴ You can't make assumpⁿs on latency or TP based on each other.
They affect each other but don't determine each other

AVAILABILITY

Fault Tolerance :- How ~~tolerant~~ resistant is a sys to failures. What happens if a server in sys fails? DB fails ? → is sys gonna still be operaⁿal

Availability → % of a given period of time (eg:- month, year) for which the sys. is atleast operaⁿal enough to get its primary fⁿs satisfied

Less availability (LA) → less uptime of a sys. during given time period
- → lose customers (existing)
- → lose the prospect of getting new customers → money lost
- bad publi.

Sys.s that gotta hv. high availability → any amt. of downtime costs a lot

→ DUE TO THEIR FN
eg:- sys. supportin airplane software

→ DUE TO THE MAGNITUDE OF PEOPLE ACCESSIN
eg:- Youtube, Facebook

→ CLOUD PROVIDERS
All services, platforms (dependent on cloud providers) businesses get heavily affect if cloud provider sys. fails

service ≡ sys.

Date:
M T W T F S S

eg of cloud providers : Azure, AWS, Google Cloud, Oracle Platform

abbr.

Measuring availability (abbt)

Availability is measured in terms of % of sys.'s uptime in a given year. (90% abbt → sys. in up. 90% of the time in a yr)

Most powerful services, sys.s have ~~abbr~~ abbt really HA. i.e. 99.9%, 99.99% etc.

Nines → Percentages with the no. 9

eg:- 99% abbt → 2 nines of abbt

Abbt	Downtime /yr
2 nines (99%)	87.7 hr
3 nines (99.9%)	8.8 hr
4 nines (99.99%)	52.6 mins
5 nines (99.999%)	5.3 mins

Highly available sys → abbt > 5 nines abbt.

SLAs and SLOs (abbt. guaranteed explicitly

SLA → Service Lvl Agreement → Agreement b/w service b/w service provider and customers / end users of service that guarantee customers amount of abbt, error free utility and some other objectives (known as SLOs → service lvl objective), failing which service the service provider pays back the customer some % of their fees

SLA

eg:- AWS (provider) Netflix (user)

SLO 1 → 1) Netflix get 99.9999% abt.
SLO 2 → 2) Netflix get x errors while using my service

∥

If AWS fails to meet SLOs, it returns 10% of the service fees Netflix pays to it every month

NOTE: Achieving higher abt is really difficult and may accompany tradeoffs like

- higher latency
- lower thruput

As a sys. des. gr, you gotta decide which part of sys you want high abt. (HA) and which part you don't really need HA.

eg:- Stripe (payment service for business

→ Core services → handling payments, charging customers

These gotta have high HA else Stripe, as well as its client businesses, platforms lose lotta money

→ Business monitor dashboard → used by client platform to use business stats.

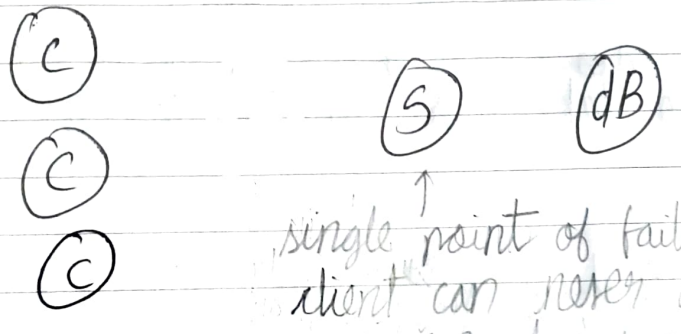
Doesn't really require HA, something that if fails, doesn't cause catastrophic losses

How a sys. is made "AVAILABLE"

1 Single pts of failure (parts of the sys. which, on failing cause entire sys. to fail) should be removed → for this we use **redundancy** while designing the sys.

Passive Redundancy → act of duplicatin, or triplicatin or multiplyin even more, certain parts of sys.

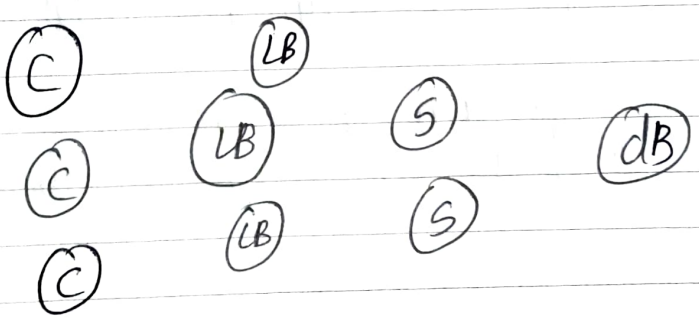
Simple sys ⟹



C - client
LB - load balancer
S - server
dB - database

single point of failure as, if it fails client can never access database so purpose of sys. has failed

Redundant sys ⟹



Load balancer → to ~~reduce~~ equally distri. load across all servers (prev. 1 server from becomin too overloaded & actin as single pt. of failure

↓

Multiplied load balancers in this sys to avoid 1 LB from gettin too overloaded

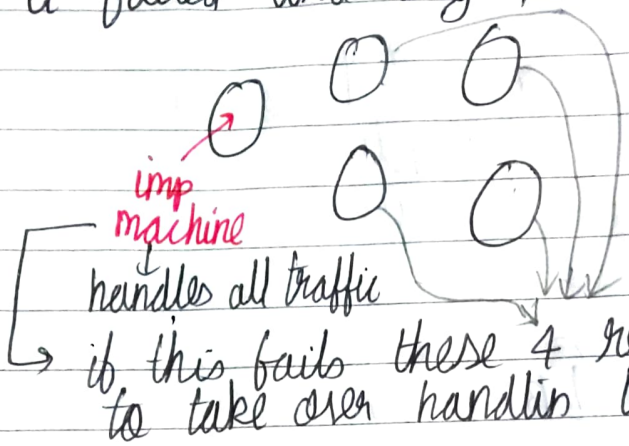
→ duplicated

eg:- twin engine sys. in airplane

Date:
M T W T F S S

Active redundancy - Multip machines that work together in such a way that only 1 or few of the machines are gonna be typically handlin traffic or doin work ('imp machine'). If an 'imp machine' fails, the others machines know it failed and they'll take over

eg:-



2] Have rigorous processes in place to handle to handle sys failures cuz its possib. that these failures might req. human interven^n

∴ Have these backup op^{n}s in mind while sys. des. to get a crashed sys., up in runnin in proper timeframe.

CACHING

1] Caching in algos → To avoid doing same ops. especially computationally complex ops. that take a lotta time multiple times
∴ caching used to improve time complexity (T.C.)

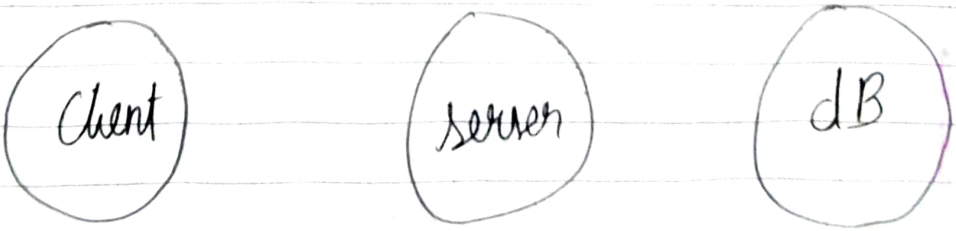
2] Caching in sys. → Used to reduce (improve) the latency of a sys. (speeds up the sys.)

3] CACHING → Storing data in a locn that's diff from the original data locn s.t. it's faster to access this data frm memory

It's a way to design a sys. s.t. if we were originally using I/ops. or data transfers that take a lotta time (eg:- Network requests) the sys. is designed is s.t. we do diff types of ops. or data transfers that are faster.

I Caching in diff places in a sys.

eg:-



Regular op. → client makes net. req. to server to get some data from dB and then data transferred from dB → server → client

1) Caching at client level - Client caches a data value s.t. it no longer needs to go to server to retrieve it

2) Caching at server level - You want client to always interact with server but maybe the server doesn't need to go to dB for data retrieval each time
∴ Go the dB once → get data → cache in server → transfer data to client as per net. req

3) Cache in b/w components - eg:- cache in b/w server and dB

not in our control when we des. sys. → 4) Caching at hardware level - eg:- CPU caches - make it faster to retrieve data from memory.

5) Caching occurs by default, at many other such kinda lvls. of sys.

II

Advantages / Utility of Caching

Caching speeds up sys → 1) Avoid large magnitudes of net. reqs. → If the client wants to access the same data again and again, no new net req. (client → server → dB → server → client) done each time as we cache reqd. data so client doesn't need to go to dB each time
Caching done at client or server lvl.

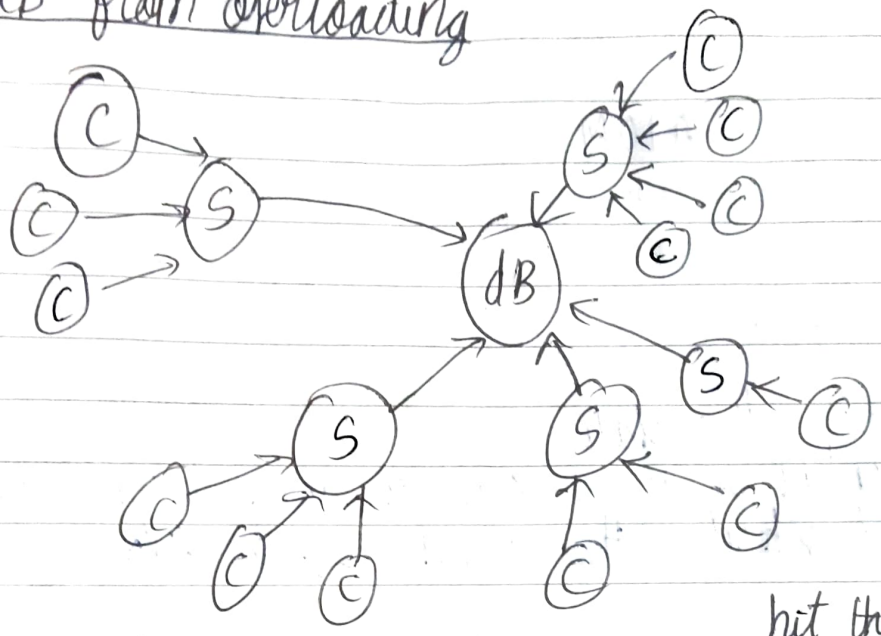
Caching speeds up sys. → 2) Avoid redoing a computationally long, bad TC, complex operation each time
eg:- say on net. req. from client → server,

everytime, this time - consumin algo runs. To avoid performin this op multip times, caching is done.

3) dB optimized

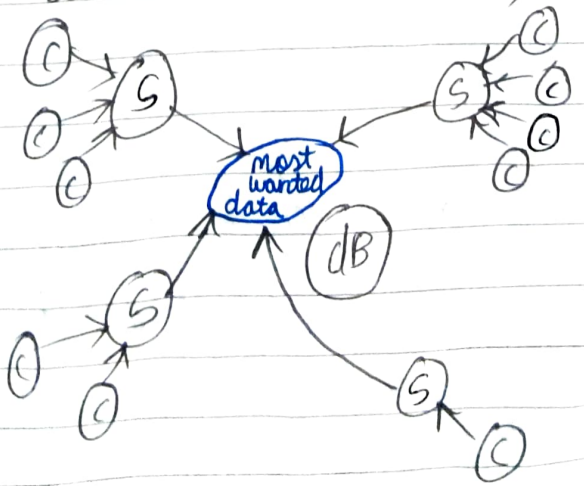
Prev. dB from overloading

eg:-



Multip clients across multip servers hit the dB with the same net. req. eg:- Each client makes indiv. net. req. to server to get Instagram profile of a popular celeb.

Indiv. req. are fast so speed is not the concern. We use cachin to prev. same data being read from dB 10^6 times (or somethin like that)
∴ a cache is created, outside dB, for that data



If we don't cache, too many req.s to dB for this data + in gen. other data accumulate and overload dB

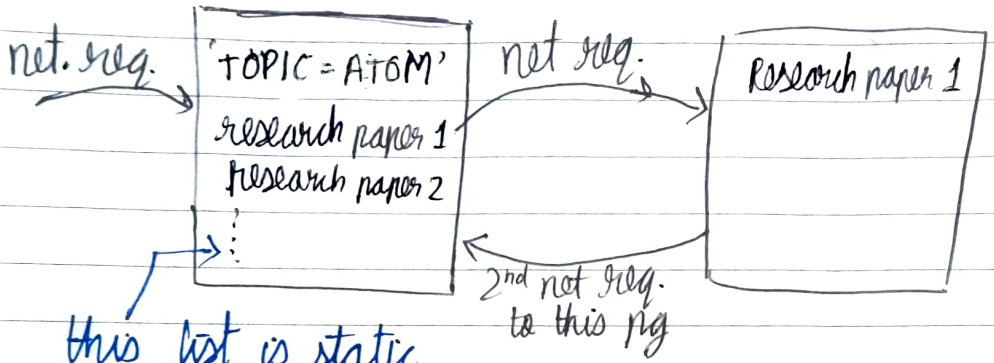
k-V = key value

Date: M T W T F S S

III

CACHING in ac^n

1) Reloadin static (immutable) components or content in a website
eg:- ResearchGate : every once a yr papers published on site



content so if we cache it after 1st net req. to this pg of website, the 2nd net req. to list pg will be a lot faster

2) Runnin code on algoexp. (AE)

i) Running testin code takes ≈ 1 sec.
ii) for customer code → runnin testin code reqd.
for AE soln code → w.r.t this code is correct code ie- it'll past all test cases so no need to run, test

Cache stored → server level
OR
→ detached from server in an indep. comp.

Redis (popular in-memory dB) ⇒ a k-V store used for caching.

All AE solns. codes pre-stored with a unique 'key' in this Redis cache

Page

Date:
M T W T F S S

Caching in ac^n

When a code is called to run on AE → http req. made to AE servers → req (bunch of bytes) hashed down to a single signature (key)

Cache with Redis (k-v) store is checked whether this key is pre-stored

if pre-stored → code called is AE soln code. no need to run. Takes < 1 sec to show result

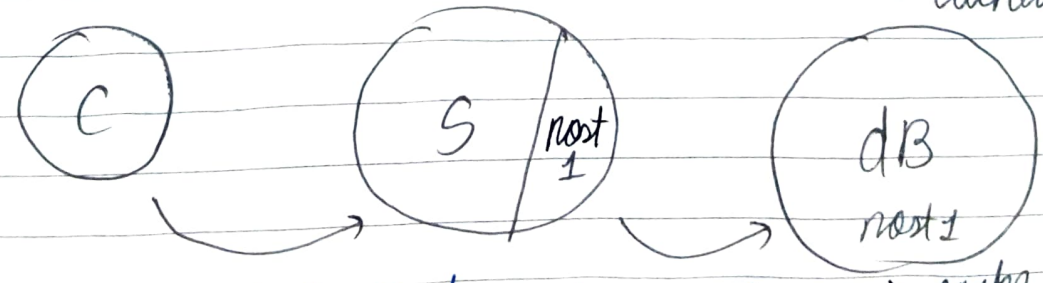
if not pre-stored → Code called is customer's code so run it. Takes 1 sec. to give result

Until now we discussed cachin for a sys. that needs to read data.

IV Cachin and Sys. where users can read, write, edit data.

eg:- Readin, writin, editin posts of FB or LinkedIN

1) When you write a post to FB, say that using net reqs. that post is → stored in dB
→ cached in server



2 sources of truth for given post → cache in server
→ dB

2) When you now wanna update your post, dependin on type of cachin in server : post stored in the

JS server.js > app.get('/nocache/index.html') callback > database.get('index.html', page) callback

```
const database = require('./database');
const express = require('express');

const app = express();
const cache = {};

app.get('/nocache/index.html', (req, res) => {
  database.get('index.html', page => {
    res.send(page);
  });
});

app.get('/withcache/index.html', (req, res) => {
  if ('index.html' in cache) {
    res.send(cache['index.html']);
    return;
  }

  database.get('index.html', page => {
    cache['index.html'] = page;
    res.send(page);
  });
});

app.listen(3001, function() {
  console.log('Listening on port 3001!');
});
```

JS database.js > ...

```
const database = {
  ['index.html']: '<html>Hello World!</html>',
};

module.exports.get = (key, callback) => {
  setTimeout(() => {
    callback(database[key]);
  }, 3000);
};
```

Date:
M T W T F S S

dB → updated instantly
dB → updated async. ly.

Write-thru cache	Write-back cache
1) dB and cache both updated instantly when you edit post	1st cache is updated instantly Then dB is updated async. ly → updated after a period of time eg. 5 hrs, 30 days etc. → once cache is full and you gotta evict stuff outta cache
2) Takes time as you still gotta go to dB and come back for an edit	Takes less time as only cache update at the instant of edit
3) Edit secure in 2 sources of truth at the instant of edit	Edit secured only in cache at instant of edit ∴ if somethin happens (eg:- delete) to our cache before dB is async. ly updated, we lose the edit.

V

Staleness

eg:- YouTube comments sec^n.

You don't wanna update data asyncly as it might happen that:

- → someone posts a comment (SERVER 1)
- → other person reads it frm diff. server (SERVER 2)
- → 1st person edits comment (but dB not updated and his server's cache updated asyncly yet)
- → other person still sees original comment (unedited) and replies to unedited comment (STALE DATA)

Cache in each server accessing YouTube dB

as SERVER 1's cache ↓ don't want this to happen by SERVER 2

Cache in server 1 is STALE ∵ it hasn't been updated properly

Soln: Make cache outside servers into a common comp. for them all outside dB
∴ Each time cache is updated from server 1, server 2 accesses this latest updated cached data

NOTE ∴ While sys. des. ing just see if that part of sys. is ok with staleness or not and accordingly choose how you gonna cache the data

eg:- YouTube
- → view count on vids → lil bit of staleness is fine
- → comment sec^n → even a lil bit of staleness is NOT fine

Date:
M T W T F S S

VI

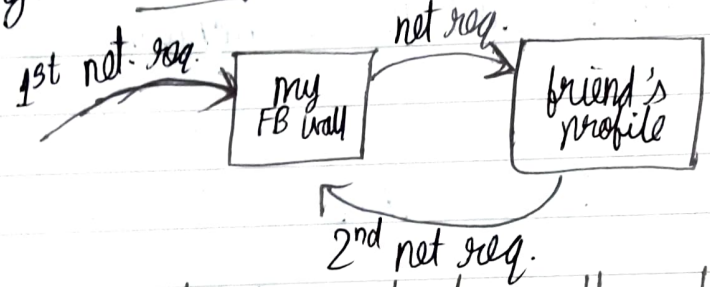
NOTE: WHEN TO USE CACHING?

✓ 1) Static (immutable) data → Cachin is easy to implement, fast while using

eg:- Research papers website

X 2) Dynamic (mutable) data → Cachin is tricky to implement as data stored at 2 diff loca^n's and you gotta make sure data in both loca^n's is in sync.

eg:- Facebook (FB) wall → not cached



On Facebook website, the data you see on your wall after 1st and 2nd net req. to FB wall page is diff.

✓ You're doin only 1 single thing eg: either of readin or writin data

X You're doin multip things → readin, writin, editin data.

✓ You don't care abt staleness of data or if you can invalidate (get rid of) stale data in caches in a distributed manner when dealing with a distributed sys.

VII

CACHING - EVICTION POLICIES

We might wanna evict (delete) data frm cache for several reasons. eg: data is stale, cache's storage is full.

To evict data from cache, we follow certain rules a.k.a evicⁿ policies

1) LRU policy → Least Recently Used (LRU) pieces of data in a cache are removed if you fr some way of tracking what pieces of data are LRU. cuz we care least abt em.

2) LFU policy → Least Freq. ly Used (LFU) pieces of data evicted

3) LIFO or FIFO → Data ejected on a Last In First Out or First In First Out basis

4) other ways.

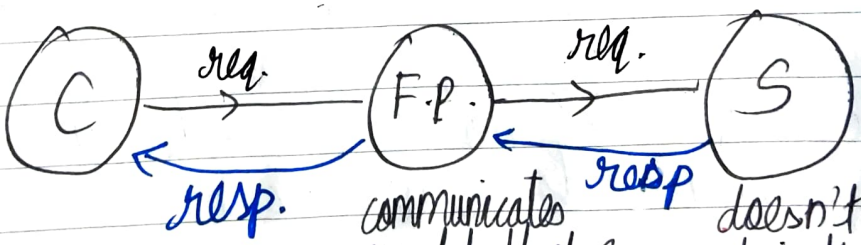
Choice of evicⁿ policy depends on kind of product you wanna do sys. des. for

PROXIES

PROXY:
Just a machine-server set up to hide the IP address and other details of an interacting machine (eg:- client, server, etc.)
2 types (primary) :-
i) Foward proxy (usually ref-ed to as - 'proxy')
ii) Reverse proxy

Foward Proxy (F.P) and Reverse Proxy (R.P)

1] **Foward Proxy (F.P.)** → on client's team
i) It is a server located b/w client/set of clients & server/set of servers
ii) It acts on behalf on clients/set of clients by hiding clients IP address from server and instead supplying its own IP address.
iii) When a client sends a req. to the server and this F.P. has been properly confg-ed by client, the req. goes as



server has no idea abt the client and that an F.P exists

iv) F.P. serves as a way to hide the identity of client by issuing req. from server by changing source IP address sent in req. to its own IP address (i.e. IPA) instead of clients IPA eg:- VPN s

v) Some types of ~~proxy~~ F.P.s make client IPA visib. to server in some way, but, typically original source IPA is replaced

NOTE: VPN → a fwd proxy b/w client and server that hides the client's identity
∴ client can access servers restricted to it. Thus, client can, for eg, access a website not availa. in his/her country but avail. in other.

2] Reverse Proxy (R.P.) → on server's team.
i) A server b/w client / set of clients and server / set of servers
ii) When a client interacts with a server. eg:- sendin a req.

client sends req. client thinks its sendin req. to destina server (i.e. doesn't know abt R.P.

req → R.P. req → server
resp ← resp ←

Sends its IPA to client who thinks that this IPA is server's IPA

Client has no idea abt server or the fact that an R.P. exists in middle

eg:- say google.com sets up an R.P.

google.com —req + DNS query to get back IPA of Google server→ R.P

client's req → Google server

client browser ← returns its own IPA + resp. ← resp.

Use of R.P.s → Powerful tool for sys. des.

1) eg:- Config. R.P. to filter out reqs. you want your sys to ignore

2) eg:- Log stuff, gather metrics → can be done by R.P.

3) eg 3:- R.P. can also cache stuff (eg:- HTML page) Thus, server doesn't get bothered a lot

4) eg 4:- As a load balancer → a server that can distri req. load amongst a bunch of servers

NOTE: 1] ~~Malicious~~ Malicious client → sends a fuckload of reqs. to a server to bring it down.
Now R.P. acts as load balancer will distri these reqs across all servers, thereby safeguarding sys. from malicious clients, viruses etc.

2] NginX is a popular web server that can be used as an R.P.

eg:- Code on app page:

1) We set up an R.P. for any req. comin to port 8081 of our web service
2) Each time a req. is directed to the endpt='/' at port 8081, the req. header on the req. known as 'systemexpert-tutorial' and set it to 'true'
3) Then we gonna prwd this req. to the server that pts. to localhost:3000 (its name is 'nodejs-backend')

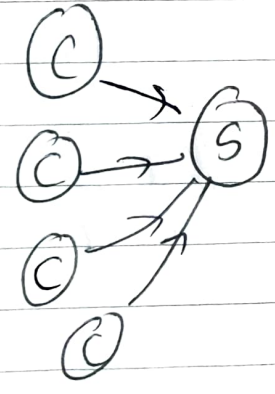
Load Balancers

Load Balancer → (i) A server that sits b/w a set of clients and set of servers which distributes load / reroutes traffic evenly across all servers in our set.
(balances workload)

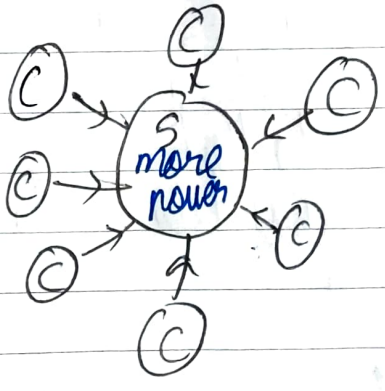
(ii) This prevents 1 server from getting too overloaded by too many client reqs.

(iii) Also clients do not know abt load balancers' existence so load balancer ≡ reverse proxy

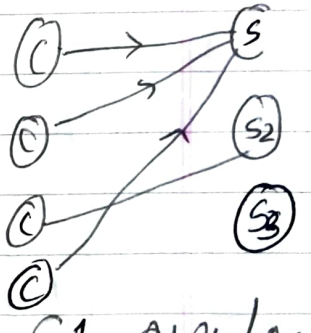
Case 1: Server overload due to lotta reqs.



Case 2: Vertical scaling: (power of server ↑ but still 1 old power, so overload

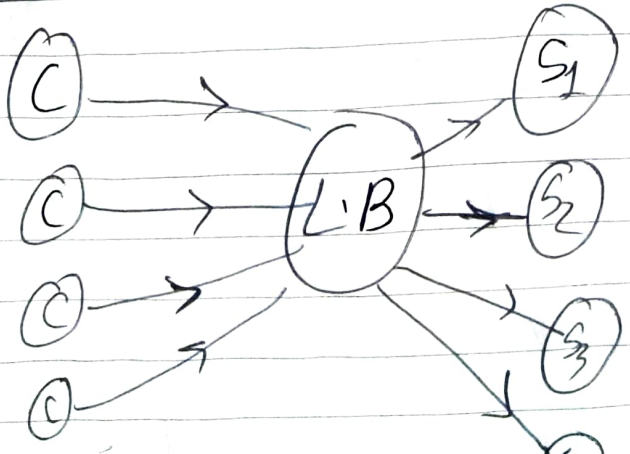


Case 3: Horizontal scaling
Multiple servers but still most client reqs to 1 server.



S1: overloaded

Case 4: Load Balancer



reqs evenly distri.
① No overload of a server
② Throughput ↑
③ Latency ↓ (no server clogged)

Other places where load balancin can be done:

1) b/w servers and dB
2) at D.N.S layer i.e. same service → multip servers so L.B. directs you to multip servers.

eg:- 'amazon' → single domain gets multip IP addresses (1 for .com, 1 for .in) for multip servers

amazon.com → Amazon D.N.S. in the US. → US server

amazon.in → Amazon D.N.S. in India → Indian server

eg2: Netflix US vs Netflix India

eg on opp page → we are accessin diff. IP addresses of google.com i.e. diff. servers and we get allotted the server related to our specific IP address (abbr. IPA) by the load balancer

NOTE Hardware L.B. → physical machine dedicated to load balancin

Software L.B. → not physical machines (eg:- a cloud services L.B. → cloud servers)
use: more power, customiza"

SOFTWARE L.B.

1) How does L.B. know which servers it can access?

(i) Sys. designers config. L.B and server to know abt each other

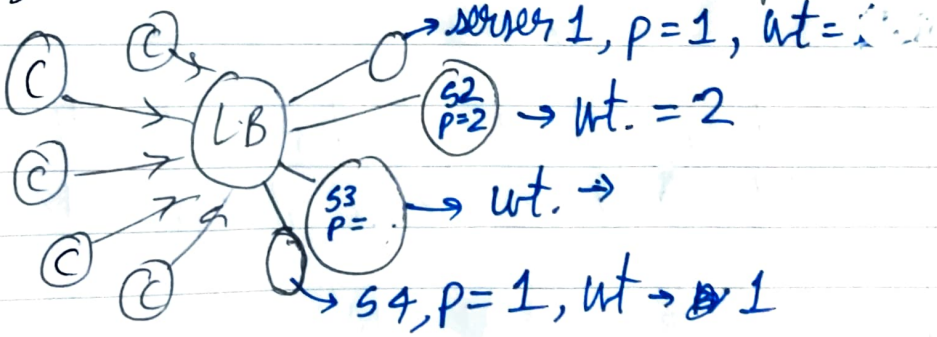
(ii) On addin / remov'n old server, it registers/ deregisters itself w.r.t. LB

2) Ways to select servers to direct load to

(1) Random redirec^n → No logic used. Just random selec^n of server from available servers
↓
by chance, one server may get overloaded

(2) Round Robin → L.B. goes thru all servers in order and allots a batch of reqs. to a server and so on until all servers hv equal load so for new batch of reqs., it'll again start frm 1st server
↓
server's power not optimally used.

Let size of server ∝ its power (p)



(3) Weighted Round Robin : L.B. goes as per round robin approach but if a weight is added on 2nd server, L.B. will send it a couple more reqs.
∴ Power of each server ⇒ optimized

Request Batch	Round Robin (RR)	Wtd. R-R.
1	S1	S1
2	S2	S2 } avg wt = 2
3	S3	S2 }
4	S4	S3
5	S1	S3 } wt = 3
6	S2	S3 }
7	S3	S4

(3) Performance/load Based : L.B. will do performance/health checks on servers

- how much traffic a server is handlin at any given time
- how long a server is takin to respond to traffic
- how many resources is the server usin
- etc.

Based on these checks, it allotes reqs (more reqs. low high performance servers

(4) IP based server selen : On gettin requests from clients L.B. hashes the IP address of the client, and dependin on this hashed value, the client's req. gets directed to a specific server

Use :- Caching

eg :- You're cachin results of certain reqs

in your servers, it's helpful to redirect the client to a server in which the response to ... request has already been cached

eg:- i) Client 1 searches for "cheese pizza"
~~ii) ... knows that the response to this request is ... in server 1 ...~~
ii) L.B. hashes down IPA of client to say "1" → basically a UID
iii) Now, it sees that in server S3 a result for a prev req. made with UID = 1 is cached and ready
iv) L.B. redirects client 1's req. to server S3
v) Client 1 gets his/her response super fast.

(5) Path-based → L.B. selects server or set of servers based on path of the reqs.

eg:- Algo Expert has servers S1, S2, S3 .... S25

All reqs related to payments → S1, S2 .... S9

All reqs related to running code → S10 - S18

All reqs related to watching vids → S19 - S21, S22, S24, S25

All reqs related to "about" page → S23

Date:
M T W T F S S

Use:

1) Deployment of big change to any service (eg:- running code service) affects only the servers allotted to this service.

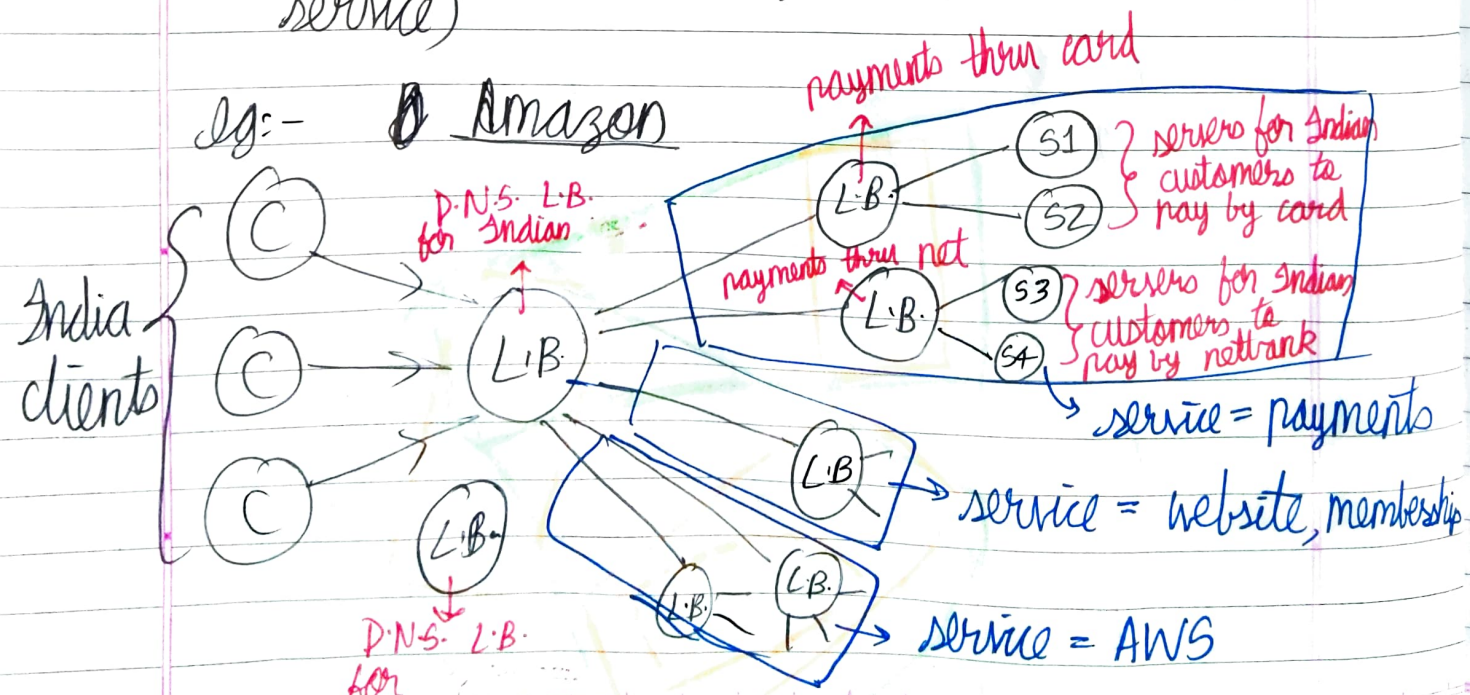
2) If one set of servers for a service start failing atleast other services are still up and running. (unaffected)

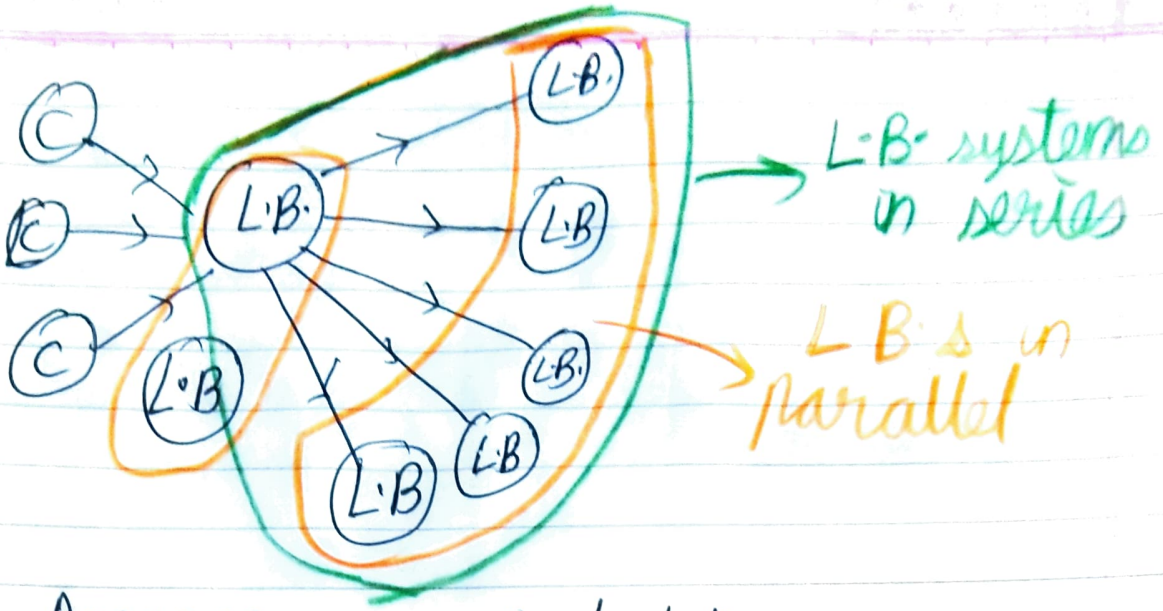
MULTIPLE LOAD BALANCERS

1) In series → to keep dividing a batch of reqs into smaller batches based on some criteria (eg:- path) on the req.

2) In parallel → To (1) just prevent 1 L.B. itself from getting overloaded or (2) Redirect to diff. servers based on req. criteria or server criteria (eg:- performance, diff kinda service)

eg:- Amazon





Amazon eg:- Just L·B·s

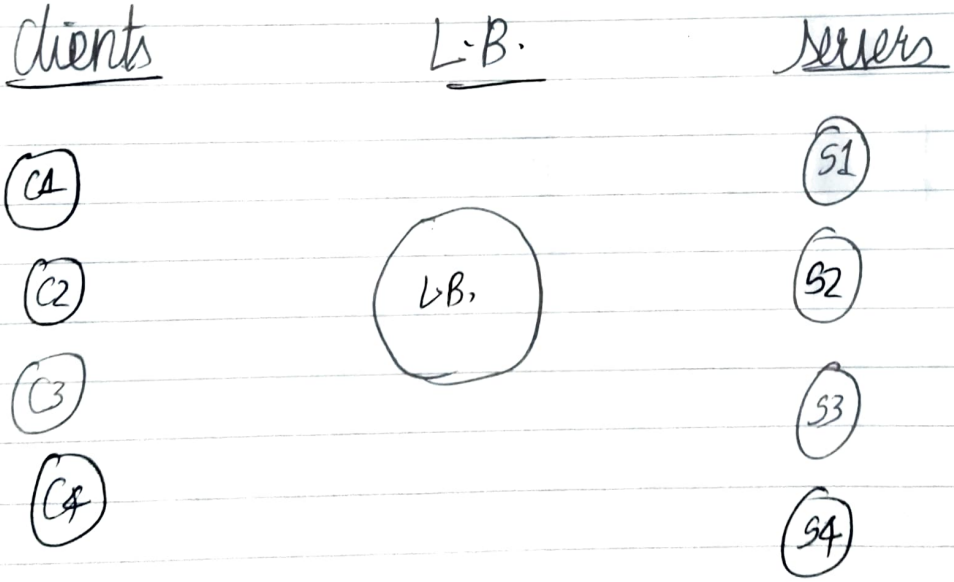
code on opp. page → shows wted R·R· selec^{n} meth.

Hashing

Hashing: An algo ie. performed to transform an arbitrary piece of data (str., arr, or other data type or struc) into a fixed size value, typically an int.

eg in sys. des:- Arbitrary Data → IP address, HTTP request, username,

1 Need for Hashing



(1) While server selec" strategies such as round robin or random get your work done in normal scenarios, they fail to optimize cache hits in a in-memory cache scenario

(2) In Memory Cache

a) Cache hits → no of times a req. recieves a cached res. from our sys. (faster)

(SSS)

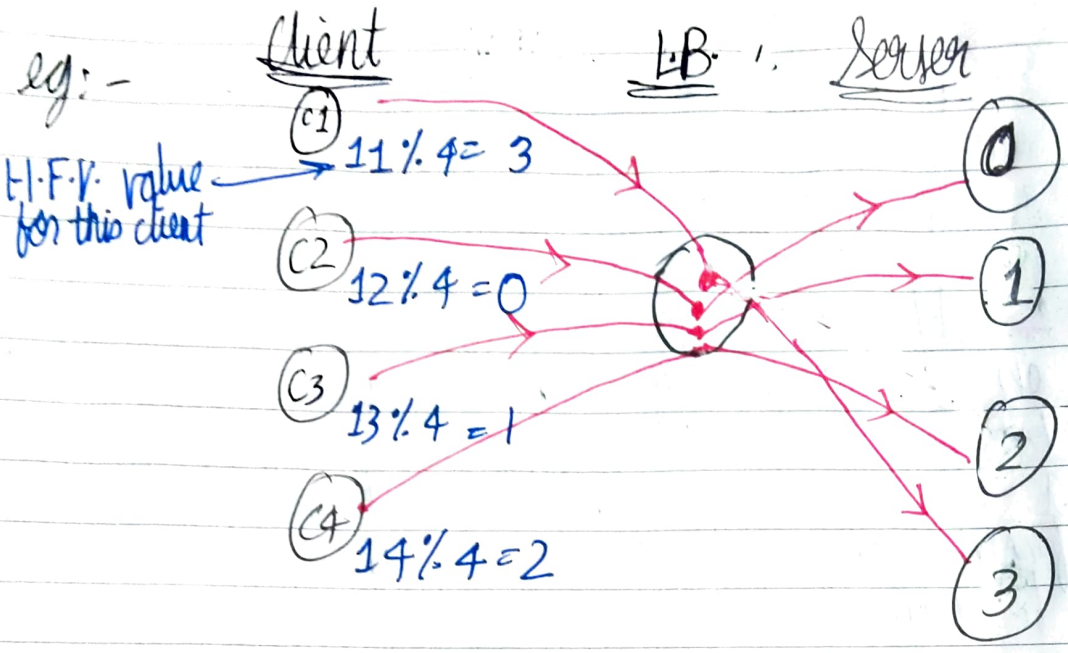
6) For this round robin or random server selection strategy don't work as they do not guarantee you that if the 1st time a client makes a req. and it gets processed and cached in S1, the 2nd time, when client makes the same req., he/she will be redirected to S1 only
∴ Chances of cache hits ↓

2) ∴ If you sys. des is s.t., it relies heavily on in-memory cache in its servers, but your L.B. is rerouting req.s from client, following a SSS that doesn't guarantee that: reqs from same client OR same reqs., will be routed to the same server every time, then, your in-memory caching sys. falls apart

3) ∴ Hashing will basically take these reqs → hash them to a fixed value → based on this value, req is allotted a server. → the 1st time req is processed, cached in that server → any subseq. times, the same req. is again sent to L.B. → now for the same req. we get the same hashed fixed rate (H.F.V.) → same hashed fixed value → so same server → same server so same resp. i.e. the cached resp.

2] Simple Hashing → server allotted to client (SATC) = (H.F.V.) % (no. of servers)

eg:-



(1) Uniformity in Hashin f^n (H.F.) → No. of collisions i.e. same server allotted to a no. of clients should be minimized.
i.e. clients should be distri.ed across servers equally, uniformly

NOTE: Typically you never write your own H.F, You use pre-made industry-grade H.F.s such as MD-5 or SHA-1 or Bcrypt

(2) Cache hits ↑ for same reqs.

(3) Problems:- Addin or removin (or dyin) of servers completely fucks up cache routes (C.R. → a route which a req. must follow to get its matchin cache a.k.a. route for cache hits)

mem → memory

eg:- Addn a server to curr. eg. after in-mem. caches have been created for reqs ~~which already~~ in their resp. SATC
Say server S5 is added now

Client	H.F.V.	$n_i = 4$	$n_f = 5$	$SATC_i$	$SATC_f$
C1	11	4	5	3	1
C2	12	4	5	0	2
C3	13	4	5	1	3
C4	14	4	5	2	4

n_i = initial no. of servers
$SATC_i$ = initial server (the one with in mem. cache) alotted to a req. from this client

→ As visible, no new reqs (i.e. reqs after addi^n of S5) are goin on their cache route
→ ∴ They won't reach server with cached res.
→ ∴ Cache hits ↓

3] Consistent Hashing

SAP → Server Allotment Procedure

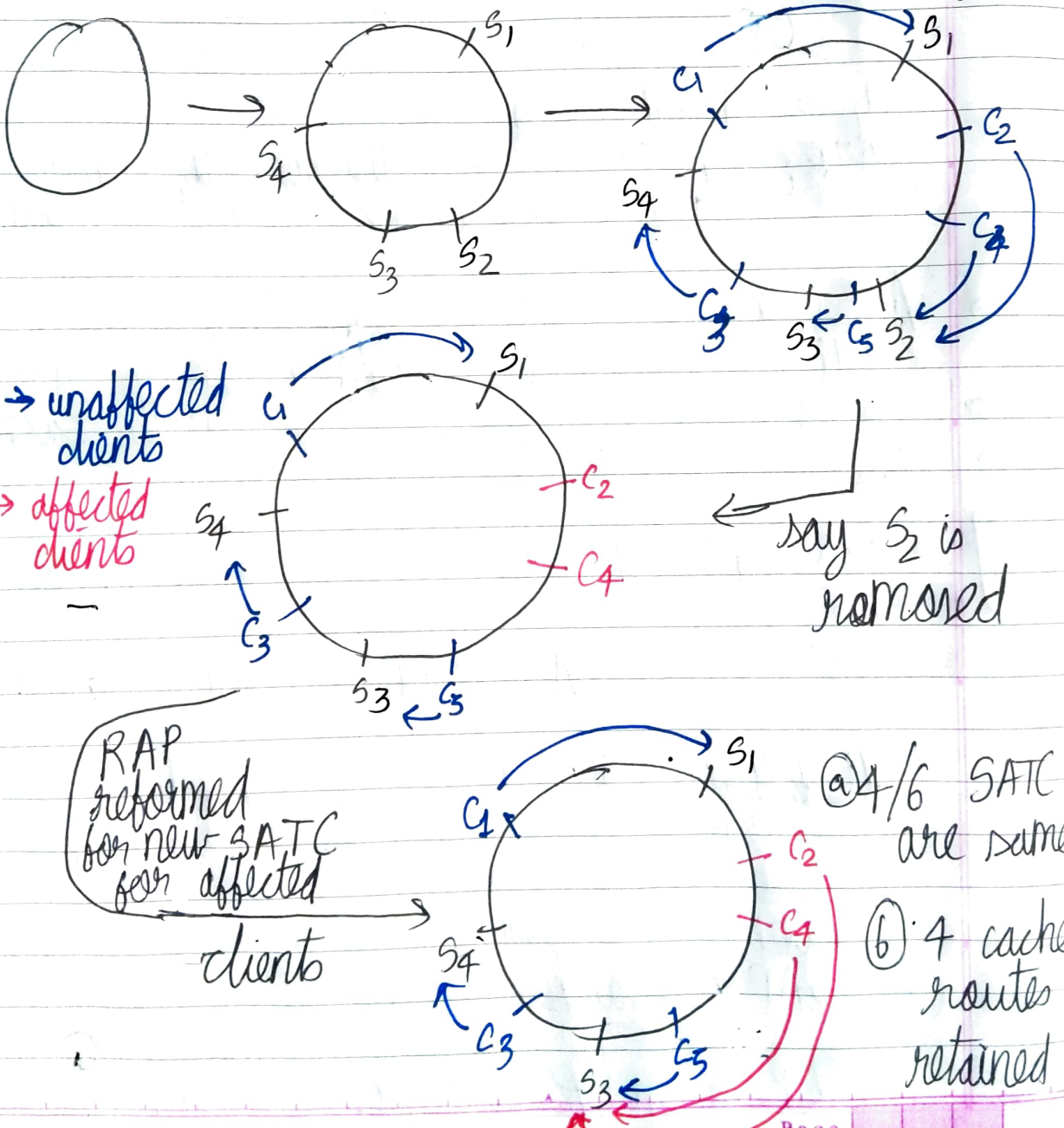
Visualiss^n : ① Servers are placed on a circle (preferably at equal dist. coverin entire circle in equal parts.
② Clients are now placed on the same circle
③ SAP → Matchin client to its closest server if we traverse circle in ? dir^n
SAP is performed to get SATC for each client

Date:
M T W T F S S

④ Why circle?:- Cuz it's a shape that is closed

∴ It is easy to just say that on addin or deleting a server, we know the next closest server just for the clients affected and so when we re-perform SAP to get SATC for each client we retain cache routes for all unaffected clients

eg:-

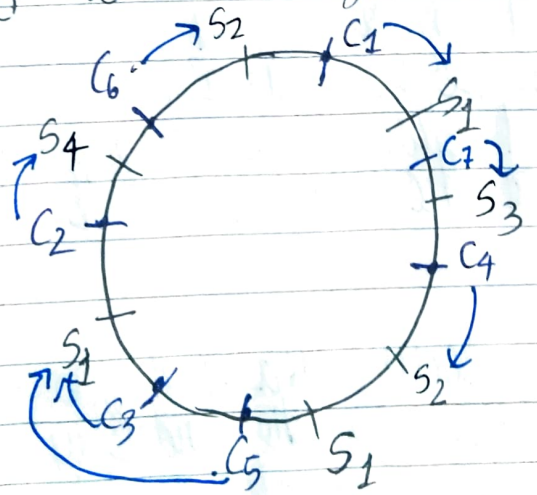


Thus, we retain atleast some cache routes irresp. of addi^n / removal of ^ server.

NOTE: ① Also, as visib C_1 is very far frm S_1 ↷ high latency
This prob. can be solved by arrangin servers at equal dist.s, coverin entire area of circle → real world set up servers in geographical loca^{n}s s.t. you minimize dist.s b/w all^ servers and clients mappin to em

② We can traverse circle in ↻ or ↺ dir^n. Dir^n doesn't really matter as long as it's fixed.

③ Say if you have a more powerful server i.e. server scaled vertically, just place it on more pts. in circle to ensure more clients are directed to it → real world place powerful server/ scale a server vertically, in geographical loca^{n}s where no. of clients is high.



Power → $S_1 > S_2 > S_3 = S_4 = S_5$

S_1 serves 3 clients
S_2 serves 2 clients
S_3, S_4 serve 1 client

Date
M T W T F S S

Rendezvous Hashing:

i) For every client, H.F. calculates and allots score for each server (i.e. ranks the servers)

(H.R.S.)

ii) The highest ranking server is chosen as the destina" server for our client. A uniform H.F. will ensure that for each client, we'll a diff. H.R.S. i.e., a diff. destina" server

iii) Thus, each server acts as an H.R.S for same client

iv) Removing server → If a server is removed then we go to the client associated with it and we then allot the 2nd highest ranking server for that client as the new destina" server for that client.

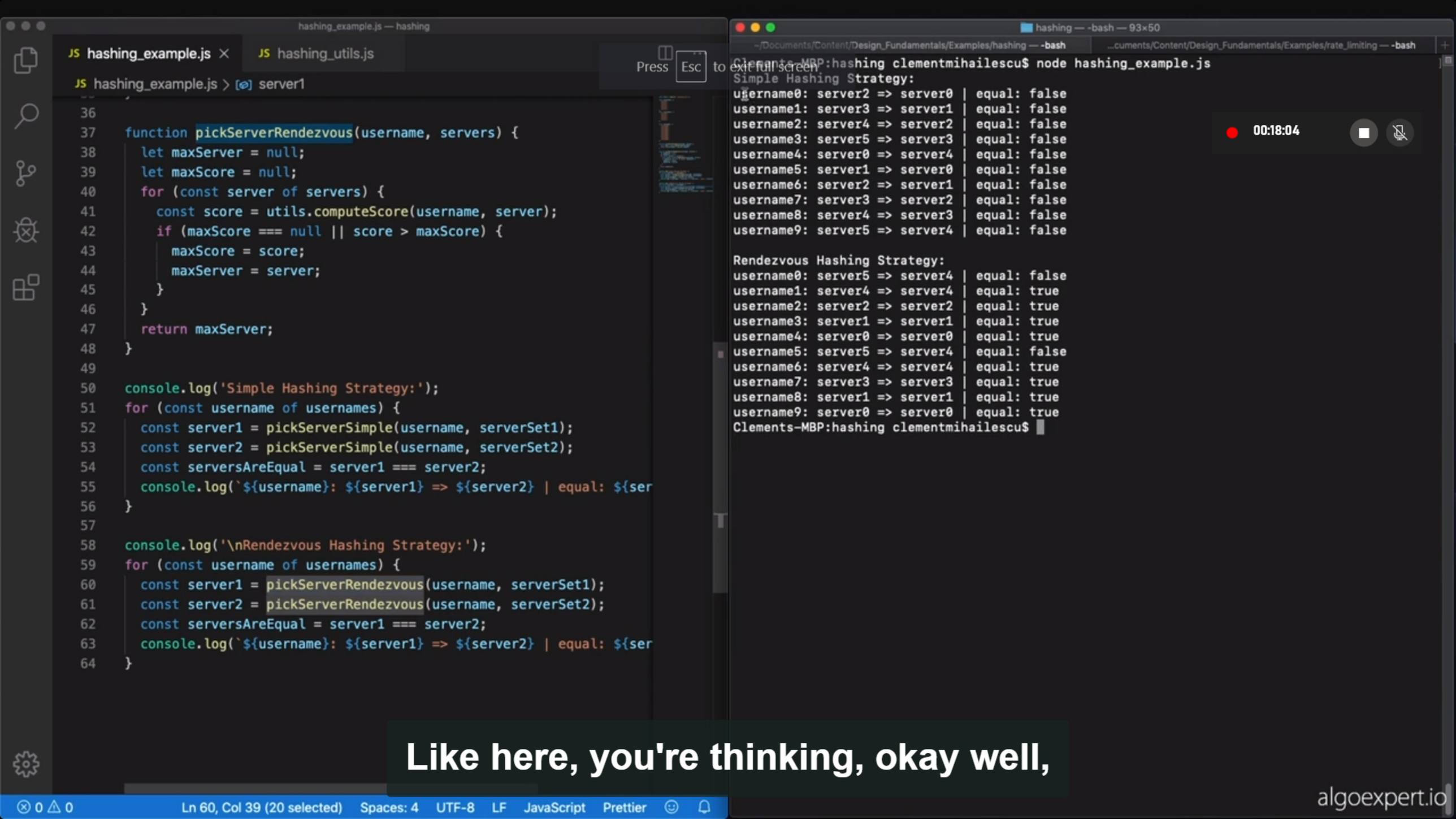
v) Adding a server → Re-ranking carried out for clients but new server will be pushed as H.R.S. only in 1 ranking so 1 client's destn. server will change but for other clients' destn. servers won't change so cache routes won't change so reqs. will still keep getting to the server with the in-mem. cache.

Note: SHA → Secure Hash Algos → collec" of cryptographic H.F.s in the industry. These days, SHA-3 is a popular choice to use in a sys.

Page

```
const serverSet1 = [ 'server1', 'server2', 'server3', 'server4', 'server5',]
const serverSet2 = [ 'server1', 'server2', 'server3', 'server4',]

const username = [ 'username0',  'username1', 'username2',  'username3',.....  'username9']

//SIMPLE HASHING
function hashString(string) {
  let hash = 0;
  if (string.length === 0) return hash;
  for (let i = 0; i < string.length; i++) {
    charCode = string.charCodeAt(i);
    hash = (hash << 5) - hash + charCode
    hash |= 0   }
  return hash}

//RENDEZVOUS HASHING
function computeScore(username, server) {
  const userNameHash = hashStrinng(username);
  const serverHash = hashString(server);
  return (userNameHash * 13 + serverHash * 11) % 67}

//SIMPLE HASHING
function pickServerSimple(username, servers) {
  const hash = utils.hashString(username);
  return servers[hash % servers.length]}

//RENDEZVOUS HASHING
function pickServerRendezvous(username, servers) {
  let maxServer = null;
  let maxScore = null;
```

```
    for (const server of servers) {
      const score = utils.computeSCore(username, server);
      if (maxScore === null || score > maxScore) {
        maxScore = score;
        maxServer = server
      }
    }
    return maxServer
  }


  //SIMPLE HASHING
  console.log('Simple Hashing Strategy')
  for (const username of usernames) {
    const server1 = pickServerSimple(username, serverSet1);
    const server2 = pickServerSimple(username, serverSet2);
    const serversAreEqual = server1 == server2
  }

  //RENDEZVOUS HASHING
  console.log('Rendezvous Hashing Strategy')
  for (const username of usernames) {
    const server1 = pickServerRendezvous(username, serverSet1);
    const server2 = pickServerRendezvous(username, serverSet2);
    const serversAreEqual = server1 == server2
  }
```


hashing_example.js — hashing
hashing_example.js
hashing_utils.js
Press Esc to exit full screen
hashing_example.js > server1
function pickServerRendezvous(username, servers) {
let maxServer = null;
let maxScore = null;
for (const server of servers) {
const score = utils.computeScore(username, server);
if (maxScore === null || score > maxScore) {
maxScore = score;
maxServer = server;
}
}
return maxServer;
}
console.log('Simple Hashing Strategy:');
for (const username of usernames) {
const server1 = pickServerSimple(username, serverSet1);
const server2 = pickServerSimple(username, serverSet2);
const serversAreEqual = server1 === server2;
console.log(`${username}: ${server1} => ${server2} | equal: ${ser
}
console.log('\nRendezvous Hashing Strategy:');
for (const username of usernames) {
const server1 = pickServerRendezvous(username, serverSet1);
const server2 = pickServerRendezvous(username, serverSet2);
const serversAreEqual = server1 === server2;
console.log(`${username}: ${server1} => ${server2} | equal: ${ser
}
hashing — -bash — 93×50
Clements-MBP:hashing clementmihailescu$ node hashing_example.js
Simple Hashing Strategy:
username0: server2 => server0 | equal: false
username1: server3 => server1 | equal: false
username2: server4 => server2 | equal: false
username3: server5 => server3 | equal: false
username4: server0 => server4 | equal: false
username5: server1 => server0 | equal: false
username6: server2 => server1 | equal: false
username7: server3 => server2 | equal: false
username8: server4 => server3 | equal: false
username9: server5 => server4 | equal: false
Rendezvous Hashing Strategy:
username0: server5 => server4 | equal: false
username1: server4 => server4 | equal: true
username2: server2 => server2 | equal: true
username3: server1 => server1 | equal: true
username4: server0 => server0 | equal: true
username5: server5 => server4 | equal: false
username6: server4 => server4 | equal: true
username7: server3 => server3 | equal: true
username8: server1 => server1 | equal: true
username9: server0 => server0 | equal: true
Clements-MBP:hashing clementmihailescu$
00:18:04
Like here, you're thinking, okay well,
algoexpert.io

Rela^{n}al Databases

1) lotta dBs out there

2) 2 major categories depending on struc imposed on these dBs

(1) Rela^{n}al a.k.a SQL dB (eg:- POSTGRESQL)

(i) A type of dB that imposes on the data stored in it, a tabular like structure (data stored as table)

(ii) Rows (record) → instances of the entities that the resp. tables represent

Columns (attribute) → Attributes of the entities

eg:- table → payments record for a shop

row → each customer payment details such as date, amount, transaction medium

colum → the attr.

Cust. name	Date	Amt	Transac^{n} medium
A	17/9	10	Credit Card
B	15/9	45	Cash
C	14/9	3	Netbanking
D	13/9	87	Cash

(iii) Schema → Define the shape of the table i.e. it will basically specify what attr (col) each entity (row) will have

eg:- schema for a book of library

Each row → book name

For each row, cols → borrowed on, returned on, details of payment, condition of book

Date:
M T W T F S S

② Non rela'al dB a.k.a noSQL (eg:- mongo DB, Google Cloud Datastore)

(i) dBs don't impose tabular struc. on data stored in them. Sometimes they may impose some kinda other struc. but def'ly not table

(ii) Flexible, less rigorous

SQL → Structured Query Lang.

i) Most rela'al dB (rdB) support SQL

ii) SQL is used to perform powerful complex queries in rdBs

iii) Because rdBs support SQL, they come with powerful querying capabilities and this is also the reason why most people choose rDBs over non-rDBs for part of their sys.

iv) Why SQL > Python, JS

Cuz for Python, JS you gotta load data in mem. to perform these kinda queries

So when we talk abt large scale distri. sys, we got Tbs of data and we can't do this trivially

SQL dBs aka RDBs

1] Must use ACID transac"s → transac" supportin ACID props.

Atomicity Consistency Isola" Durability

i) Atomicity - If a transac" consists of multiple sub-opn then these sub-opn are gonna be considered as 1 unit so they'll either all succeed or all fail

eg:- ₹ funds transfer in banks

→ sub-opn 1 → Reduce money in person 1's acc

→ sub-opn 2 → Increase money in person 2's acc

ii) Consistency (aka Strong Consistency) → dB is never invalid and never stale

- (i) Any transac" in dB is gonna abide by all the rules in the dB → dB is never invalid
- (ii) Any future transac" in dB, is gonna consider any past transac"s in dB i.e. no 'stale' state' in dB where 1 transac" has exec.ed but another transac" doesn't know it has exec.ed → dB is never stale

iii) Isola" - Multip. transac"s can occur at the same time but under the hood, they will actually hv been exec.ed as if they had been done sequentially (i.e. put in a queue) one-by-one

iv) Durability - When you make a transac" in a dB, the effects of that transac" in the dB are permanent

i.e. data stored in the dB is effectively stored in disc (not mem.)

Database Index (dBI)

A dBI is a data struc that improves the speed of data retrieval ops on a dB at the cost of addi"al writes and storage space to maintain the index data struc

Consider this eg for bank passbook

Month	Date	Amount credited	Amt debited
January	28th	10000	—
Feb	7th	5000	—
March	15th	700	—
April	27th	2000	—

Say we wanna figure out on which day of the yr was the highest amt. credited. (Ans:- Jan 28th)

(i) Conventional Meth :- Traverse thru entire look thru entire dB to find that amt

O(N)T

(ii) Using dBI :- Auxiliary data struc created i.e. optimized for fast searching on a specific attr. (col.) in the table.

Say our dBI = table that has all credited amts stored in sorted order (O(NlogN)) and

Key-Value Stores

Sometimes rDBs are too cumbersome ~~@~~ ~~t~~ : when weighed against the services they offer. Also, they impose a tabular struc. on data viz not reqd. everytime. In those cases, we use non-rDBs for more flexibility and ease

Key-Value store → i) A popular non-rDb (i.e. noSQL)
ii) Data stored as k-v pairs.
iii) Allows mapping of keys (~~strings~~ usually strings) to ~~values~~ arbitrary values.

eg1:

Key	Value
foo	-48.8
bleh	ApplePie
whut	[x, y, z]

eg 2: hashTables

k-v store →
- flexible → not rigorous as no imposed struc.
- simple → nothin simpler to store data compared to a k-v map.
- fast

Use:

1) Caching : Values → Responses to network reqs.
Keys → Hash, IP address, username

2) Dynamic config : Basically just storing a parameter that diff. parts of your sys. rely on in a separate place so all these parts can access it

eg :- using config" when you wanna switch what code you wanna run depending on what envirnt. you're in → (development, prod", testing)

Here, this param. is stored in a k-v store (usually an obj. in .json or .yaml file) as working with diff. values of this same param (key) is a lot easier

NOTE: Advantage of using a k-v store → ∵ values are accessed directly thru keys, no search thru DB or any other fancy lookup reqd.
∴ Latency ↓, Throughput ↑ of our sys.

eg:- DynamoDB, Redis, Etcd, Zookeeper

NOTE: 2 types of k-v stores based on where they write their data to:

i) Write data to disc → a) Slow to get back data from disc
b) Data persists even after k-v store server itself crashes

ii) Write data to mem → a) Faster to retrieve data from mem
b) Data lost if k-v store server crashes

Use :- Caching → As honestly, it doesn't affect the sys. much if caches crash but we do want reading from cache = fast

Date:
M T W T F S S

~~Also, in~~

∴ when a caching involving k-v store goes down, we pretty much don't lose anything except some cache hits viz not that imp. a loss.

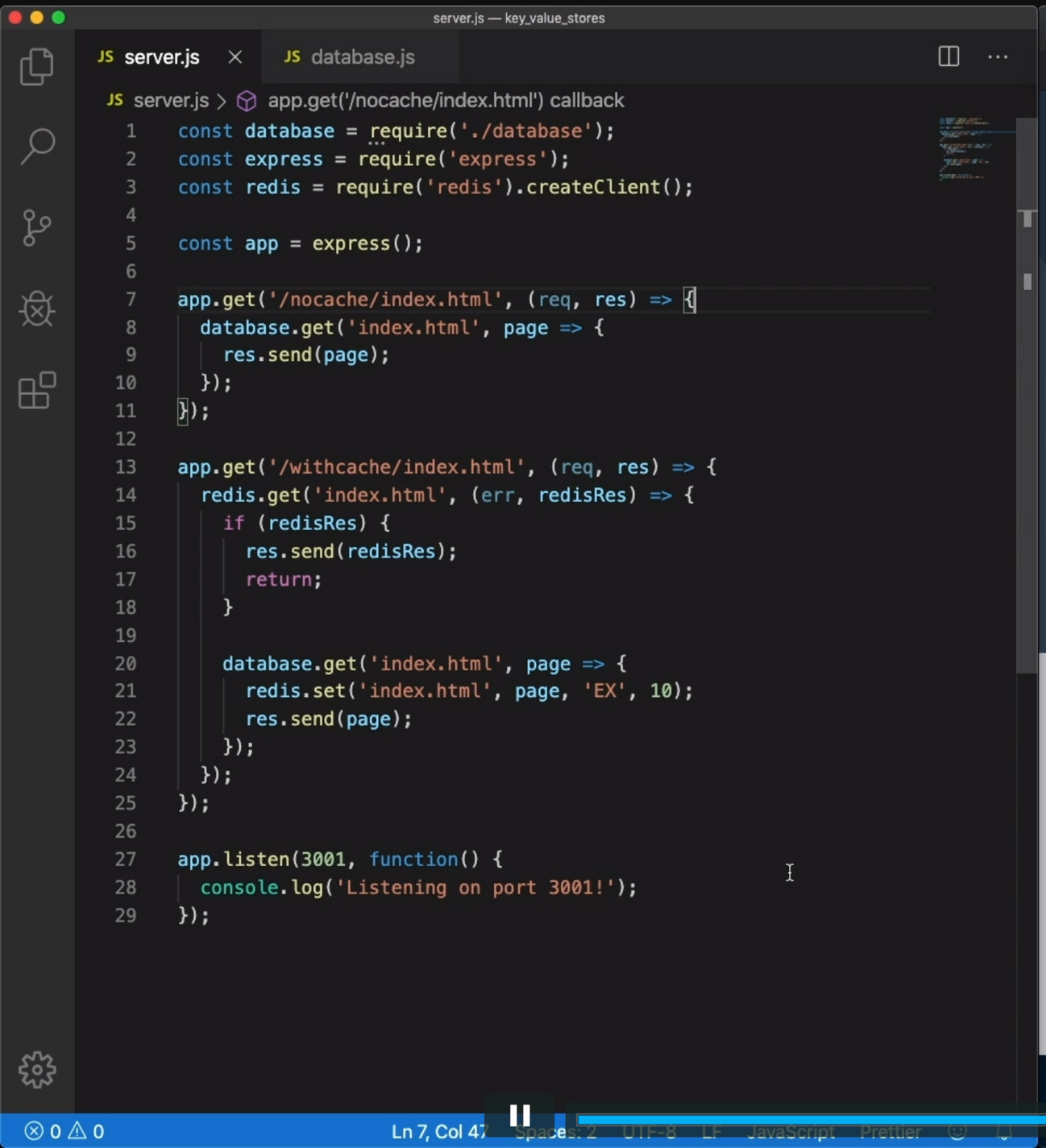
---

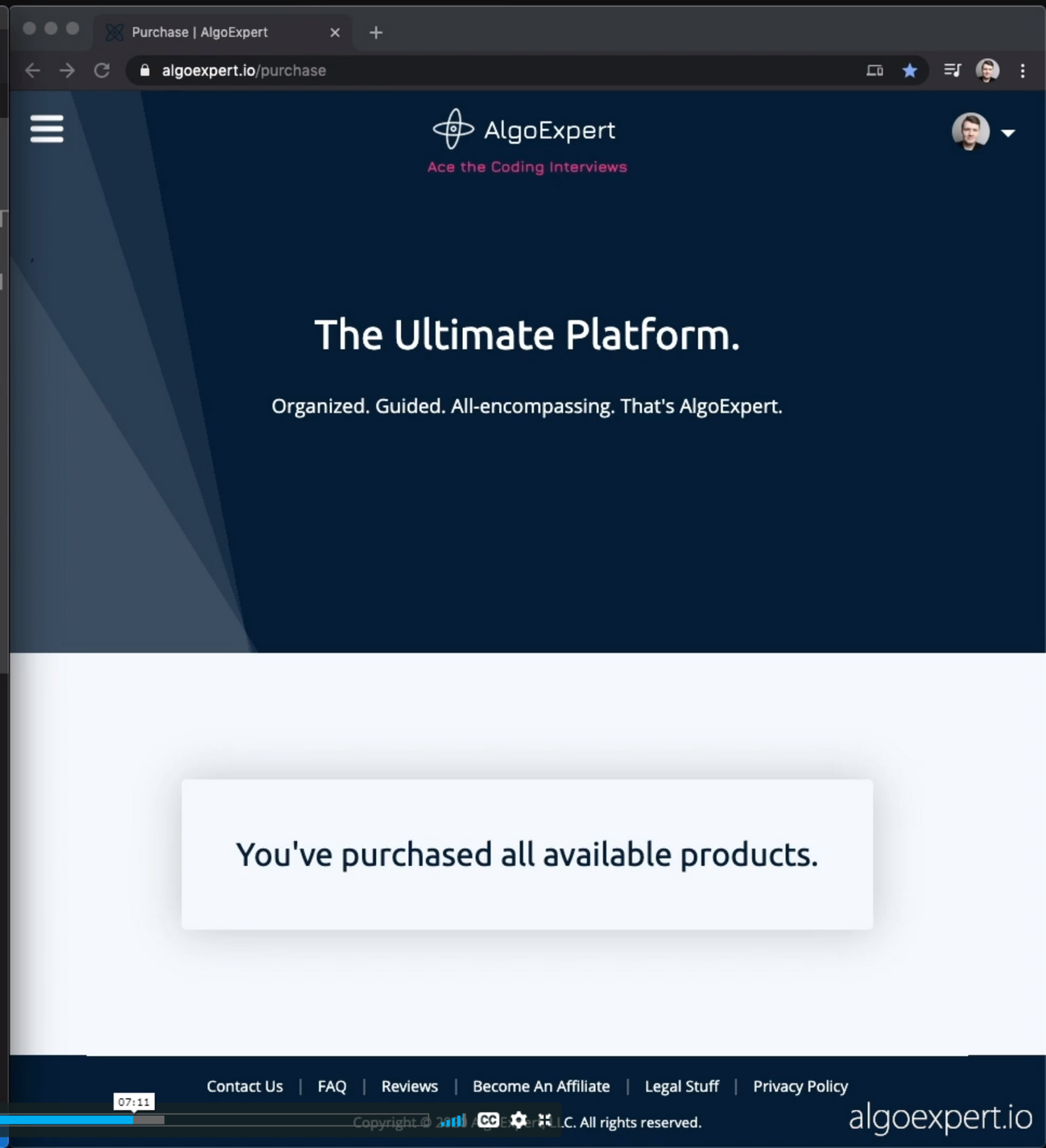
eg' of k-v stores:

1) Etcd
   - → strong consistency
   - → high availability
   - → Use: implement leader elec^n in a sys.
   - → writes data to disc

2) Redis
   - → writes data to mem. a.k.a in mem. k-v store
   - → V. lil persistent storage
   - → Uses: best-effort, fast caching implementa^n of 'rate limiting'

3) Zookeeper
   - → writes data to disc
   - → strong consistency
   - → high availability
   - → Use: Store imp config^n Implement leader elec^n


server.js — key_value_stores
server.js
database.js
server.js > app.get('/nocache/index.html') callback
const database = require('./database');
const express = require('express');
const redis = require('redis').createClient();
const app = express();
app.get('/nocache/index.html', (req, res) => {
database.get('index.html', page => {
res.send(page);
});
});
app.get('/withcache/index.html', (req, res) => {
redis.get('index.html', (err, redisRes) => {
if (redisRes) {
res.send(redisRes);
return;
}
database.get('index.html', page => {
redis.set('index.html', page, 'EX', 10);
res.send(page);
});
});
});
app.listen(3001, function() {
console.log('Listening on port 3001!');
});
Ln 7, Col 47
Spaces: 2
UTF-8
LF
JavaScript
Prettier


Purchase | AlgoExpert
algoexpert.io/purchase
AlgoExpert
Ace the Coding Interviews
The Ultimate Platform.
Organized. Guided. All-encompassing. That's AlgoExpert.
You've purchased all available products.
Contact Us | FAQ | Reviews | Become An Affiliate | Legal Stuff | Privacy Policy
07:11
Copyright © 2019 AlgoExpert LLC. All rights reserved.
algoexpert.io

Date:
M T W T F S S

Configura"

A] Config" → set of params / consts that multip parts of our sys / app use so we store em in a separate (seemingly isolated) file and access em everywhere

Use 1: Storin config files for params that affect certain code blocks in your app in a certain manner dependin on what envmt you're in → devmt., prod", testin

B] FORMAT:
Usually config files are stored in .json or .yaml but they can be stored in any format and depends on kinda work people's pref-, etc. but it pretty much doesn't differ in terms of overall f"ality of the file

C] 2 types of config":

1) Static config: Config" bundled with code so if you wanna change any specific param in config file, you gotta redeploy entire code (i.e. make it go thru entire testing, code review process etc.) cuz config" is packaged with code

Advantage: Each time you submit code to code-base, code undergoes thorough review process. ∴ if change might break the app, the breakage will likely be caught in personal tests, quality

assurance tests (testing that happens before app is deployed). Safer approach

Cons → slower to see changes as entire app's gotta be redeployed

2) Dynamic Config": Config" is completely separated from app code. ∴ Config" changes are instantaneously visib in app.
This is a bit complex as its backed by a dB that your app is queryin to see what the curr. config"

Use : eg :- build a UI on top of a config. ∴ any changes in UI → super easy to implement, visualize w/o redeployment of app code

Pros : Fast changes in app. Deployin new features real fast.

Cons : Risky cuz not a lotta thorough review, testing.

Implementa" : Dynamic config" is used with tools built around it to make it safe

eg :- i) building a review sys on top of a UI built on top of ~~config~~ dynamic config"
ii) access controls → only certain ppl in org can make config" changes
iii) build deployment sys around config"

Date:
M T W T F S S

eg:- app related to config" is deployed only after specific periods of time

period of time after which app is deployed automatically
eg:- twice / week

>>

period of time after which config" changes are deployed
eg: twice / day

a build deployment sys. around config"

ii) Only few users (say 1%) actually see the config" change for the 1st few hrs/days after app deployment

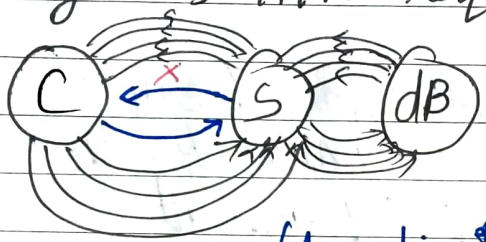
RATE LIMITING

Rate limiting:

1. Settin a threshold of sorts on certain opns. past which these opns return errs. Limitin the amt. of opns. that can be performed in given time

eg:- Only 5 HTTP req. from client every 10 secs.

In IDs



— : for opn 1, 2, 3, 4
— : for 5th opn

→ (1 machine * many reqs) = many reqs

2. Denial of Service (DoS) attack ⇒ Malicious client floods the sys with too many req and effectively clogs the sys with way more traffic that it can handle and thus, sys is brought down

3. Rate limitin prevents cloggin of servers by DoS attacks as past a certain no. of reqs, it'll throw errs. and stop takin in reqs.

eg:- code exec^n engine on AlgoExp. Run code op^n is rate ltd. to prev. someone from spammin the op^n i.e. cloggin the dB with reqs.

4. Rate limiting can also be dependent on other factors:

i) Based on user: Identify user issuin req.

Date:
M T W T F S S

by lookin at req. header or authentica^{n} credential and ~~we~~ once identified we can rate limit no of opns. by user eg:- 3 opns/min for a student to access his/her grades on insti. website

ii) Based on IP address
iii) Based on region of the world
iv) Based on whole sys. eg:- a sys as a whole should never allow servers to handle $> 10^4$ reqs per min.
v) Based on a certain org. fallin under same IP address.

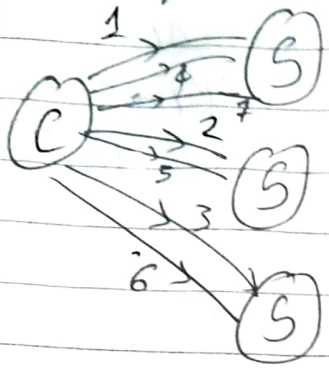
(5) DDoS attack → Distributed DoS attack

(many machines so diff IP address) * (some reqs by each machine) = (many reqs to servers)

might not be identifiable as part of same org/grp.

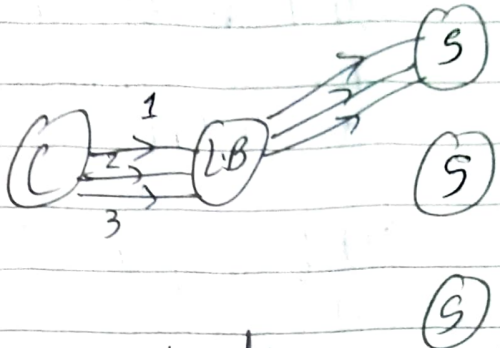
(6) In large-scale distri sys → we gotta hv. v. powerful rate limiting ~~to prot~~ → i.e. we need to hv really powerful load balancin to ensure that a client's req are always routed to the same server (amongst a set of servers cuz its a big sys) to ensure that we have some prev access records of this clients reqs. to the sys, stored in-memory in the same server place so we can compare and say whether ant. of reqs. this client is issuin is within a safe boundary or not

W/o L.B. in



Client can flood all servers as rate lim. is met/server but not across servers (sys as a whole) so sys is flooded with traffic

With L.B

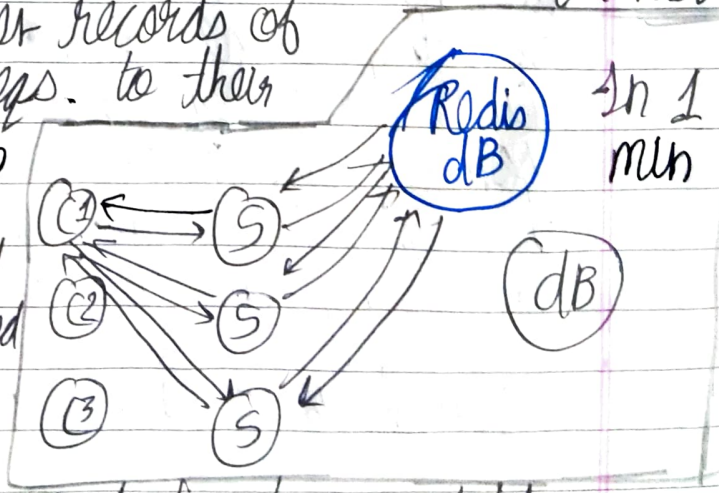


Client always routed to same server → it has prev access records of this client so it'll rate lim. client if it's floodin with too many reqs

∴ It is imp. to config load balancing keepin rate lim. in in mind

(7) ∴ In most cases, large scale distri. sys's use handle rate lim. in in a separated service or dB instead of in-mem. in servers all your servers speak to this separate dB and realise if they gotta rate lim. client or not

which stores prev records of all clients' reqs. to their resp. servers

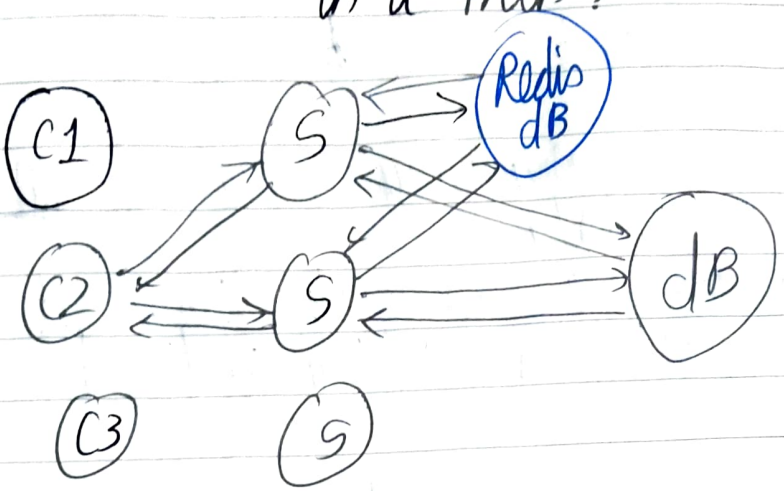


eg! - Redis. → eg 3 req/min allowed

↓ separate dB that stores prev client access records and does rate lim logic and likewise, tells a server to rate lim a client or not

on its 3rd req.

In our eg:- C1 is blocked / rate ltd from all our servers as it issued >2 reqs. in a min.



C2 has ≤ 2 reqs per min so it get backs data both times from main dB

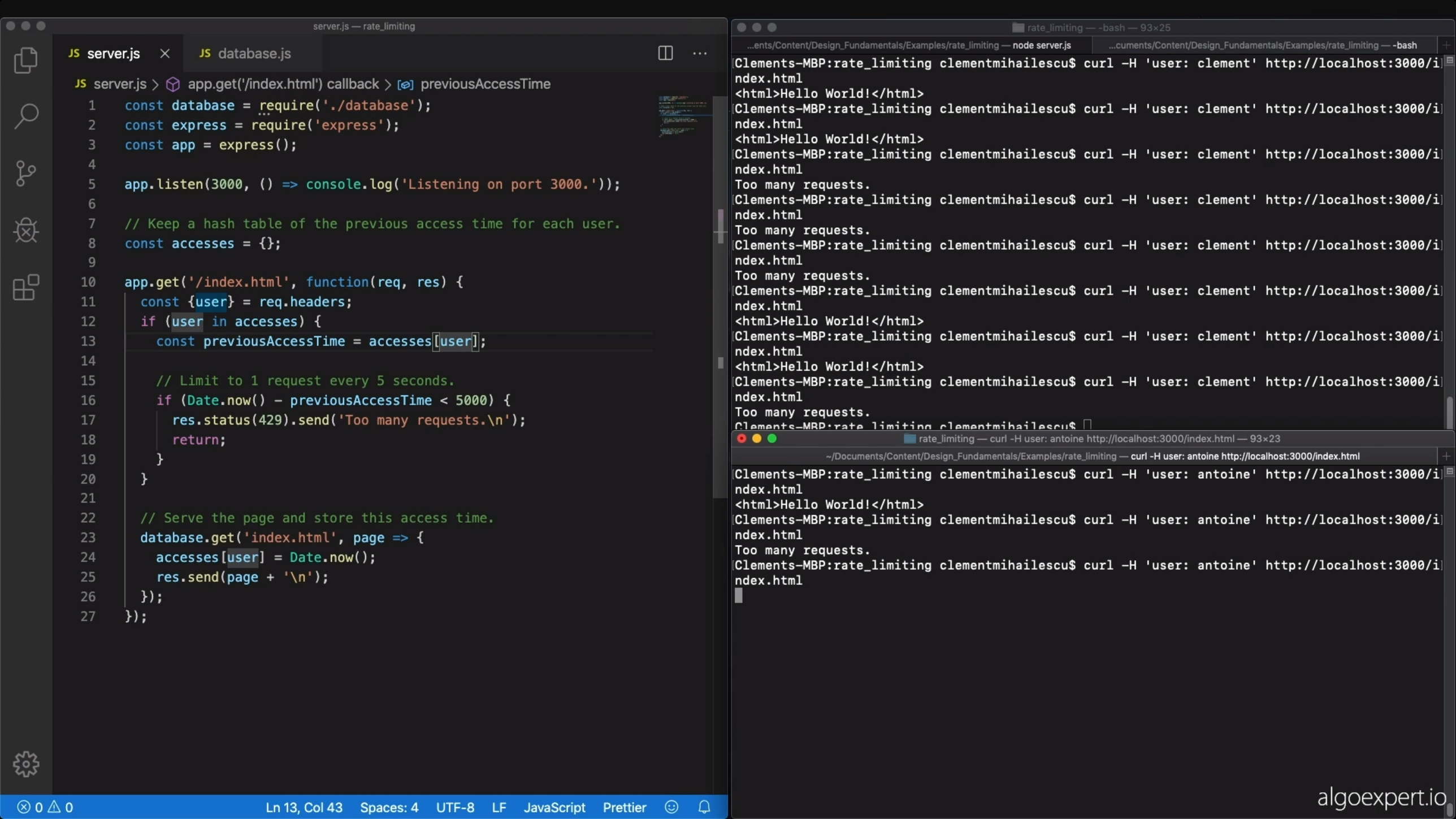
(8) Tier based Rate Limiting

eg - HITP reqs:

∴ req./0.5s but only 3 reqs/10sec but only 10 reqs/min

Thus, we ~~dont~~ give users bit more freedom whilst still maintaining a strong grasp over their power to keep req. in

Complex to code this out tho



SPECIALIZED STORAGE PARADIGMS

1] BLOB STORAGE (BLOB = Binary large Object)

i) Blob → arbitrary piece of unstructured data
eg - video file, img file, large txt file, large binary (compiled code) file

ii) Blob store → Storage soln for blobs.

iii) Info abt Blob stores

(1) They don't count as a dB since they only only allow storage and retrieval of data based on the name of the blob.

(2) They're usually used for storin large size blobs for small and large scale sys.s

(3) Blob-store accesses blobs usually thru a key or blob name
It specializes in storin massive amts. of unstructured data i.e. blobs.

(4)

Blob	k-v store
Blob accessed thru key	Value accessed thru key
Optimized to store massive amts of unstructured data → high availability and durability of data	Optimized for latency

Date:
M T W T F S S

(5) Really complex prob → accessin blobs from blobstore, so generally relyin on popular blobstore soln. (eg:- Google Cloud Storage, Amazon S3) is suggested as these are big companies that have that kinda infra. to support blobstore soln.
Money is charged based on
- how much storage is used
- how often are blobs stored, retrieved

2] Time Series Database (TSDB)

i) TSDM - special kinda dB optimized for storin and analyzin time-indexed data

ii) Time Indexed Data - Data pts. that specifically occur at a given moment in time
Data ≡ events that happen at a given time say every millisecond

To perform time-series like computa"s on this data, eg
↓
TSDB is used
- computin avg
- computin local maxima, minima
- aggregatin all data b/w 2 specific pts in time

iii) Use cases:

① Monitorin: Lookin, analyzin, computin a bunch of events occurin in our sys. at a given timestamp.

② IoT : Lots of devices which are constantly sending or capturing telemetry or some other data in their environments.

③ Stock Prices, Cryptocurrency prices → changing all the time

(iv) Eg :- InfluxDB, Prometheus

3] Graph dB

(i) Used when within the data that you store, there are a lotta relationships b/w the data (i.e. indiv data pts. in dataset) / multiple lvls of relationship b/w data

eg :- Facebook accounts

dataset
- data pt 1 → acc 1 : Harry, age : 16, eye-color : green
- data pt 2 → acc 2 : Ron, age : 17, eye-color : brown

} RELATIONSHIP : friends

(ii) In SQL table format, querying certain pieces of data that rely on a lotta diff relationships b/w data stored in tables, becomes really complicated

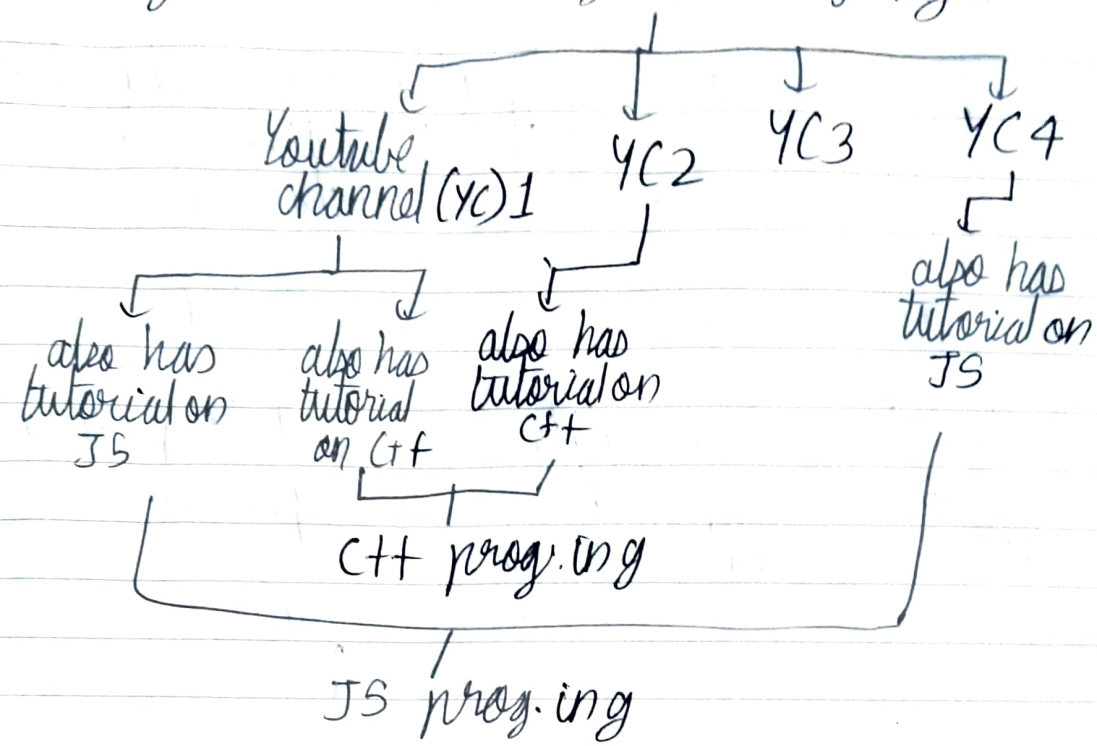
(iii) Graph dB → ① dB built on top of the graph data model.
② ∴ the concept of a relationship b/w indiv data pts is high priority
③ Complex queries on deeply connected data done very fast

(iv) Use cases :

① Social Networks

(2) Websites with relq^n ships b/w people and pages or content

eg:- Content → Python Prog. ing



(v) Most popular graph dB - Neo4j

(vi) Cypher (.cql) - A 'graph query lang' originally developed for the Neo4j graph dB but is now used in a lotta other dBs (aka "SQL for graphs")

Cypher queries are lot simpler than SQL queries when dealin with deeply connected data

eg:- to find cypher query in Neo4j

MATCH (some_node : Some Label) -[: SOME_RELATIONSHIP]-> (some_other_node : Some Label { some prop : val })

4] Spatial dB

i) Optimized for storing spatial data

ii) Spatial Data : literally any data that deals with geometric space.

eg:- loca"s on a map, restaurants in a locality

iii)

Database idx	Spatial idx
i) Used to perform complex queries on 1 col (i.e 1 attr say latitude) really fast	Used to perform complex queries on 2 cols (i.e 2 attrs → latitude, longitude) Also, these queries are referred as spatial related queries

eg.s of spatial related queries :→ finding all loca" in vicinity of a specific loca"

→ dist b/w 2 loca"s

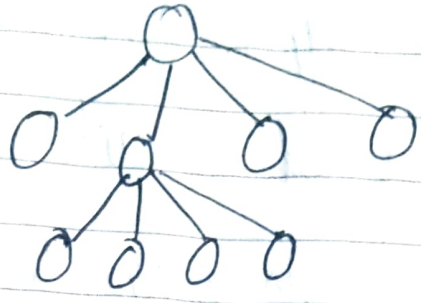
eg.s of a spatial idx :→ Quad tree
→ R tree
→ K-D tree
→ M tree

iv) Quad tree

① A tree data struc used to index 2-D spatial data

② Rule : Each tree node has no. of child nodes → == 0 (leaf node)
→ == 4

eg: of quad tree



#4 or nothin

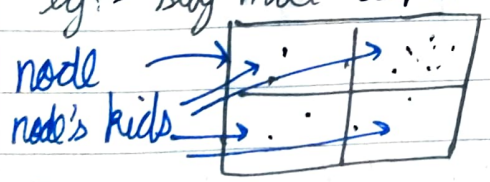
(3) Quad tree nodes → contains some form of spatial data that has a max capa.

(a) for node value
- < max capa. node is undivided i.e. leaf node condi"
- > max capa. node is given 4 kids nodes and its data entries is split across the 4 kid nodes

(4) Graphical visualisa" of quad tree

(a) quad tree node = rect.
(b) quad tree node value = dot inside rect.
(c) max capa. of quad tree node value = max no. of dots a rect. can accomodate before we split it into 4 kids each hjin some of these dots.

eg:- say max capa. = 10



(d) q entire quad tree = grid filled with rect.s that are recursively subdivided into sub-rect.s

Date:
M T W T F S S

eg on opp page: storing loca^{n}s in the world,
let max-node-capa. = n

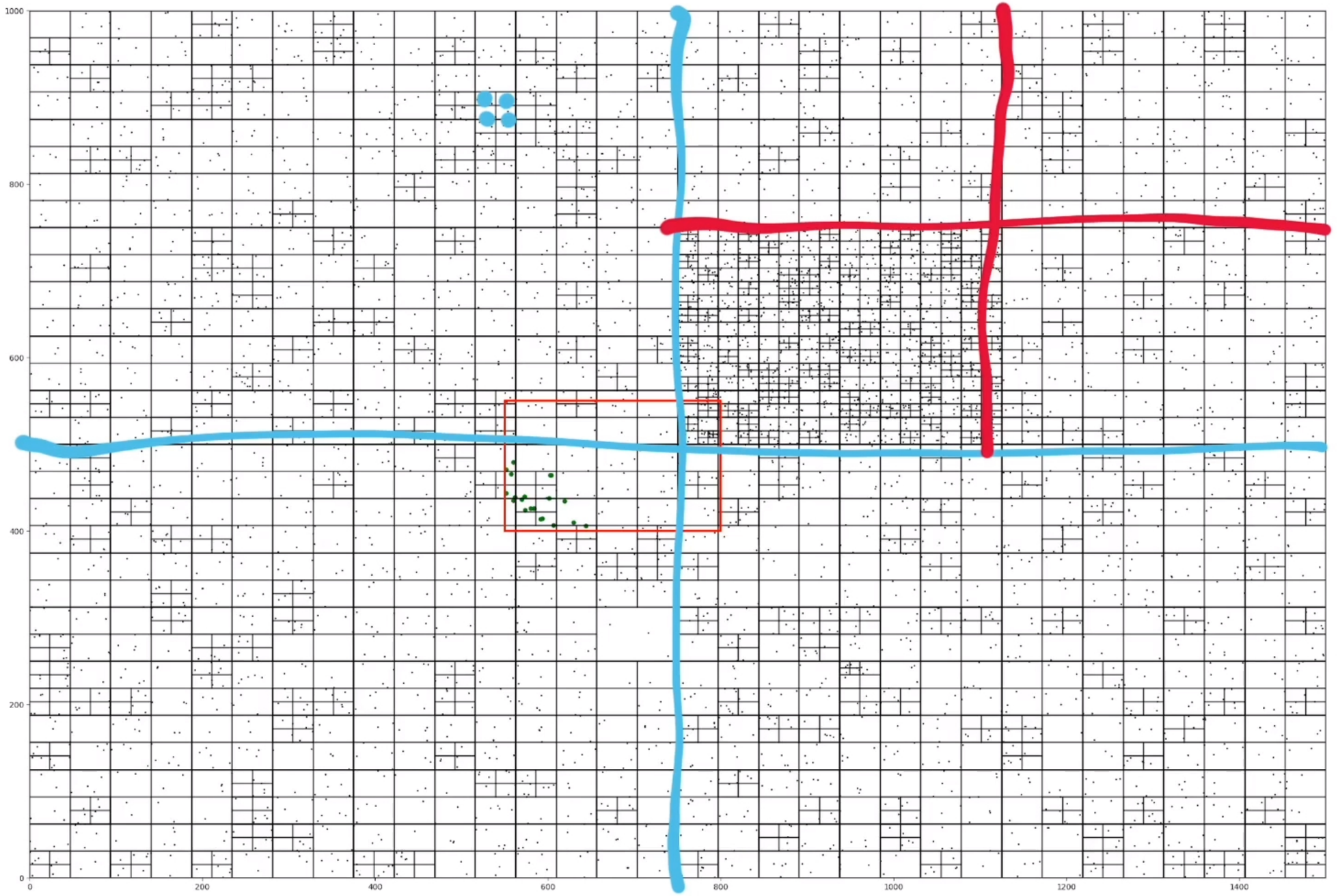
i) Root Node ≡ world ≡ outermost rect.

ii) If entire world has more than n-loca^{n}s, outermost rect divided into 4 quadrants (each representing a ¼ region in the world)

iii) Regions that have
- > n loca^{n}s → further subdivided into 4 small sub-regions (sub-rects.) i.e. corres. quad tree node is given 4 kids nodes
- < n loca^{n}s → undivided rects ≡ leaf nodes in quad tree

iv) Parts of grid with
- many subdivided rects → represent densely populated areas eg: cities
- few subdivided rects → represent sparsely populated areas. eg: villages

v) Finding a given loca^{n} in quad tree: $O(\log_4 x)$T
where x = total no. of loca^{n}s
Analogy: kinda like Quadrary search ≈ version of Binary Search


1000
800
600
400
200
0
0
200
400
600
800
1000
1200
1400

Replicaⁿ and Sharding

1) A dB is critical to sys.
2) ∴ Sys.'s performance ∝ dB's performance.
3)

	dB	Sys
eg 1)	unavail.	unavail.
eg 2)	low thruput	low thruput

4) ∴ While des-in a sys and makin it performin, we gotta ensure its dB is performant, else sys will fall apart

I] Replicaⁿ

The act of duplicatin data frm 1 dB server to others

Use case (i) : To act as backup if main dB server fails

(1) When main dB goes down, clients can't perform any opns. (read, write, etc.) on the data

(2) To prev. clients frm experiencin this we establish a secondary dB (backup) i.e. a replica of the main dB.

(3) For this to happen the main dB, on receivin req.
- → performs whatev opn (read, write, etc) the req. is for
- → updates the replica s.t. replica === main dB i.e. in 'sync' with dB at any given pt. in time

④ We do this to ensure that replica DB can take over in case main DB fails

⑤ main DB fails → replica takes over as new main DB → after some time → main DB comes back up → now, main DB is updated by replica ← now, they swap roles, i.e. main DB take over again as main DB → replica goes back to acting as a standby.

⑥ Write opns to main DB take a lil longer as you gotta write data in ~~main~~ DB and replica to ensure replica is always 'in sync' (i.e. up-to-date) w.r.t. main DB
"Each write opn of main DB should sync. by be done in replica. If write opn. fails on replica due to some reason (eg:- network partn"), then the write opn. should also not be completed on main DB"

⑦ Replica should NEVER have stale data.

⑧ Redundancy of sys ↑

Use Case (ii) : To move data closer to clients, thereby ↓ latency of accessing ~~data~~ specific data.

① Consen" : When you save data to a DB, it gets

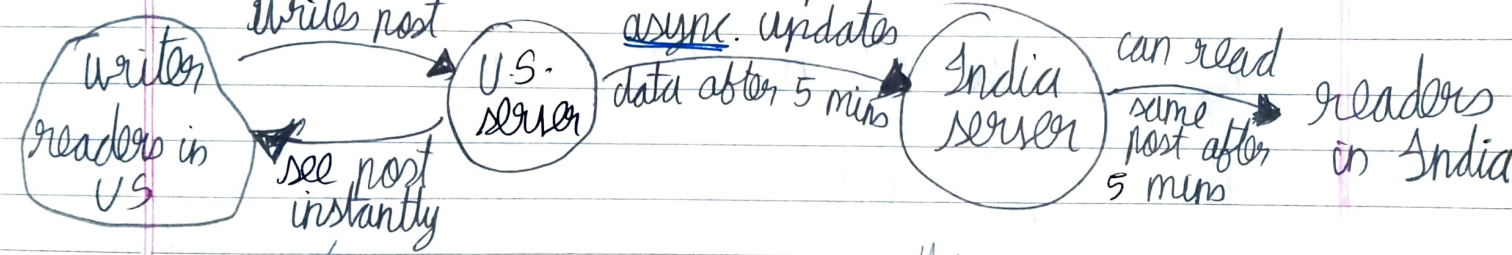
stored but time taken & dist. of server from user.

(2) To bring server closer to user, i.e. ↓ latency, we use a replica server close to user and user stores his/her writes to it. (takes lesser time as server is closer)

(3) We async. ly update data (say once/5min) present in all servers to have consistent data everywhere

(4) Used only in sys.s where we can afford the delay this async. update causes

eg:- 1 user in US writes post on LinkedIn. 1 user in India reads this post



reason → server is closer (not like the cases where server is on other side of world so you gotta make a time consuming round trip to 1st write data (writer) and then read data (reader))

NOTE: vice versa Indian writers' posts are also read by US readers 5 mins after post is published

eg:- If you don't need result of a deployment of new feature in your app to be seen all over the world instantly.

Date:
M T W T F S S

II Sharding

① It may so happen that main dB is handling way more reqs (say, 1 billion/sec) than it should be and thus, a bottleneck starts forming at main dB server

② ∴ We split up main dB data into small portⁿs of data and store em in separate small servers (known as shards) and use em to provide data to diff. kinds of reqs.

hori. scaling of a server

③ Shards a.k.a. 'data portiⁿs'
Sharding : Act of splitting a dB into 2 or more pieces (shards)
Advantage : Throughput ↑ (as no bottleneck formed)

eg:- say we got an rdB → table

- certain rows stored in shard 1 (eg:- user's name starts frm A-M)
- other rows stored in shard 2 (user's name starts frm N-Z)

bad sharding strategy as no. of user's with names frm A,B,C >> X,Y,Z

④ Popular sharding strategies

i) Based on clients' regions
ii) Based on type of data being stored
eg:- data → split → shard 1 → user data
→ shard 2 → payments data
iii) Sharding based on the hash of a column (only done for structured data)

⑤ Hotspots :- Certain shards that get more traffic than other shards cuz of the kinda data they store.

Cause → shardin key or hashin f^n is sub-optimal
Effect → uneven distri. of workload across shards

⑥ Minimizin hotspots i.e. evenly splittin up data

i) Use a hashin f^n (H.F.) that guarantees uniformity in determinin what data goes to which shard.

ii) Also, we need to ensure this H.F. doesn't change much to ensure that a particular type of data always goes to the same server

iii) Consistent hashin is kinda useful (H.F is const)

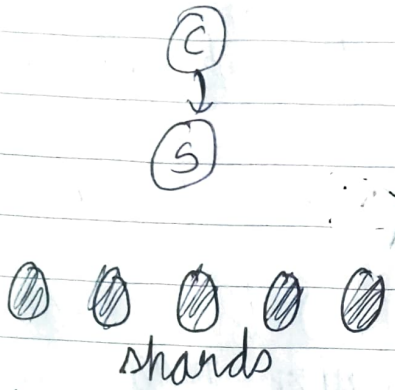
- Addin a new shard → consistent hashin minimizes the pieces of data we gotta migrate from existin shards to new shard
- Shard goes down → consistent hashin can't do anythin
  - soln :- replicate each shard.

∴ Give a lotta thought to comin up with a good logic to split up data while designin a sys.

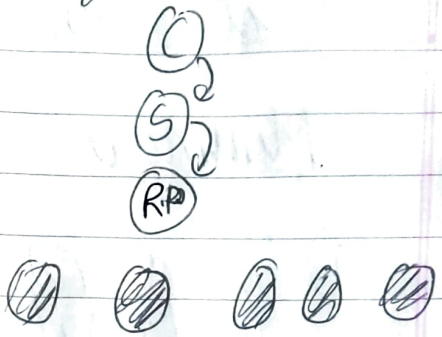
Date:
M T W T F S S

(7) Storage of shardin strategy logic

M-1 ⇒ stored in server



M-2 → stored in reverse proxy b/w server & shards



Place of storage basically decides that based on clients req which shard is it → gonna req. data frm
OR
→ write data to

```
//REVERSE PROXY - STORES LOGIC TO SPLIT INCOMING DATA (WRITE REQUEST) INTO SHARDS

const axios = require('axios');
const express = require('express');
const SHARD_ADDRESSES = ['http://localhost:3000', 'http://localhost:3001'];
const SHARD_COUNT = SHARD_ADDRESSES.length;

function getShardEndpoint(key) {
  const shardNumber = key.charCode(0) % SHARD_COUNT;
  const shardAddress = SHARD_ADDRESSES[shardNumber];
  return `${shardAddress}/${key}`;
}

app.post('/:key', (req, res) => {
  const shardEndpoint = getShardEndpoint(req.params.key);
  console.log(`Forwarding to ${shardEndpoint}`);
  axios.post(shardEndpoint, req.body).then((innerRes) => res.send());
});

app.get('/:key', (req, res) => {
  const shardEndpoint = getShardEndpoint(req.params.key);
  console.log(`Forwarding to ${shardEndpoint}`);
  axios.get(shardEndpoint).then((innerRes) => {
    if (innerRes.data === null) {     res.send('null');     return;    }
    res.send(innerRes.data);
  });
});

app.listen(8000, () => {  console.log(`Listening on port 8000`);});
```

```
//TO BASICALLY FORWARD DATA TO REVERSE PROXY
const express = require('express');
const fs = require('fs');
const app = express();
const PORT = process.env.PORT;
const DATA_DIR = process.env.DATA_DIR;
app.use(express.json());

app.post('/:key', (req, res) => {
  const { key } = req.params;
  console.log(`Storing data at key ${key}`);
  fs.writeFileSync(destinationFile, req.body.data);
  res.send();
});

app.get('/:key', (req, res) => {
  const { key } = req.params;
  console.log(`Storing data at key${key}`);
  const destinationFile = `${DATA_DIR}/${key}`;
  try {    const data = fs.readFileSync(destinationFile);    res.send(data);  }
  catch (e) {    res.send('null');  }
});

app.listen(PORT, () => {  console.log(`Listening on port ${PORT}`);});
```

replication_and_sharding — node aedb.js — 93×24

~/Documents/Content/Design_Fundamentals/Examples/replication_and_sharding — node aedb.js

```
Clements-MBP:replication_and_sharding clementmihailescu$ DATA_DIR=aedb_data_0 PORT=3000 node aedb.js
Listening on port 3000!
```

replication_and_sharding — node aedb.js — 93×24

~/Documents/Content/Design_Fundamentals/Examples/replication_and_sharding — node aedb.js

```
Clements-MBP:replication_and_sharding clementmihailescu$ DATA_DIR=aedb_data_1 PORT=3001 node aedb.js
Listening on port 3001!
Storing data at key a.
```

replication_and_sharding — node aedb_proxy.js — 93×24

~/Documents/Content/Design_Fundamentals/Examples/replication_and_sharding — node aedb_proxy.js

```
Clements-MBP:replication_and_sharding clementmihailescu$ node aedb_proxy.js
Listening on port 8000!
Forwarding to: http://localhost:3001/a
```

replication_and_sharding — -bash — 93×24

~/Documents/Content/Design_Fundamentals/Examples/replication_and_sharding — -bash

```
Clements-MBP:replication_and_sharding clementmihailescu$ curl --header 'Content-Type: application/json' --data '{"data": "This is some data."}' localhost:8000/a
Clements-MBP:replication_and_sharding clementmihailescu$ curl -w "\n" localhost:8000/a
```

replication_and_sharding — node aedb.js — 93×24

~/Documents/Content/Design_Fundamentals/Examples/replication_and_sharding — node aedb.js

```
Clements-MBP:replication_and_sharding clementmihailescu$ DATA_DIR=aedb_data_0 PORT=3000 node aedb.js
Listening on port 3000!
Storing data at key b.
Retrieving data from key b.
Retrieving data from key b.
Storing data at key bar.
Storing data at key baz.
Storing data at key foobar.
Storing data at key foobaz.
```

replication_and_sharding — node aedb.js — 93×24

~/Documents/Content/Design_Fundamentals/Examples/replication_and_sharding — node aedb.js

```
Clements-MBP:replication_and_sharding clementmihailescu$ DATA_DIR=aedb_data_1 PORT=3001 node aedb.js
Listening on port 3001!
Storing data at key a.
Retrieving data from key a.
Retrieving data from key a.
```

replication_and_sharding — node aedb_proxy.js — 93×24

~/Documents/Content/Design_Fundamentals/Examples/replication_and_sharding — node aedb_proxy.js

```
Clements-MBP:replication_and_sharding clementmihailescu$ node aedb_proxy.js
Listening on port 8000!
Forwarding to: http://localhost:3001/a
Forwarding to: http://localhost:3001/a
Forwarding to: http://localhost:3000/b
Forwarding to: http://localhost:3000/b
Forwarding to: http://localhost:3001/a
Forwarding to: http://localhost:3000/b
Forwarding to: http://localhost:3000/bar
Forwarding to: http://localhost:3000/baz
Forwarding to: http://localhost:3000/foobar
Forwarding to: http://localhost:3000/foobaz
```

replication_and_sharding — -bash — 93×24

~/Documents/Content/Design_Fundamentals/Examples/replication_and_sharding — -bash

```
Clements-MBP:replication_and_sharding clementmihailescu$ curl --header 'Content-Type: application/json' --data '{"data": "This is some data."}' localhost:8000/a
Clements-MBP:replication_and_sharding clementmihailescu$ curl -w "\n" localhost:8000/a
This is some data.
Clements-MBP:replication_and_sharding clementmihailescu$ curl --header 'Content-Type: application/json' --data '{"data": "This is some data."}' localhost:8000/b
Clements-MBP:replication_and_sharding clementmihailescu$ curl -w "\n" localhost:8000/b
This is some data.
Clements-MBP:replication_and_sharding clementmihailescu$ curl -w "\n" localhost:8000/a
This is some data.
Clements-MBP:replication_and_sharding clementmihailescu$ curl -w "\n" localhost:8000/b
This is some data.
Clements-MBP:replication_and_sharding clementmihailescu$ curl --header 'Content-Type: application/json' --data '{"data": "This is some data."}' localhost:8000/bar
Clements-MBP:replication_and_sharding clementmihailescu$ curl --header 'Content-Type: application/json' --data '{"data": "This is some data."}' localhost:8000/baz
Clements-MBP:replication_and_sharding clementmihailescu$ curl --header 'Content-Type: application/json' --data '{"data": "This is some data."}' localhost:8000/foobar
Clements-MBP:replication_and_sharding clementmihailescu$ curl --header 'Content-Type: application/json' --data '{"data": "This is some data."}' localhost:8000/foobaz
Clements-MBP:replication_and_sharding clementmihailescu$
```

Date:
M T W T F S S

Leader Elec^{n}

Gonna understand entire concept thru one of its actual use cases

- Case → Managing subscrip^{n}s and payments in some sort of service such as Amazon Prime, Netflix (subscrip^{n} on recurring basis)

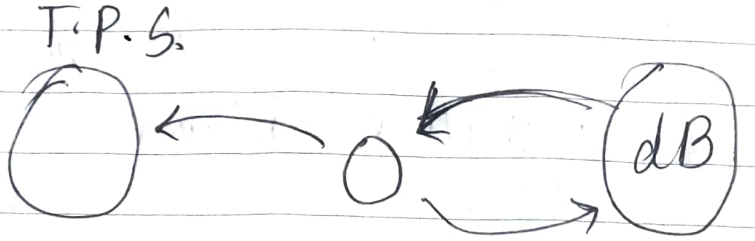
(1) dB → stores data pertainin to subscrip^{n} and users
- → is user curr. ly subscribed?
- → date for subscrip^{n} renewal
- → price

(2) 3rd party service (T.P.S.) → Service that conducts the payment transac^{n} (debit frm user → credit to you) eg. Paypal, Stripe
Needs to comm. with dB to know
- → when a user shud be charged again
- → what the charge?

No direct connec^{n} b/w dB and 3rd party service
- → diff. to implement
- → dB is pretty sensitive part of sys. Contains imp info. and its not wise to divulge all of it by allowin a T.P.S. to connect directly

Date:
M T W T F S S

(3) Service in middle → talks to dB periodically
→ figures out when a user's subscrip" is gonna renew
→ how much to charge the user
→ transmits only relevant info. to T.P.S.



Optimizing this setup.

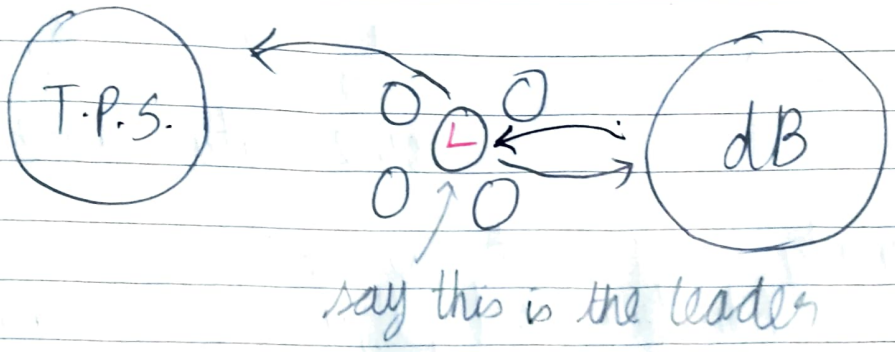
1) If the server in the middle fails, entire payment sys. collapses
We introduce 'redundancy' (passive) in our sys. by hori.ly scaling the service in the middle i.e. have 5 instead 1 server in the middle to do the transac" logic

2) All 5 servers are doin the same thing i.e. askin dB for transac" info and req. in T.P.S. to charge the user
We don't wanna duplicate the req. to T.P.S. to charge the user i.e. make that req. just once

3) Leader elec" → If you hv a grp of machine (in our case - servers) that are in charge of doin the same, instead of havin them all do that thing, we elect 1 of the machines as leader

Network parti^{n} → a some network failure that makes some machines no longer be able to comm. with other machines

and only that machine is gonna be responsib. for doin the opns. that all of the machines are built to do.



Other servers (followers) → just on standby to take over when leader fails i.e. a new leader is elected frm amongs these servers and it takes over.

4) Complexity of leader elec^{n}: Leader elec^{n}'s logic is complex cuz:

- → multip machines are made to elect a single leader (consensus - agree upon somethun together)
- → They all gotta be aware of who the leader is at any given time
- → They all gotta be capable of re-electin a new leader in case curr. leader

Difficulty of logic mainly lies in

- → multip machines that are distru.ed are sharin state. Diff. cuz ye dunno what might happen in a network (eg:- network parti^{n}
- → make multip machines gain consensus

Here, gainin consensus → agreein who's the curr. leader

5) Consensus Algo. → Complex, math heavy algos that allow multip nodes (servers) in a cluster (grp) to reach consensus i.e. agree upon some data-value.

eg:- Paxos, Raft

6) Usually, ppl just use some pre-existin 3rd party industry tool that offers them this logic whilst itself runnin a consensus algo under the hood

eg:- Zookeeper, Etcd → help implem. leader elec^n v. easily.

7) Leader elec^n usin Etcd:

(1) Etcd → k-v store
→ highly available
→ strongly consistent

Consistency → if you got multip machine (s), readin & writin to the same k-v pair in the k-v store, you're always gonna get the same, correct value irresp of when or from which machine the k-v pair is accessed.

(2) Etcd achieves high availability, strong consistency by implementin consensus algo - Raft

(3) There are gonna be multip machines that can read, write to the main k-v store

that Etcd supports. These machines need:

- high availability: To know which other machines are available after a leader dies
- strong consistency: single source of truth for all k-v pairs in store

(4) In Etcd each k-v pair can be visualised be of form:

key	value
status of machine (i.e. is it a leader or a follower)	name / IP address (basically a UID for a machine in grp i.e. node in cluster)

∴ we'll hv 1 k-v pair as follows

"leader": UID for this machine

↳ k-v pair represents curr "leader"

(5) All machines (in our case - servers) comm. with k-v store any given pt. in time we gotta hv a leader present as k-v pair in the store else we gotta elect a new one.

```
import etcd3
import time
from threading import Event

# The current leader is going to be the value with this key
LEADER_KEY = "/algoexpert/leader"

# Entrypoint of the program

def main(server_name):
    # Create a new client to etcd
    client = etcd3.client(host="localhost", port=2379)

    while True:
        is_leader, lease = leader_election(client, server_name)

        if is_leader:
            print("I am the leader.")
            on_leadership_gained(lease)
        else:
            print("I am a follower.")
            wait_for_next_election(client)

# This election mechanism consists of all clients trying to put their name into a single key, but in a way that only works if the key does not exist(or has expired before)

def leader_election(client, server_name):
    print("New leader election happening.")
 # Create a lease before creating a key. This way, if this client ever lets the lease expire, the keys associated with that lease will all expire as well.
 Here, if the client fails to renew lease for 5 seconds (network partition or machine goes down), then the leader election key will expire.
    lease = client.lease(5)

    # Try to create the key with your name as the value. If it fails, then another serer got there first
    is_leader = try_insert(client, LEADER_KEY, serve_name, lease)
    return is_leader, lease


def on_leadership_gained(lease):
    while True:
# As long as this process is alive and we're the leader, we try to renew the lease. We don't give up the leadership unless the process/machine crashes or some exception is raised
        try:
            print("Refershing lease; still the leader.")
            lease.refresh()
# This is where the business logic would go (eg: ask 3rd part service to charge user based on Db INFO)
            do_work()
        except Exception:
# Here we most likely got a client timeout (from network issue). Try to revoke the current lease so another member can get leadership
            lease.revoke()
        except KeyboardInterrupt:
```

```
        print("\n Revoking lease; no longer the leader")
        # Here we're killing the proccess. Revoke the lease and exit
        lease.revoke()
        sys.exit(1)


def wait_for_next_election(client):
    election_event = Event()

    def watch_callback(resp):
        for event in resp.events:
# For each event in the watch event, if the event is a deletion it means that the key expired / got deleted,
which means the leadership is up for grabs.
            if isinstance(event, etcd3.events.DeleteEvent):
                print("LEADERSHIP CHANGE REQUIRED")
                election_event.set()


watch_id = client.add_watch_callback(LEADER_KEY, watch_callback)

# While we haven't seen the that the leadership needs change, just sleep
try:
    while not election_event.is_set():
        time.sleep(1)
except KeyboardInterrupt:
    client.cancel_watch(watch_id)
    sys.exit(1)

# Cancel the watch; we see that the election should happen again.
client.cancel_watch(watch_id)

# Try to insert a key into etcd with a value and a lease. If the lease expires that key will get automatically
deleted behind mthe scenes. If that key was already present this will raise an exception.

def try_insert(client, key, value, lease):
    insert_succeeded, _ = client.transaction(
        failure=[],
        success=[client.transaction.put(key, value, lease)],
        compare=[client.transaction.version(key) == 0],
    )
    return insert_succeeded


def do_work():
    time.sleep(1)


if __name__ == "main":
    server.name = sys.argv[1]
    main(server_name)
```

leader_election — Python leader_election.py server1 — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server1

```
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
^C
Revoking lease; no longer the leader.
Clements-MBP:leader_election clementmihailescu$ python3 leader_election.py server1
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am a follower.
```

leader_election — Python leader_election.py server2 — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server2

```
Clements-MBP:leader_election clementmihailescu$ python3 leader_election.py server2
New leader election happening.
I am a follower.
^CClements-MBP:leader_election clementmihailescu$ python3 leader_election.py server2
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am a follower.
```

leader_election — Python leader_election.py server3 — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server3

```
Clements-MBP:leader_election clementmihailescu$ python3 leader_election.py server3
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
```

leader_election — -bash — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — -bash

```
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
^C
Revoking lease; no longer the leader.
Clements-MBP:leader_election clementmihailescu$
```

leader_election — Python leader_election.py server1 — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server1

```
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
^C
Revoking lease; no longer the leader.
Clements-MBP:leader_election clementmihailescu$ python3 leader_election.py server1
New leader election happening.
I am a follower.
```

leader_election — Python leader_election.py server2 — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server2

```
Clements-MBP:leader_election clementmihailescu$ python3 leader_election.py server2
New leader election happening.
I am a follower.
^CClements-MBP:leader_election clementmihailescu$ python3 leader_election.py server2
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am a follower.
```

leader_election — Python leader_election.py server3 — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server3

```
Clements-MBP:leader_election clementmihailescu$ python3 leader_election.py server3
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am a follower.
```

leader_election — Python leader_election.py server4 — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server4

```
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
```

leader_election — -bash — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — -bash

```
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
^C
Revoking lease; no longer the leader.
Clements-MBP:leader_election clementmihailescu$
```

leader_election — Python leader_election.py server2 — 93×24

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server2

```
Clements-MBP:leader_election clementmihailescu$ python3 leader_election.py server2
New leader election happening.
I am a follower.
^CClements-MBP:leader_election clementmihailescu$ python3 leader_election.py server2
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am a follower.
```

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server3

```
Clements-MBP:leader_election clementmihailescu$ python3 leader_election.py server3
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am a follower.
```

~/Documents/Content/Design_Fundamentals/Examples/leader_election — Python leader_election.py server4

```
Clements-MBP:leader_election clementmihailescu$ python3 leader_election.py server4
New leader election happening.
I am a follower.
LEADERSHIP CHANGE REQUIRED
New leader election happening.
I am the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
Refreshing lease; still the leader.
```

each of these amts pts. to the relevant record (row) in the main dB table
Thus, we get our largest amt by simply goin to the end of the dBI table

simple way of lookin at dBI

- auxilliary data struc of main dB that speeds up gettin data frm main dB (read opns. faster)
- auxilliary data struc so extra space reqd. for storage
- whenev. you write data to main dB, you also gotta write to dBI (write opns. slower)
- In practice, we create an idx on 1 or multip cols. in our main dB

NOTE : Eventual Consistency

- A consistency model which is unlike Strong Consistency
- In this model, reads might return a stale view of the sys.
- An eventually consistent datastore will give guarantees that the state of the data will eventually reflect writes within a time period (eg:- 10 mins, 30 days)
- eg:- Google Cloud Datastore