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 foundaʼal 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, XCD, 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

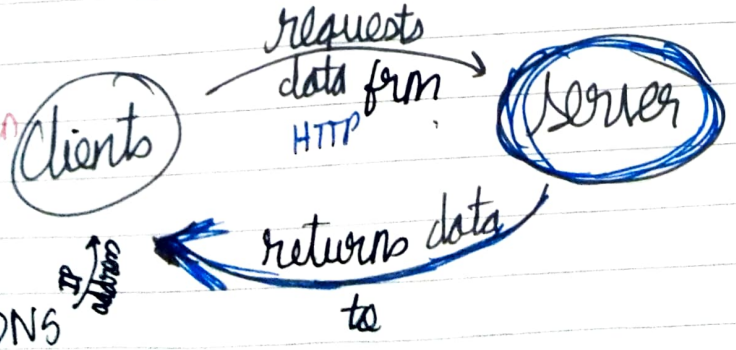
Foundaⁿ of modern internet, helps us understand how computers speak to one another. Paradigm consists of clients requestin service, data frm servers and servers providin it all

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 nw. 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

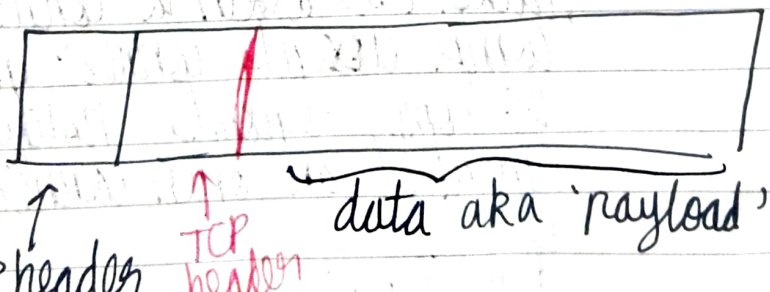
Network protocol consists of:
- types of msgs
- struc.
- order
- response to a msg, what shud it look like if present
- rules on when msgs can be sent to 1 another

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

devnr = 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. cuz its just data that fits into the underlyin IP packets

3) <u>HTTP - Hyper Text Transfer Protocol</u>

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

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

```
const httpReq = {
    host: 'localhost',        // describe destn. server
    port: 8080,
    method: 'POST',  //GET, PUT, DELETE, OPTIONS, PATC
    path: '/payments',        // describe purpose of req.
    headers: {                // here used to POST payments i.e pay
        'content-type': application/json,
        'content-length': 51
    },
    body: '{"data": "This is a low JSON format data piece"}'
}
```

POST → provides data → server
GET → retrieve data
DELETE → delete data

describe destn. server

describe purpose of req. They're guidelines subj to your server and how we em.

path → Client sends data to server which has various paths which dictate what kinda logic will occur for req eg:- auth

```
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.

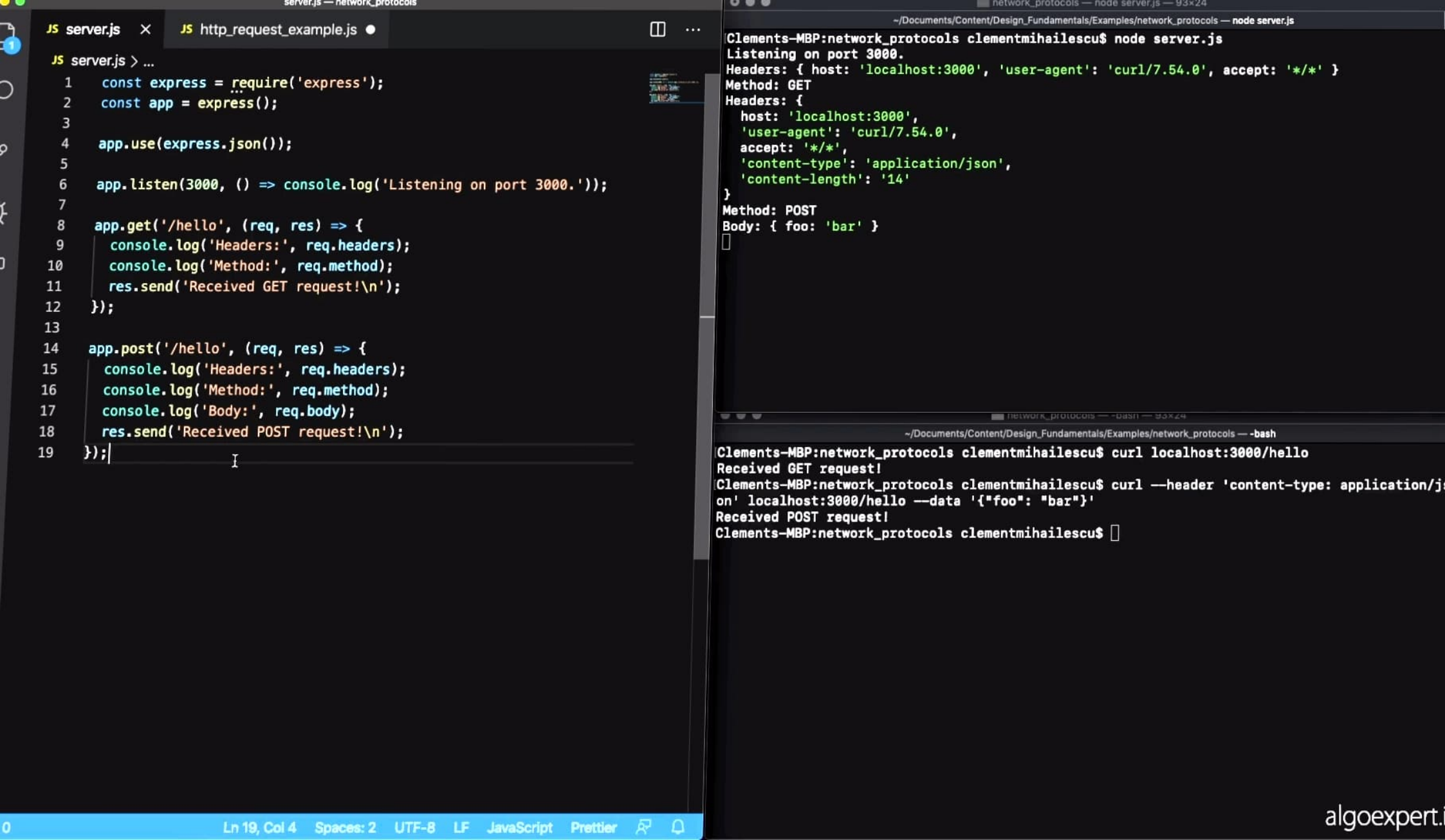
Typical HTTP req. schema

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

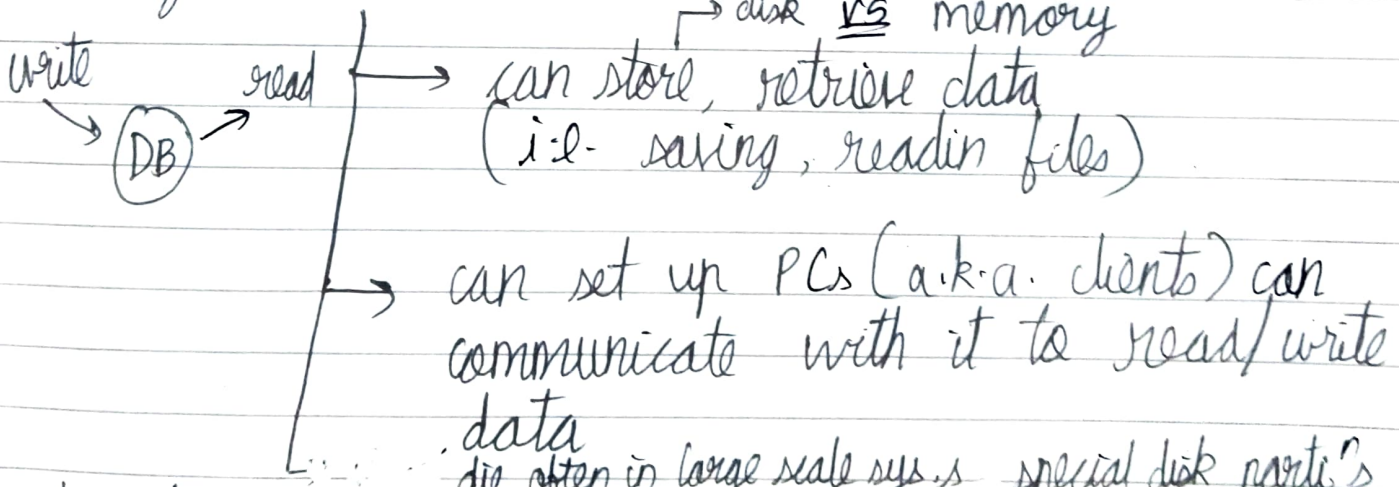
STORAGE

→ 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.

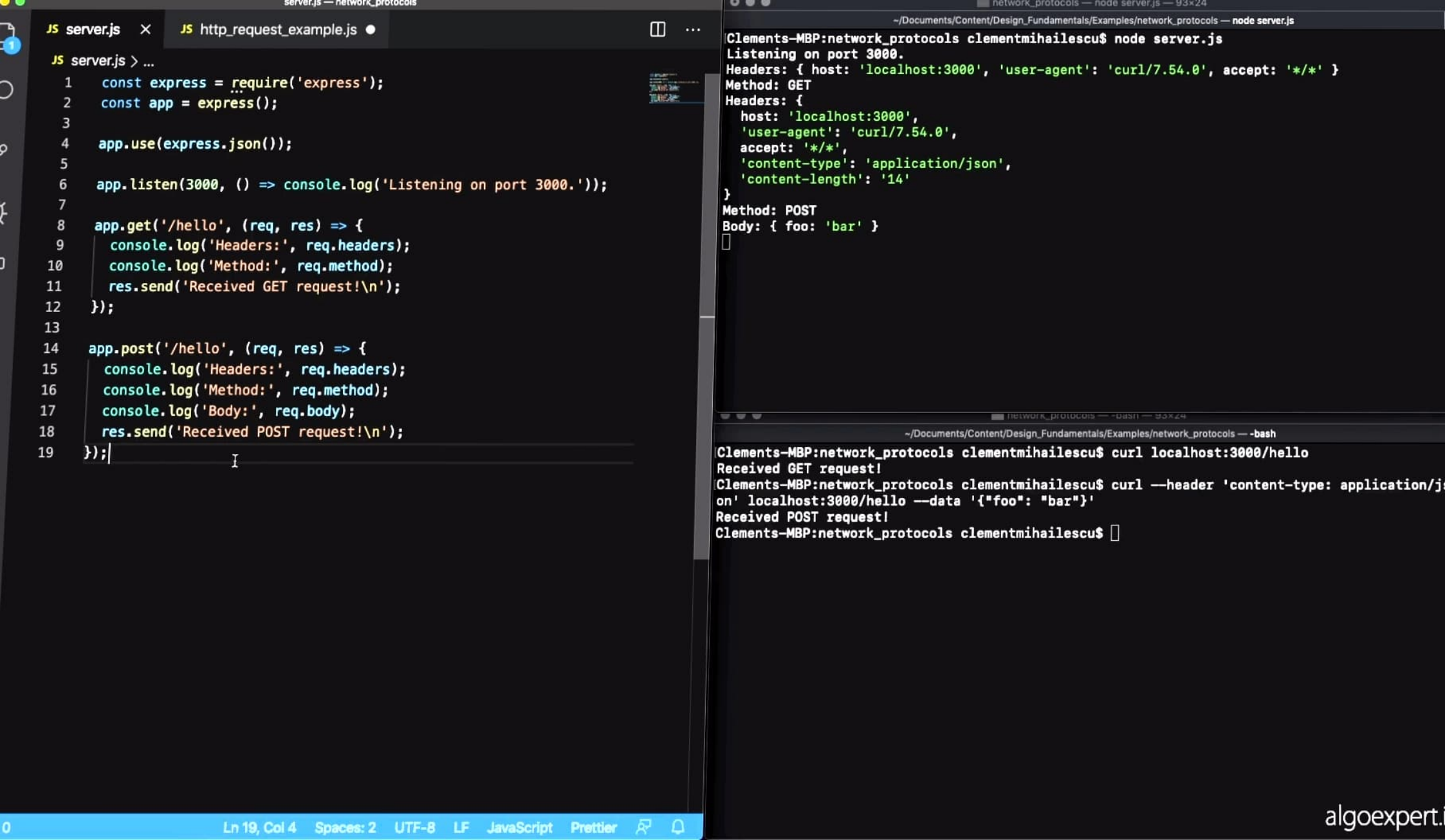
There are a lotta DB offerings to give options based on performance, data security

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

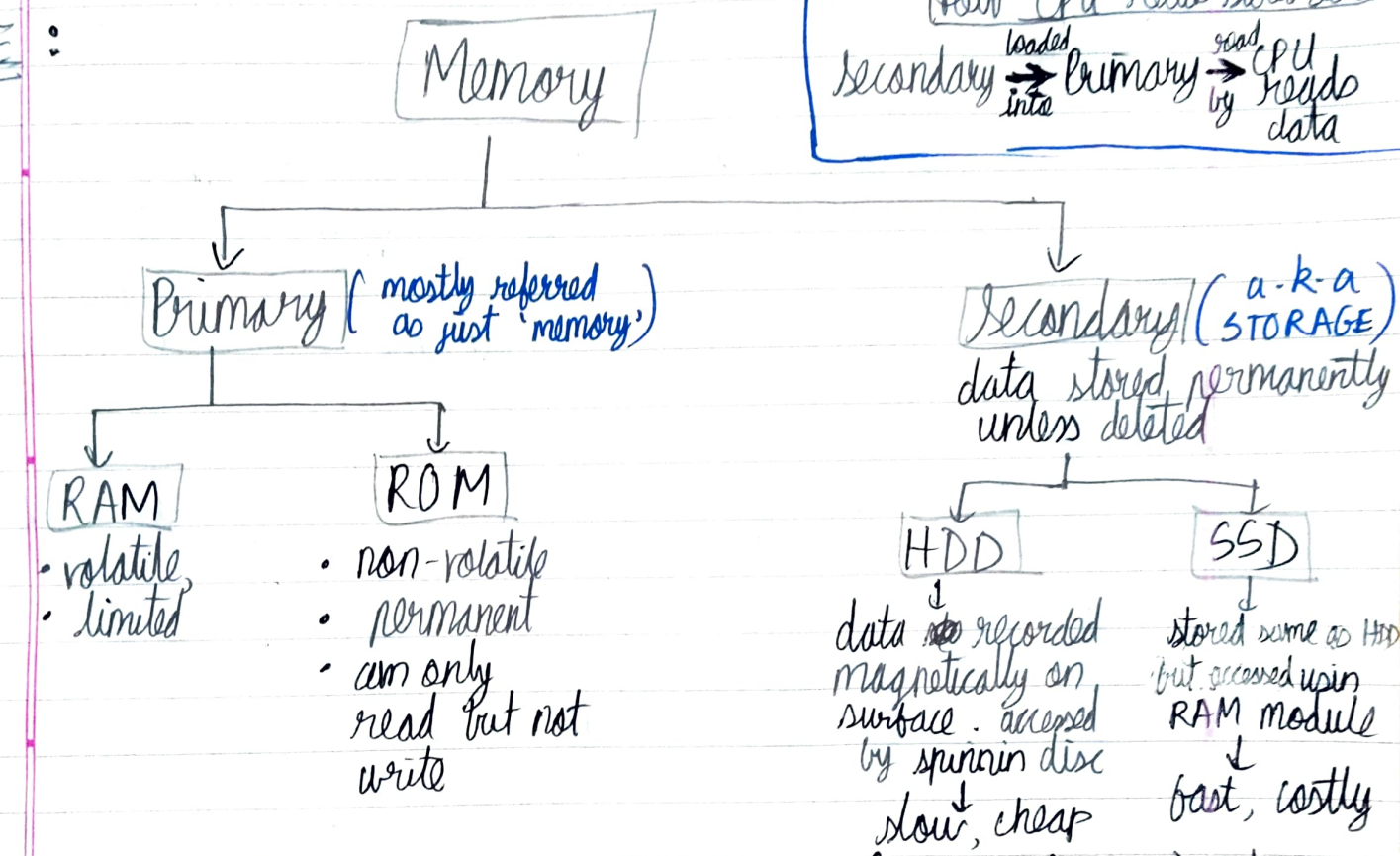

server.js — network_protocols
server.js
http_request_example.js
server.js > ...
1 const express = require('express');
2 const app = express();
3
4 app.use(express.json());
5
6 app.listen(3000, () => console.log('Listening on port 3000.'));
7
8 app.get('/hello', (req, res) => {
9 console.log('Headers:', 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

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

A] **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

- parts with lower latency
- parts with higher latency

iii) Some comparisons.

Part of sys + its f^n	Latency
Readin 1MB from memory (RAM)	$250 \mu s$
Readin 1MB from SSD	$1000 \mu s$
Sendin 1MB over 1Gbps network	$10^4 \mu s$ → (sendin to computer next to us i.e. dist. is not considered)
Readin 1MB from HDD	$2 \times 10^4 \mu s$
Sendin packet (≈ 1055 B) over network from California to Netherlands and then back to California	$15 \times 10^4 \mu s$

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

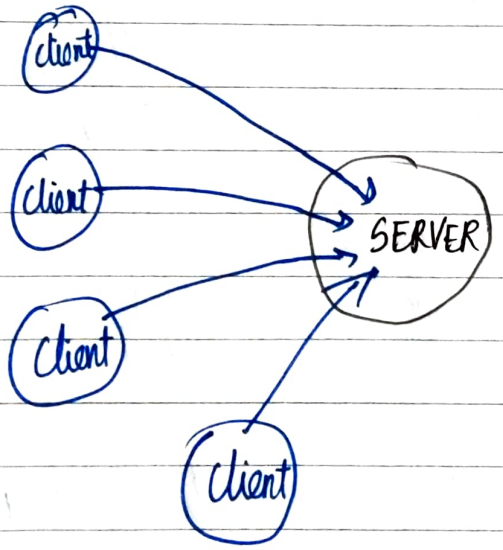
iii) Optimizin a sys → ↓ its overall latency

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.

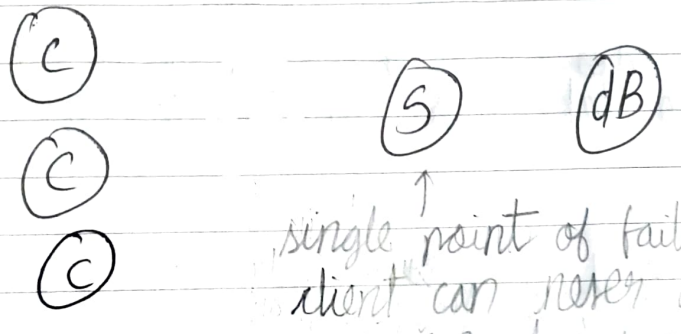
Doesn't really require HA, something that if fails, doesn't cause catastrophic losses

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.

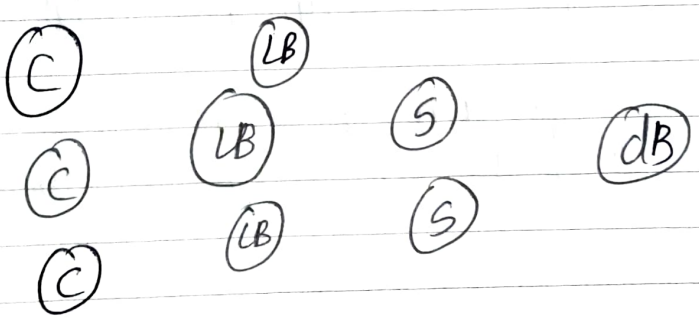
Simple sys ⟹



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

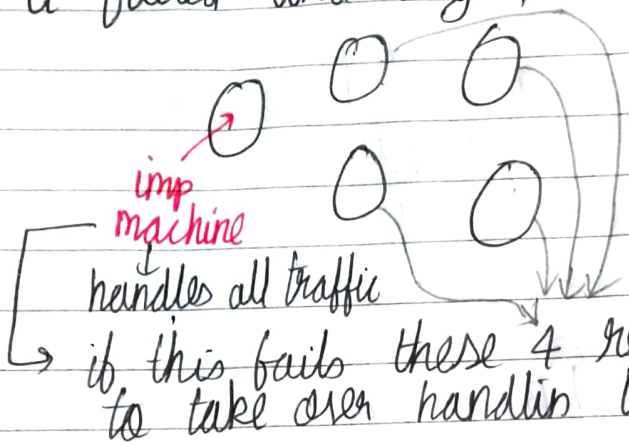
→ duplicated

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.)

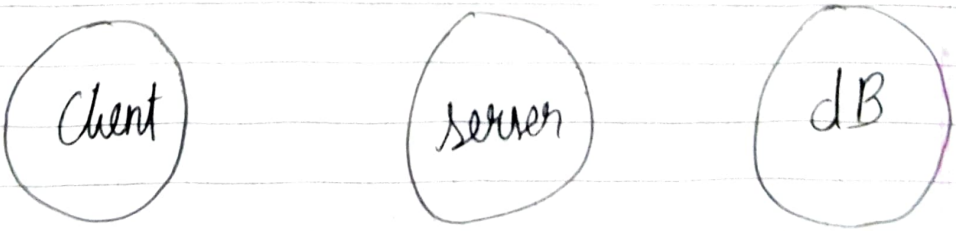
2] Caching in sys. → Used to reduce (improve) the latency of a sys. (speeds up the sys.)

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.

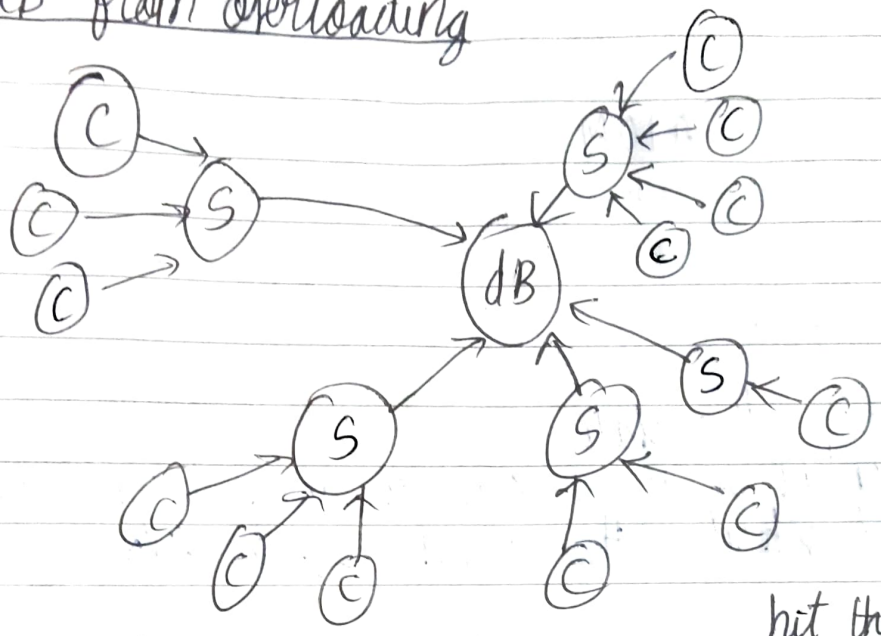
Caching speeds up sys. → 2) Avoid redoing a computationally long, bad TC, complex operation each time
eg:- say on net. req. from client → server,

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

3) dB optimized

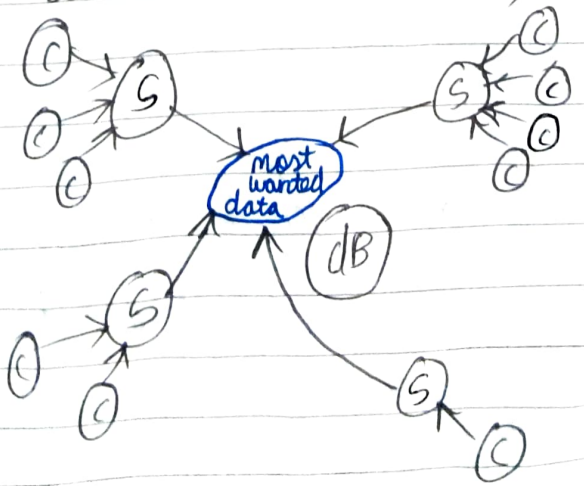
Prev. dB from overloading

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

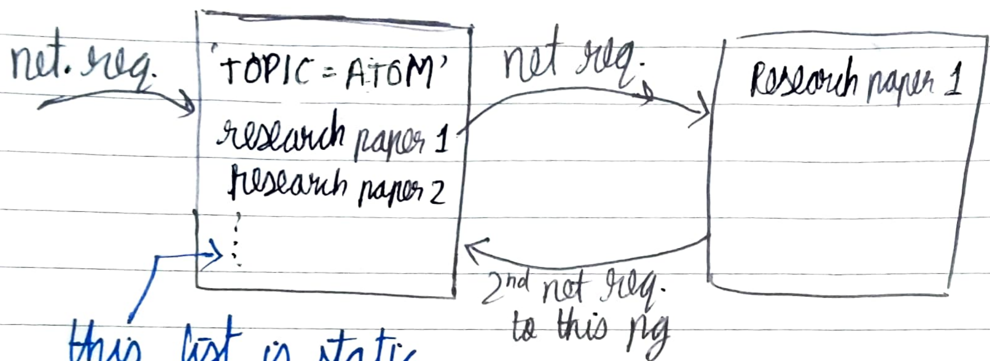
k-V = key value

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



this list is static 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 pass all test cases so no need to run, test

Cache stored → server level
OR
→ detached frm 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' ~~with~~ 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

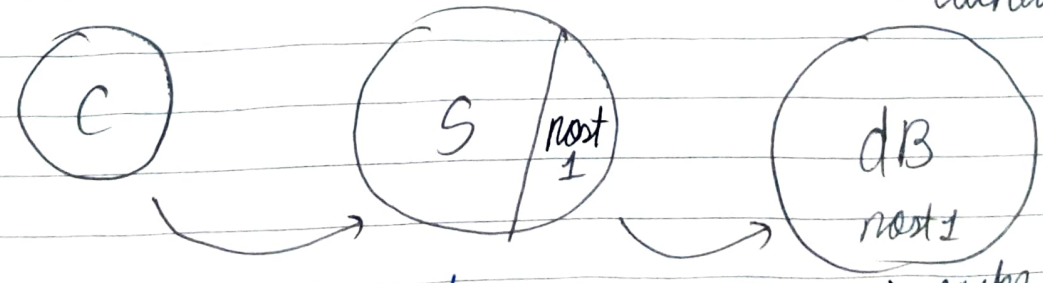
if not pre-stored → Code called is customer's code so run it. Takes 1 sec. to give result

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

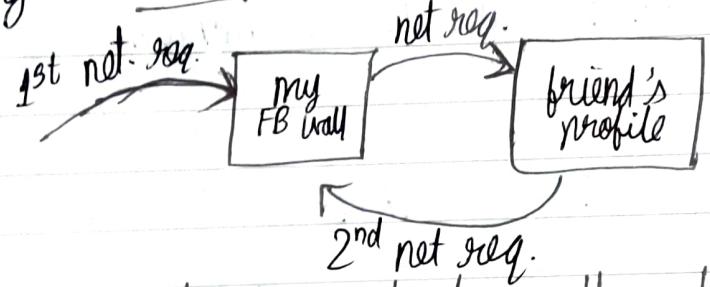
NOTE: WHEN TO USE CACHING?

✓ 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.

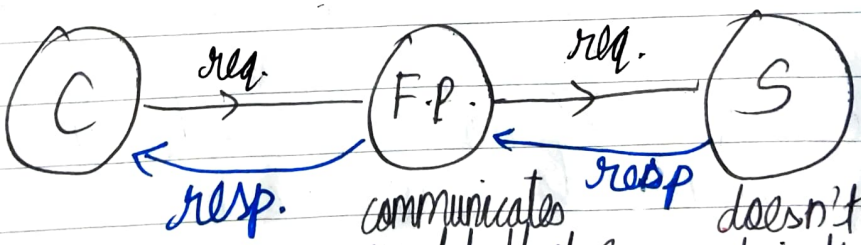
Choice of evicⁿ policy depends on kind of product you wanna do sys. des. for

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')

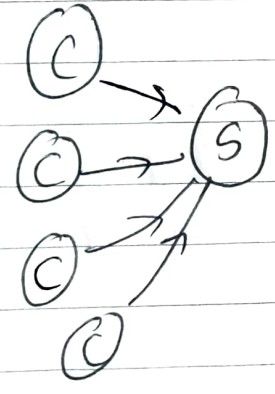
Load Balancers

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)

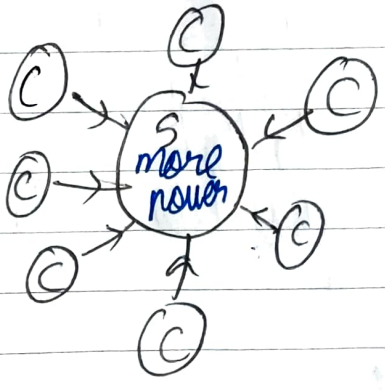
(ii) This prevents 1 server from getting too overloaded by too many client reqs.

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

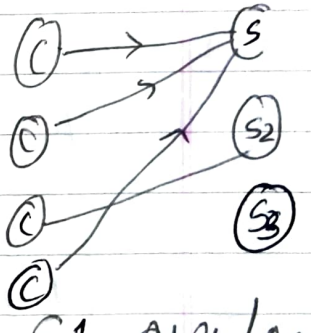
Case 1: Server overload due to lotta reqs.



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

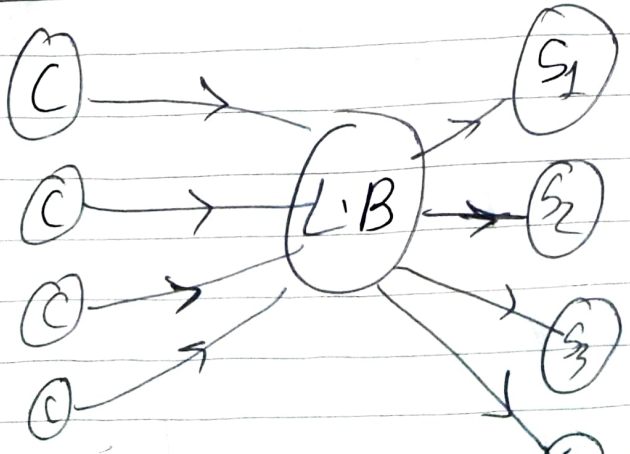


Case 3: Horizontal scaling
Multiple servers but still most client reqs to 1 server.



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

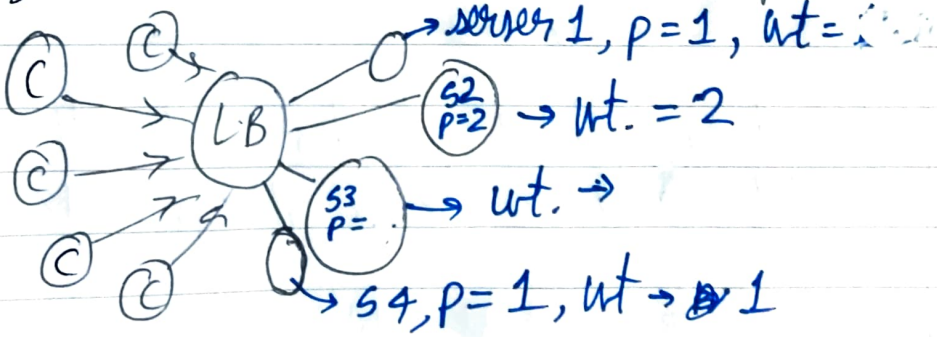
(ii) On addin / remov'n old server, it registers/ deregisters itself w.r.t. LB

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.

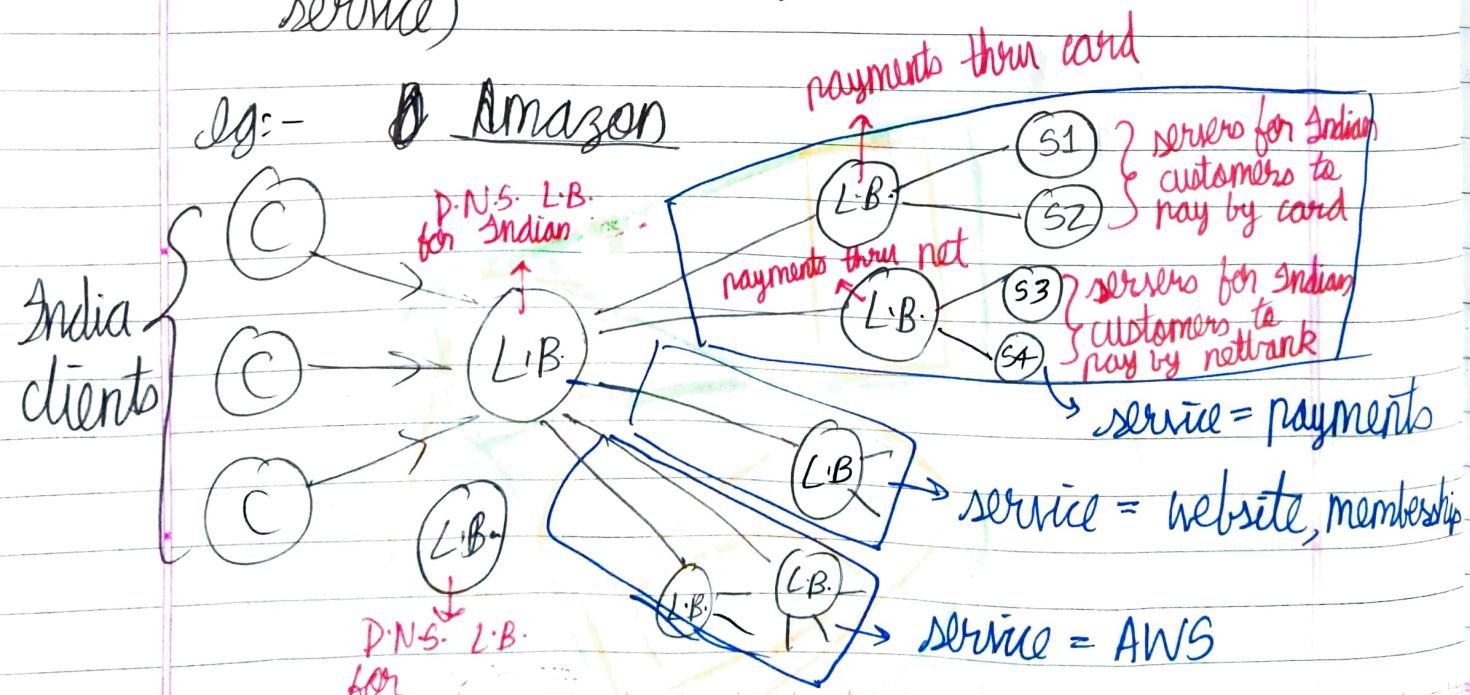
2) If one set of servers for a service start failing atleast other services are still up and running. (unaffected)

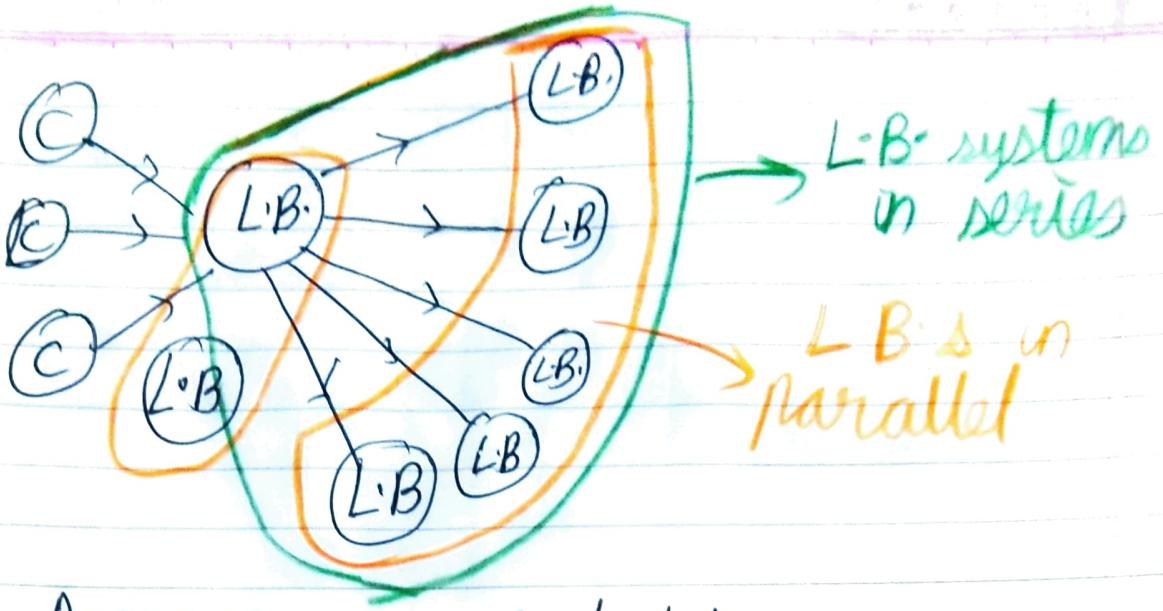
MULTIPLE LOAD BALANCERS

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

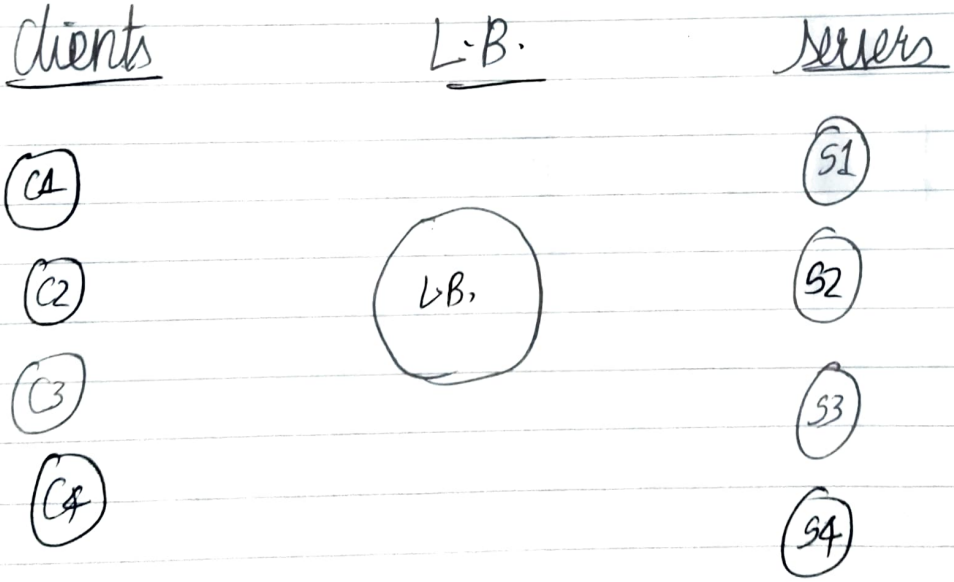
code on opp. page → shows wted R·R· selec^{n} meth.

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)

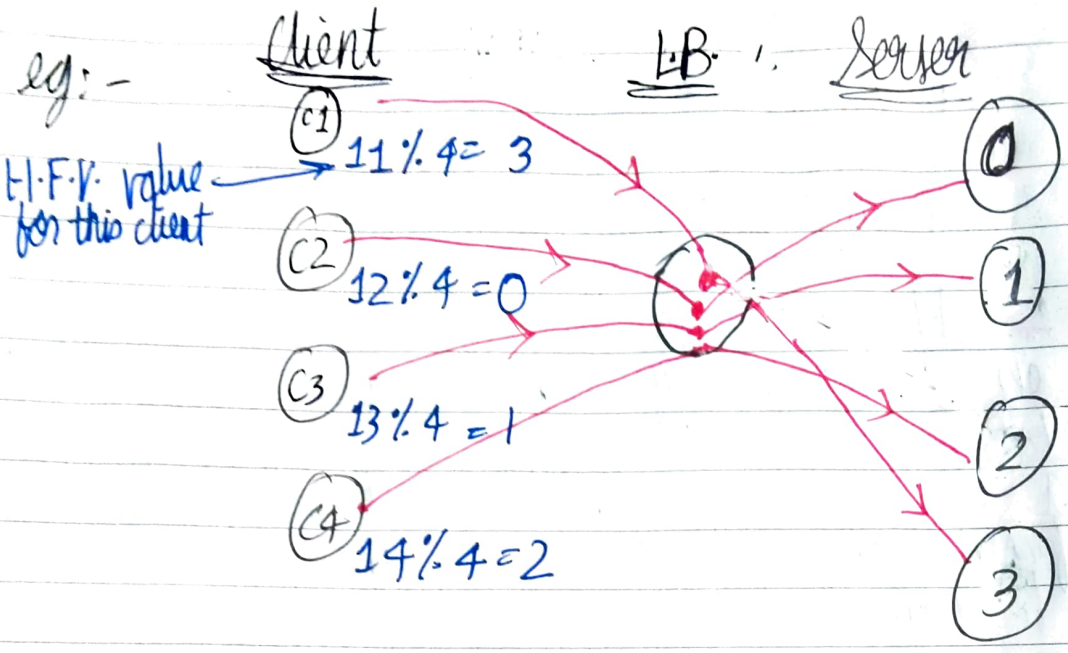
6) For this round robin or random server selection strategy don't work as they do not guarantee you that if the 1st time a client makes a req. and it gets processed and cached in S1, the 2nd time, when client makes the same req., he/she will be redirected to S1 only
∴ Chances of cache hits ↓

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

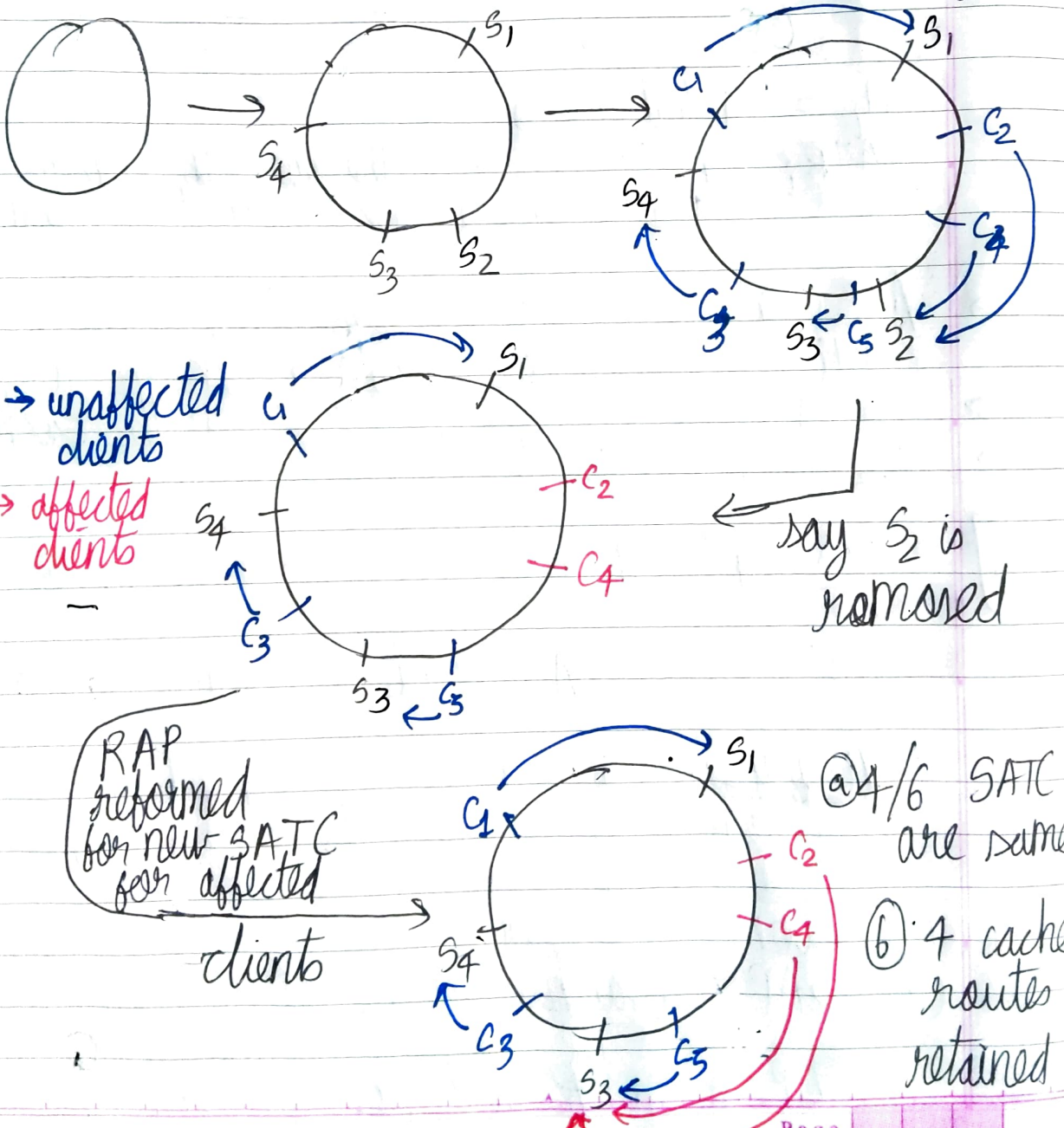
Visualiss^n : ① Servers are placed on a circle (preferably at equal dist. coverin entire circle in equal parts.
② Clients are now placed on the same circle
③ SAP → Matchin client to its closest server if we traverse circle in ? dir^n
SAP is performed to get SATC for each client

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:-



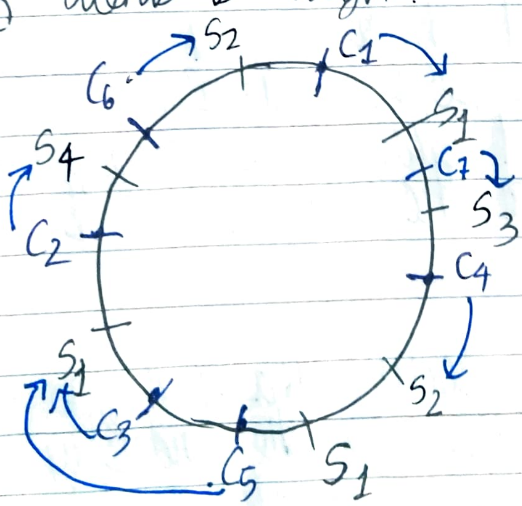
Thus, we retain altleast some cache routes irresp. of addiⁿ / removal of ^ server.

NOTE: ① Also, as visib, C_1 is very far from 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"s s.t. you minimize dist.s b/w all servers and clients mappin to em

② We can traverse circle in ↻ or ↺ dirⁿ. Dirⁿ 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"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.

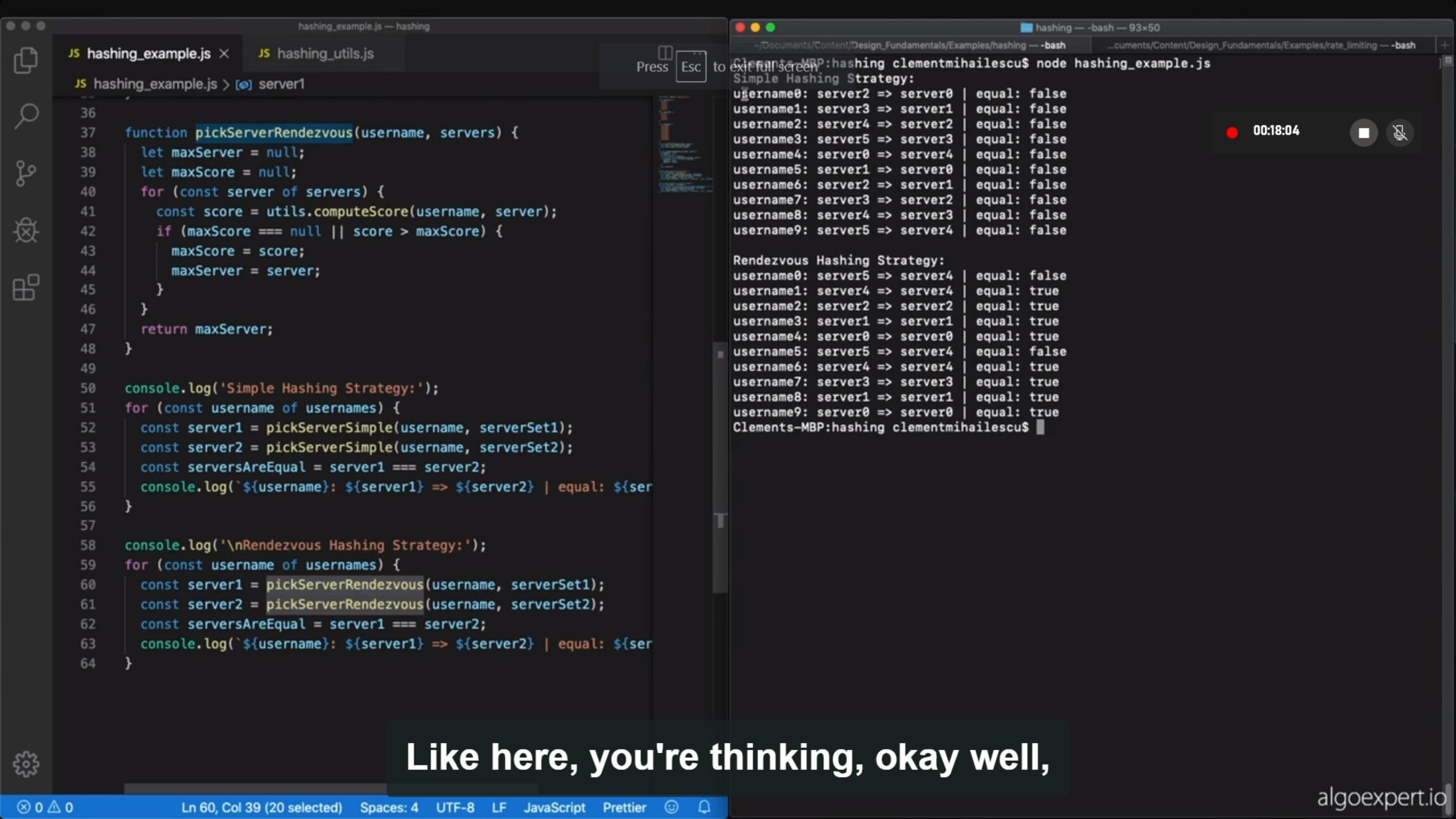
v) Adding a server → Re-ranking carried out for clients but new server will be pushed as H.R.S. only in 1 ranking so 1 client's destn. server will change but for other clients' destn. servers won't change so cache routes won't change so reqs. will still keep getting to the server with the in-mem. cache.

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.

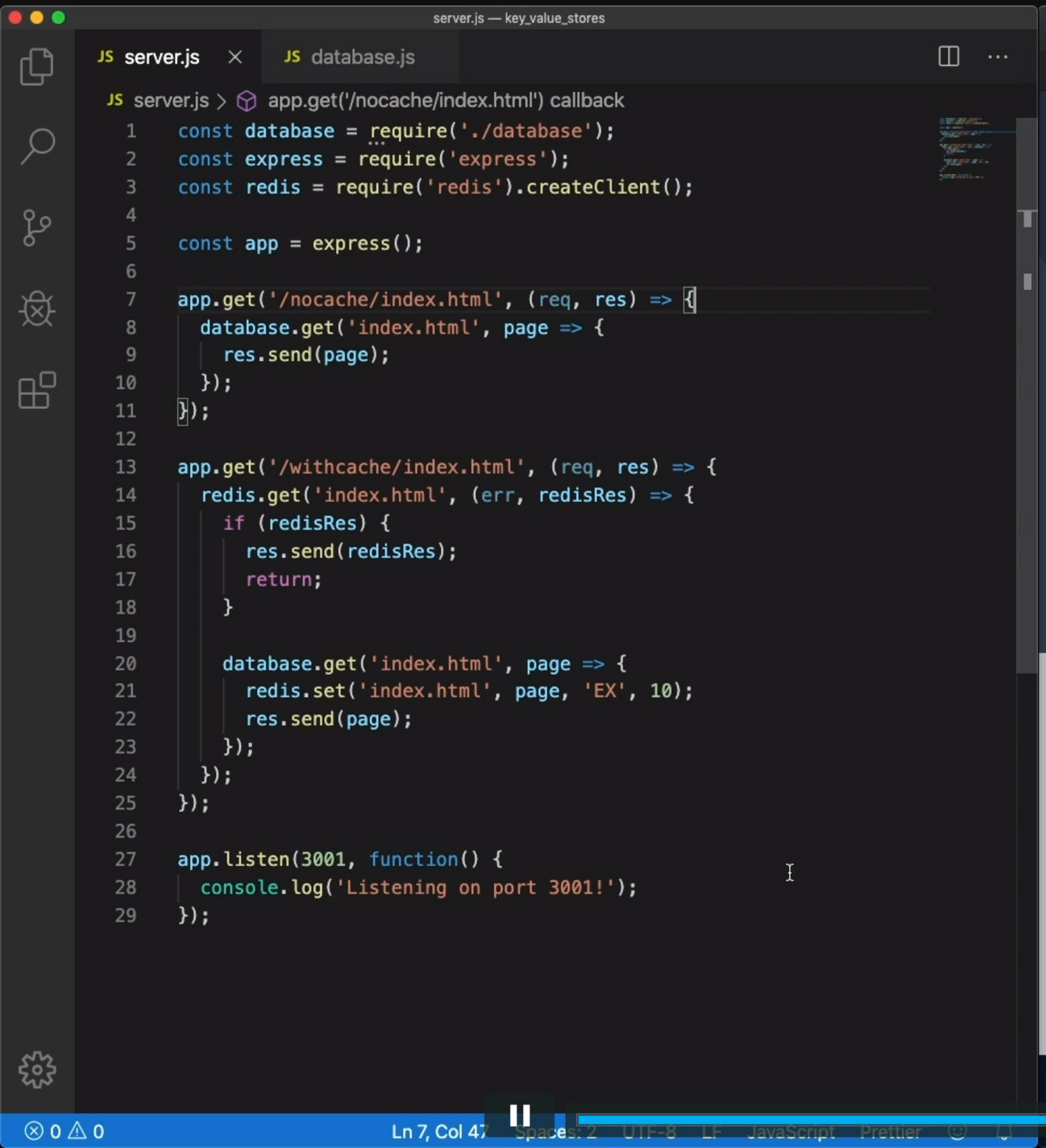
---

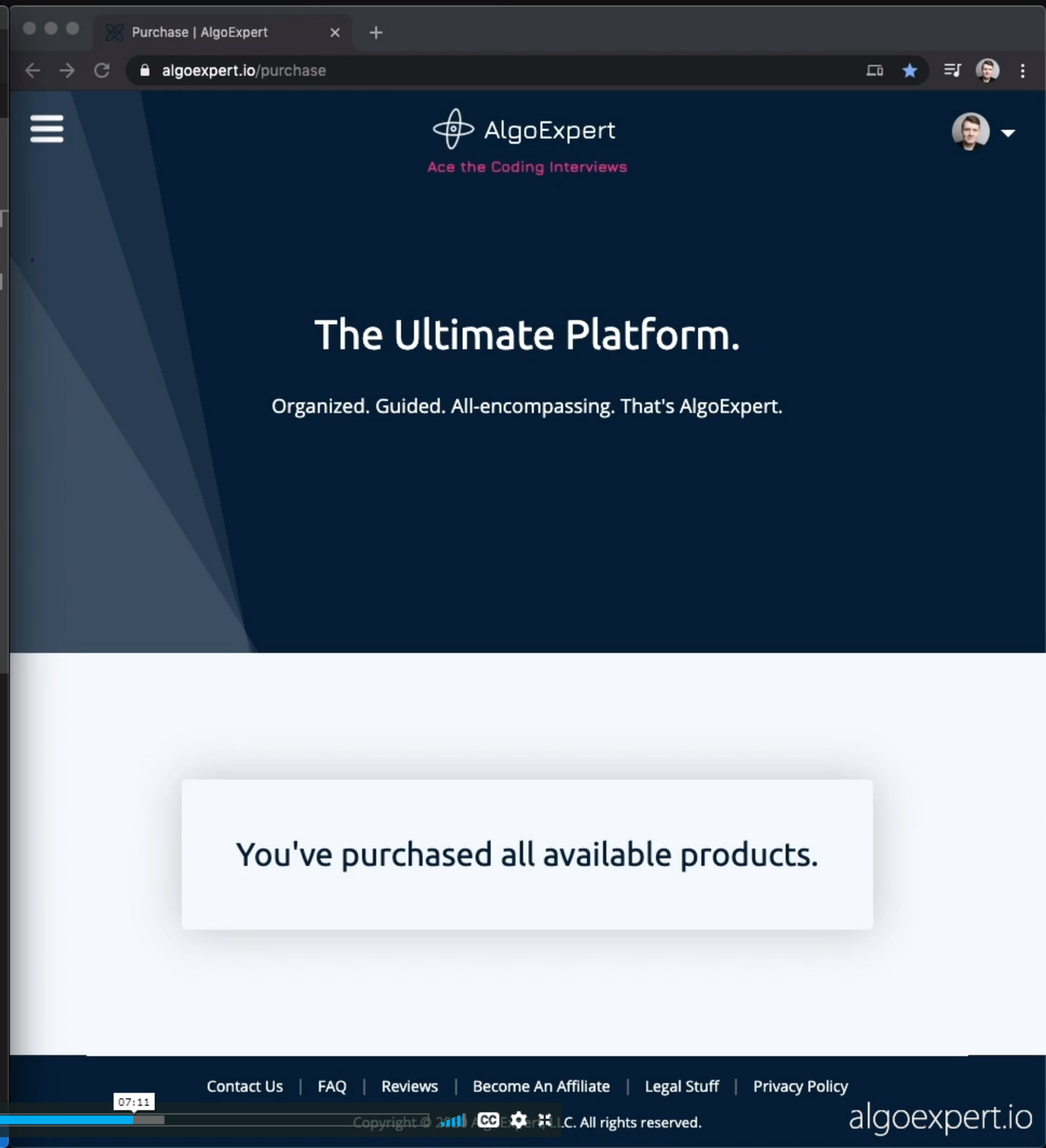
eg' of k-v stores:

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

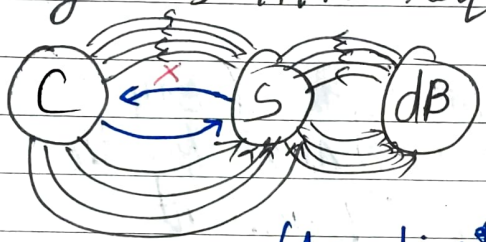
RATE LIMITING

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.

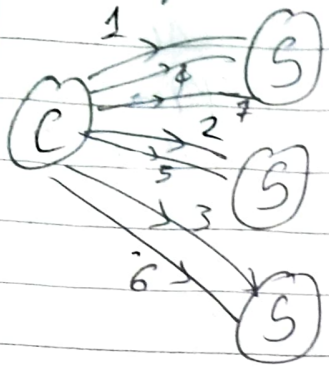
(5) DDoS attack → Distributed DoS attack

(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.

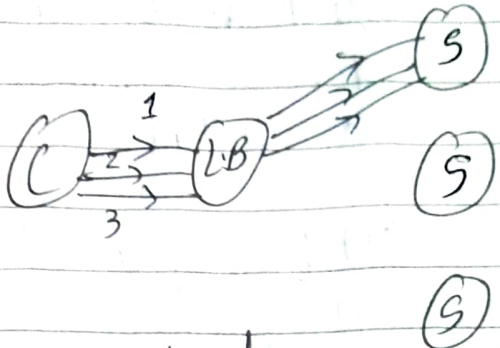
(6) In large-scale distri sys → we gotta hv. v. powerful rate limiting ~~to prot~~ → i.e. we need to hv really powerful load balancin to ensure that a client's req are always routed to the same server (amongst a set of servers cuz its a big sys) to ensure that we have some prev access records of this 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

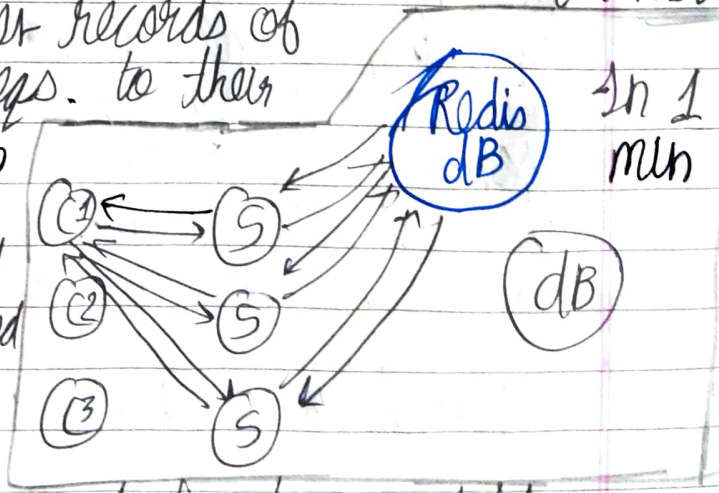


Client always routed to same server → it has prev access records of this client so it'll rate lim. client if it's floodin with too many reqs

∴ 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

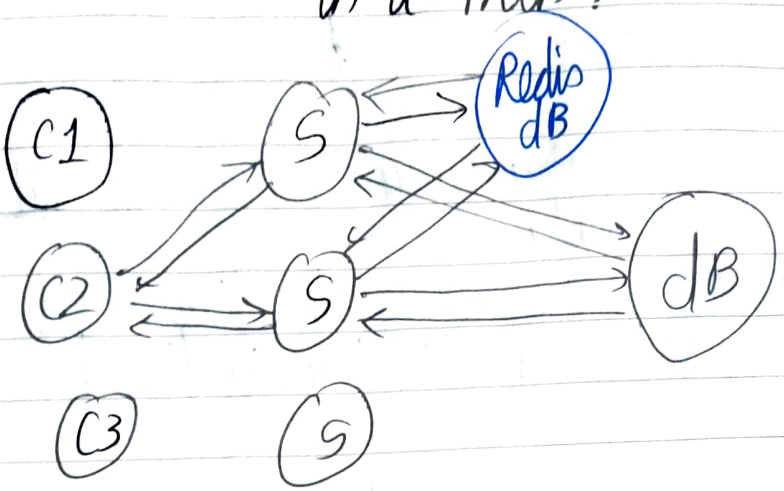


eg! - Redis. → eg 3 req/min allowed

↓ 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

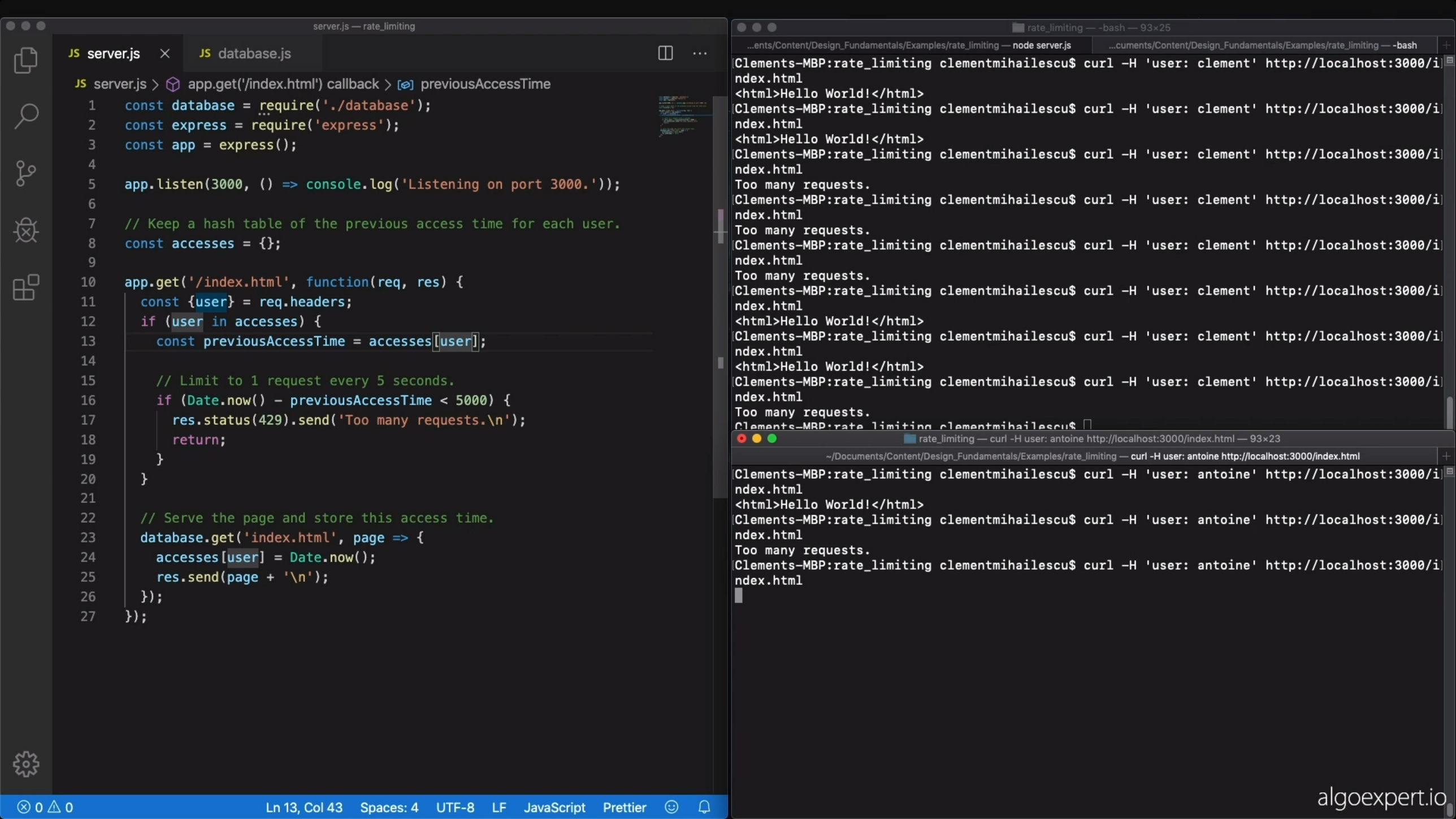
(8) Tier based Rate Limiting

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

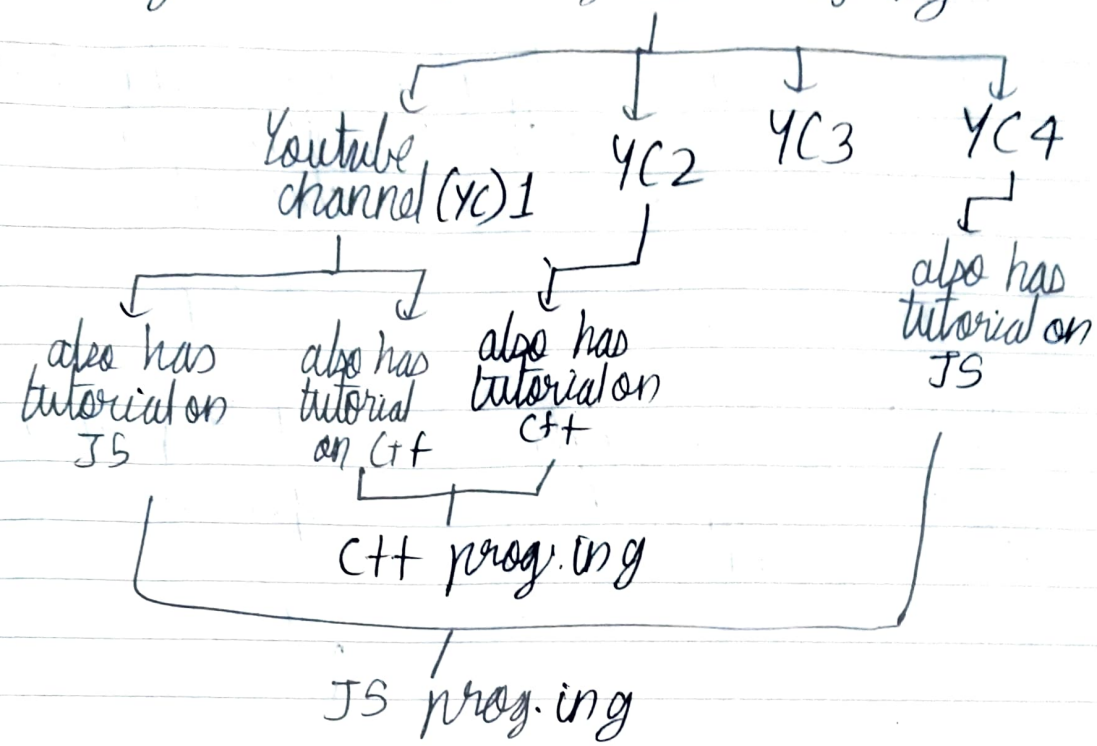
(iii) Graph dB → ① dB built on top of the graph data model.
② ∴ the concept of a relationship b/w indiv data pts is high priority
③ Complex queries on deeply connected data done very fast

(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

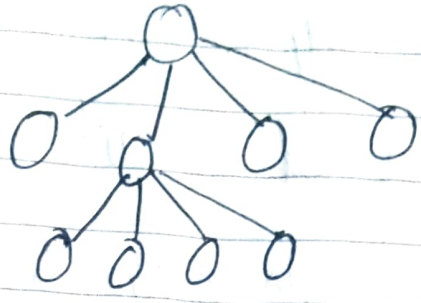
eg.s of a spatial idx :→ Quad tree
→ R tree
→ K-D tree
→ M tree

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

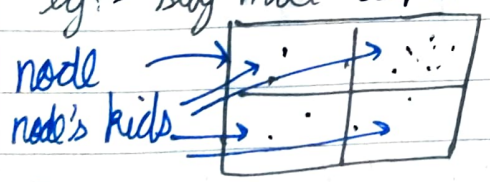
(3) Quad tree nodes → contains some form of spatial data that has a max capa.

(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

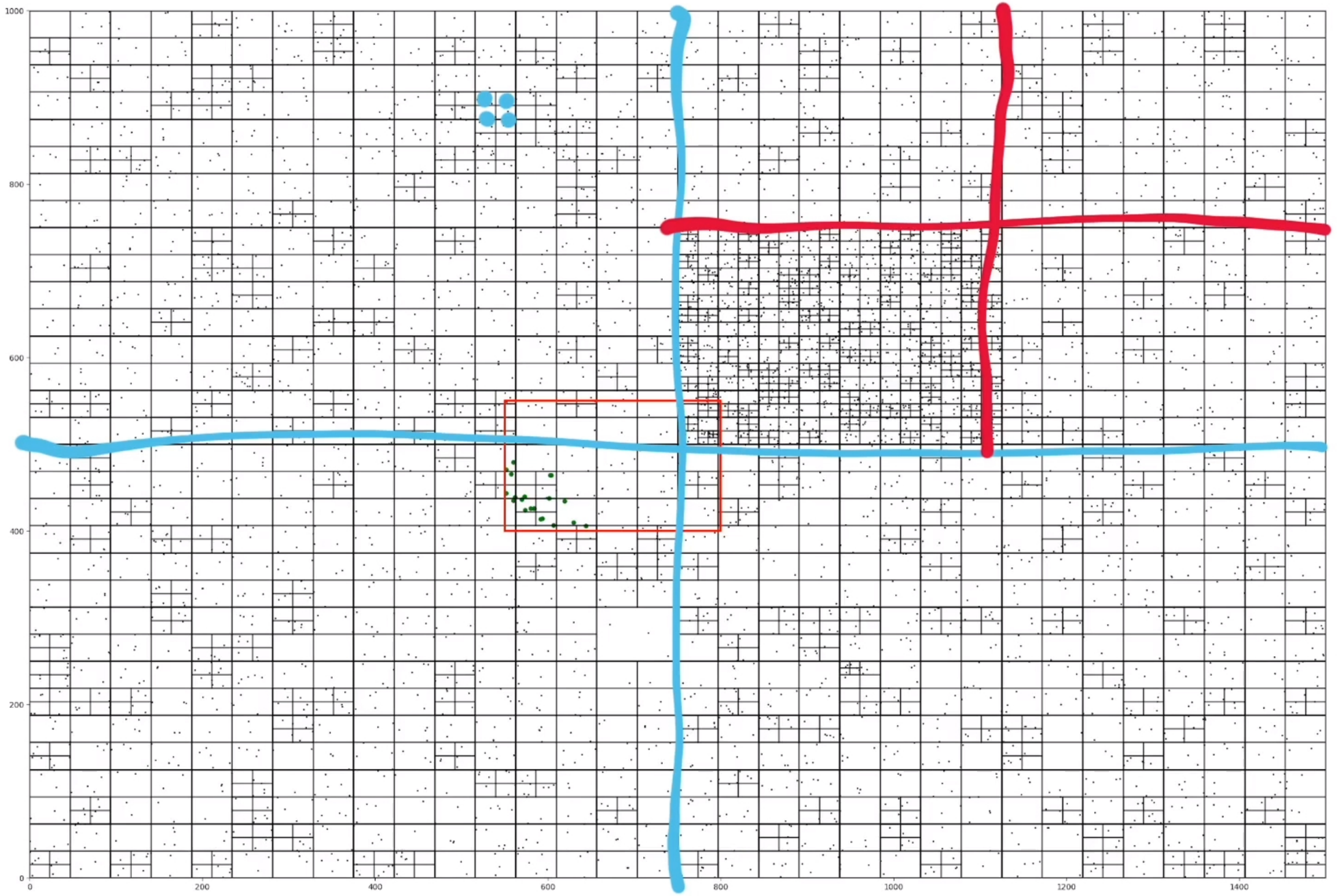
i) Root Node ≡ world ≡ outermost rect.

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

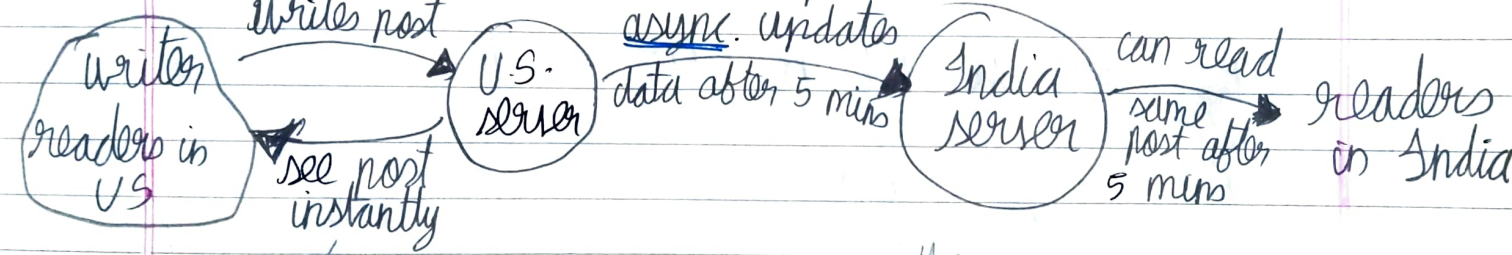
stored but time taken & dist. of server from user.

(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)

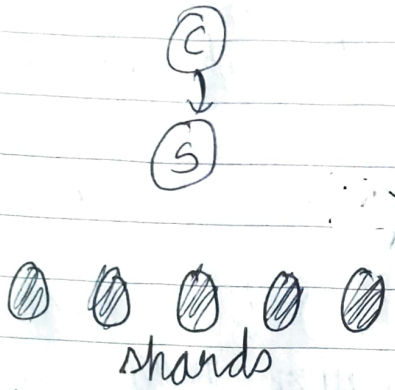
- Addin a new shard → consistent hashin minimizes the pieces of data we gotta migrate from existin shards to new shard
- Shard goes down → consistent hashin can't do anythin
  - soln :- replicate each shard.

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

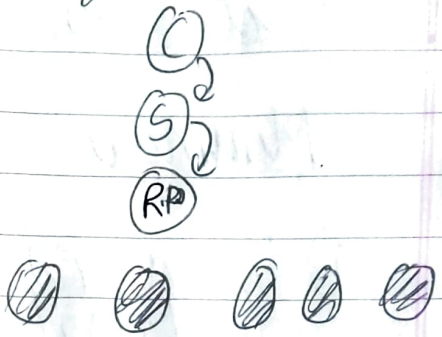
Date:
M T W T F S S

(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)

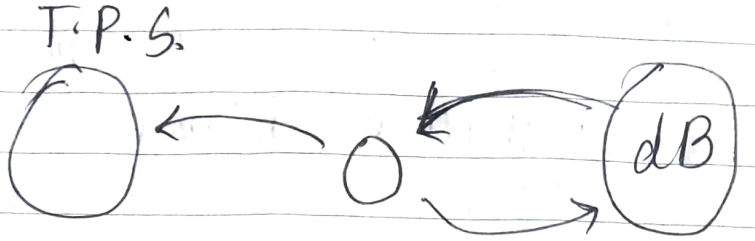
(1) dB → stores data pertainin to subscrip^{n} and users
- → is user curr. ly subscribed?
- → date for subscrip^{n} renewal
- → price

(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.

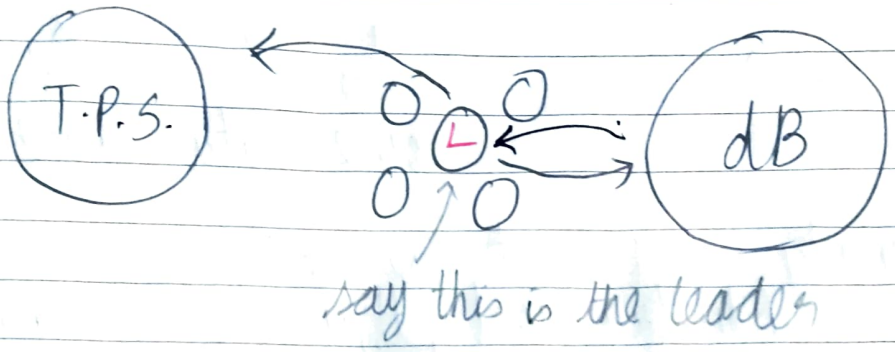
1) If the server in the middle fails, entire payment sys. collapses
We introduce 'redundancy' (passive) in our sys. by hori.ly scaling the service in the middle i.e. have 5 instead 1 server in the middle to do the transac" logic

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.
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

Peer to Peer Networks

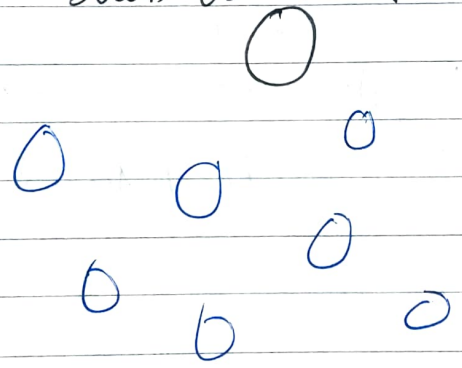
We gonna understand this P2P network concept thru use case

Problem: Gotta a find a soln. to distri abt 5GB of data frm a server (that has thruput = 40Gbps = 5GBps) to a 1000 machines all over the globe

eg:- video footage frm CCTV camera (which you get every 15 mins) and you wanna share it all over.

eg 2:- large ML models that you wanna train on 1000 machines, deployed multip times/day

I] **M-1** → Plain of 1 server → 1000 machines



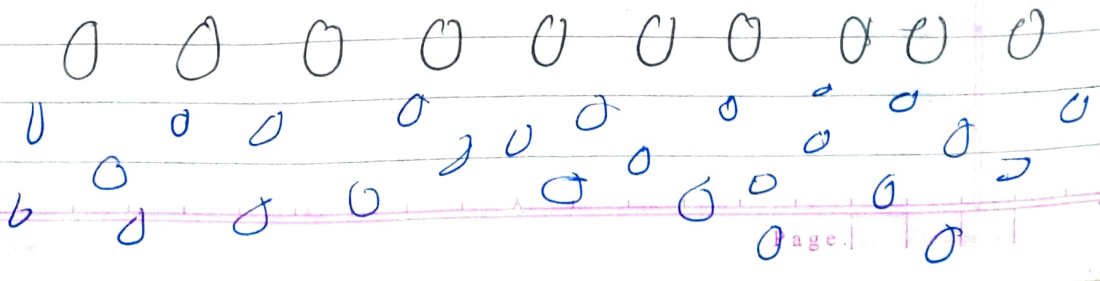
For 1 machine

5 GB file recd in = (file size / thruput) = 1s

∴ After 1st machine, next machine gets data and so on

∴ for 1000th machine.
5GB file recd = after 1000s
= 17 mins ← Pretty slow + clogging at 1 server

II] **M-2** → Hori scalin (Replicaⁿ) of server → 10 servers → 1000 machines



Now, say each server handles equal cnt of machines (say 100 machines)

Say 1 server + 100 machines = 1 set

For 100th machine in any set,
5GB file read offer ≈ 100s = 1.7 mins

still kinda slow

Disadvan. of M-2 → You gotta replicate data from main server to other servers viz pretty inefficient when the file size is real frickin large.

III] M-3 → Sharding : data from 1 server —split→ 10 shards

each of the 1000 machines gotta visit all shards to accumulate all the data

Disadvan. → cloggin at shards

IV] M-4 → P2P networks

each of the 1000 machines = peer

eg:- number each part ↑ UIDable

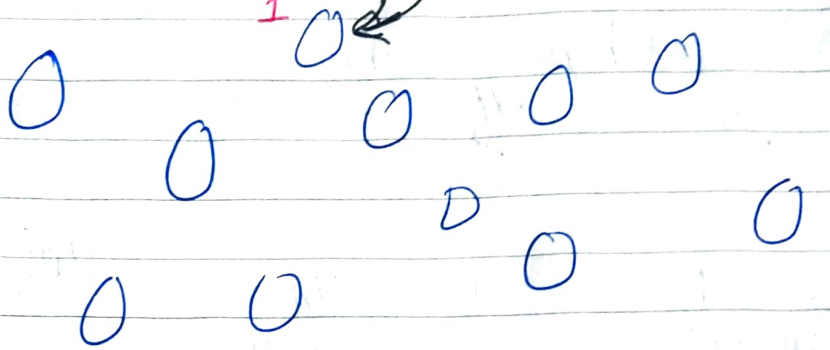
P2P network → 1) Data from main file split into parts and given to diff peers
2) Whilst some peers is gettin data from main server, other peers talk to each other to share & acquire pieces of data that they already don't have to piece togetha entire data.

Date:
M T W T F S S

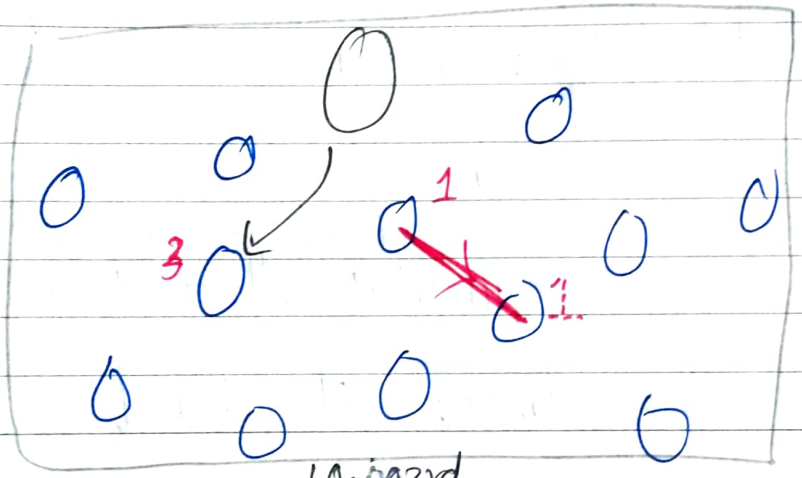
eg: 0.001st sec

each data piece = 5MB

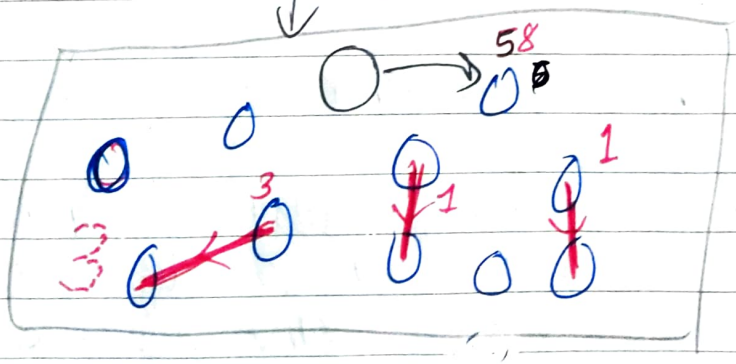
data piece 1, 2, 3, 4 ---- ,100



0.002nd sec

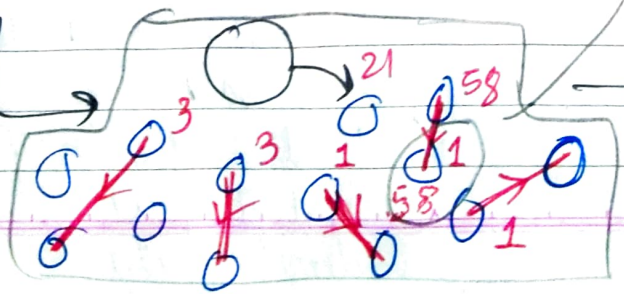


0.003rd sec



now those peer has piece 1, 58 i.e. 2 pieces outta 100

0.004th sec



flows exponentially each sec

For P2P networks to f^n optimally → peers gotta know which peers to talk to next

- to give them data
- to get back data to build up rest of the file that they're missin

Peer discovery / selec^n

Ways by which peers know what peers to comm. with / transfer data to / get data from next.

M-1 : Central DB/machine · a·k·a trackers

Whilst peers are talkin to each other they also comm. with this central machine which tells em which peer to comm. with next

M-2 : Gossip a·k·a. Epidemic Protocol

1) Peers talk b/w themselves and figure out b/w themselves -

eg.- Peer 1 talkin to Peer 2

Peer 2 to Peer 1 : Thanks for piece 1 !
Oooh btw you should totally talk to Peer 47 She got some big chunks you might wanna get a load of

(2) Analogy → people gossipin, shoppin things b/w themselves as well as abt other ppl they're talked to

→ People comin in contact, durin an epidemic, and spreadin virus (here us → a data piece) like wildfire

Implementation

(3) Every peer carries → certain chunks

→ mappings that map certain peers to certain pieces of data

eg:- Peer 2 maps Peer 33 to Peer 99 to help Peer 33 get a piece from Peer 99

(4) After each gossip ↓ knowledge that a peer has w.r.t. "what peer holds what piece of info" ↑ i.e its hash table of IP addresses (→ of other peers) that it needs to go to retrieve missin data pieces

(5) Idea of havin this knowledge viz effectively a mappin viz effectively pieces of a hash table which could be contained all (IP address of peer → piece it has) is a.k.a distributed hash table

D.H.T.

Date:

(6) P2P networks often operate upon the concept of a D.H.T. to figure out which peers hold what pieces of data

eg:- Kraken (developed by Uber) → at its peak, it distributes 2×10^4 100MB - 1GB blobs in under 30s

(7) P2P is fast af as contrary to other methods, here, all machines talk to, share with all machines

eg:- Torrenting

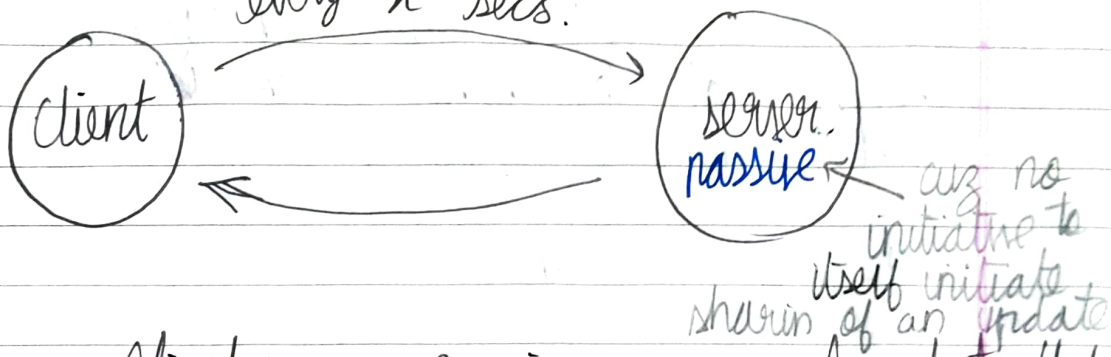
1 machine → 1 large file (eg:- movie) → spread this file in chunks to machines (peers) all over the region / world

then these peers ← work together to obtain all missin pieces i.e- puzzle em back together → each peer gets full file.

Polling & Streaming

Both deal with client getting some live updated data from server (eg:- temp, texts)

POLLING (Client talks (issues req) - Server listens (fulfils req))



Polling : Client is gonna issue a req. for data that it requires on a recurring basis following a set interval.

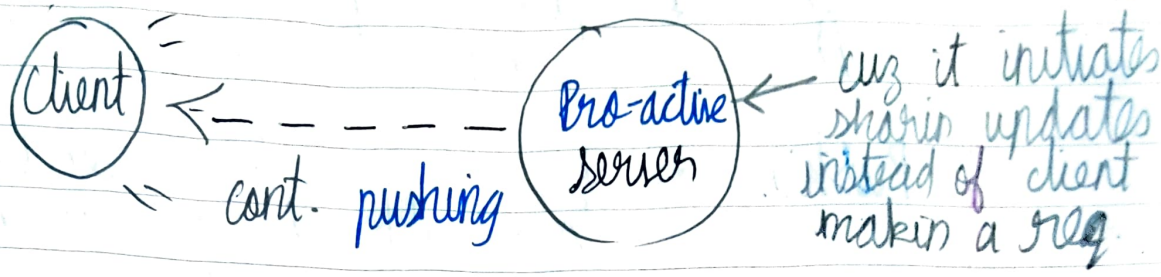
eg:- temp. → enables clients to monitor outside temp. and this temp.'s details are stored in our servers but it changes freq. ly, then the clients poll for temp. every, say, 30 sec, 1 min etc.

Limitations → No instantaneous user experience as updates are visib only after a fixed interval

eg:- can't use pollin for a chat app cuz
- ↳ text updates only after a fixed interval
- ↳ if we tryna ↓ dis fixed interval, to say 0.1 s, we get 10 reqs from each client to server per sec. If we have $> 10^4$ clients, server's clogged

Date:
M T W T F S S

STREAMING (Client listens (for updates) - Server talks (pushes updates))



Instead of issuin reqs, client opens a long lived connec^n with server thru a socket

Socket →
- a file livin on your PC, that PC can write to / read frm to comm. with another PC in a long lived connec^n manner
- portal to another machine that you can reach thru / comm. thru to enable comm.^{n} b/w 2 machines w/o repeatedly send req.
- open connec^n as long as
  - one of the machine closes connec^n
  - network connec^n is healthy.

∴ Here, a client will open a long lived connec^n with the server, thru a socket and listen to the server for any data that server might push to client thru socket.

Client → Client is streamin data frm server. will basically just listen for data (no req.) as long as connec^n is intact

Server is proactive → sends data to client as a push whenever it has an update
∴ logic is implemented on server side.

Pushin → server proactively sendin to data to client ie. it doesn't wait for a req. to do so

Streamin allows sys to hv a cont stream of data so long as server pushes the data updates it receives → instantaneously and correctly

Streamin provides instantaneous experience w/o hvin to issue too many (or any if we're bein real here) to the server.

Pollin	Streamin
Client talks (issues reqs.)	Client listens (collects updates)
Server listens (fulfils reqs.)	Server talks (pushes updates)
Broken stream of data at regular intervals. Req-resp. relat'ship b/w client-server	Cont. stream of data. Open connec" relat'ship b/w client-server.
use: temp. updates, stock updates after each hr (no live-tradin)	use: chat apps, currency exchange updates

LINK TO VIDEO OF CHAT APP POLL-STREAM EXAMPLE

polling_and_streaming — node server.js — 93×24

~/Documents/Content/Design_Fundamentals/Examples/polling_and_streaming — node server.js

```
Clements-MBP:polling_and_streaming clementmihailescu$ node server.js
Listening on port 3001!
```

polling_and_streaming — node client.js — 93×24

~/Documents/Content/Design_Fundamentals/Examples/polling_and_streaming — node client.js

```
Clements-MBP:polling_and_streaming clementmihailescu$ (for i in `seq 1 10000`; do sleep 1; echo $i; done) | NAME=Bot node client.js
```

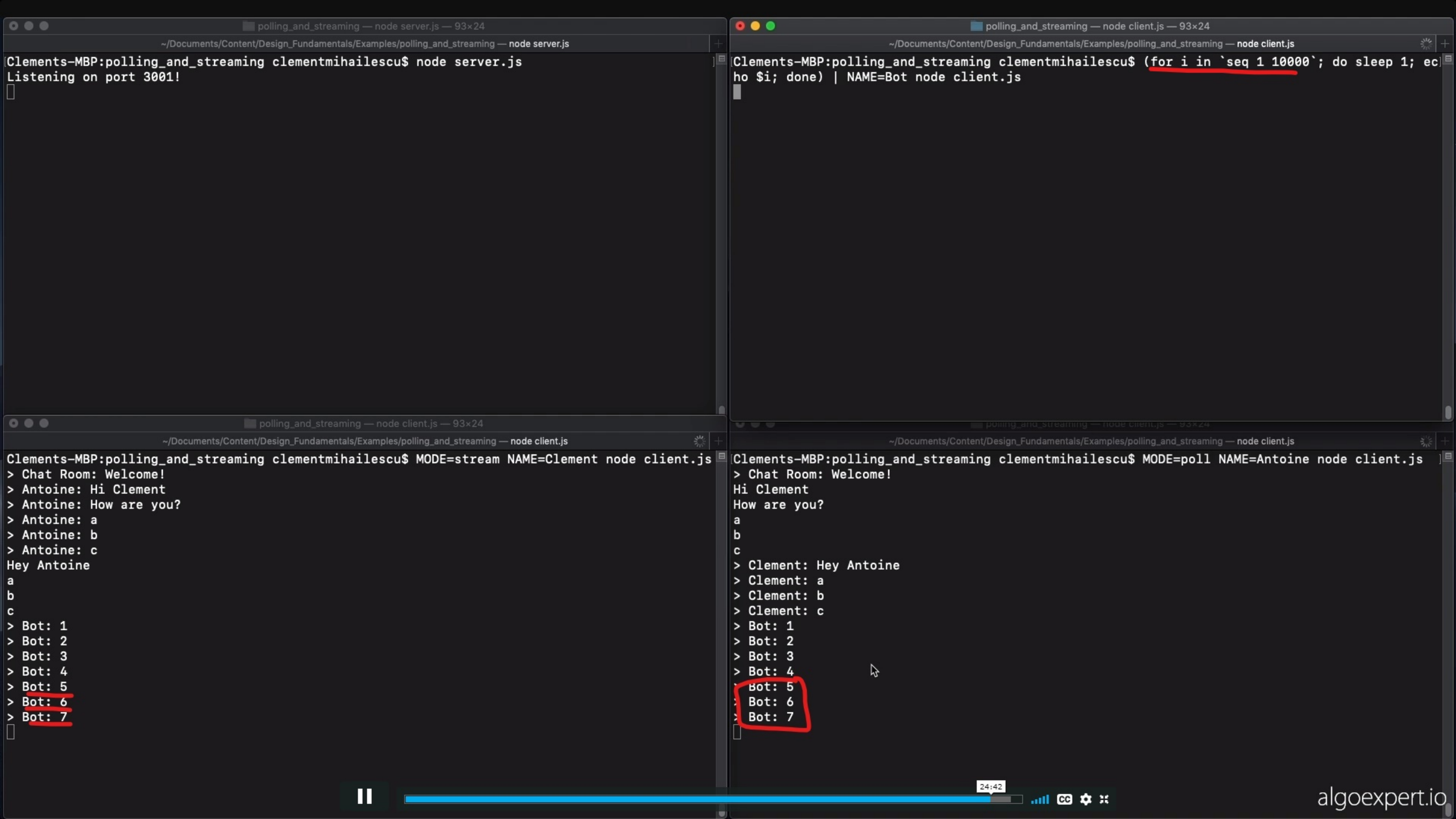
polling_and_streaming — node client.js — 93×24

~/Documents/Content/Design_Fundamentals/Examples/polling_and_streaming — node client.js

```
Clements-MBP:polling_and_streaming clementmihailescu$ MODE=stream NAME=Clement node client.js
> Chat Room: Welcome!
> Antoine: Hi Clement
> Antoine: How are you?
> Antoine: a
> Antoine: b
> Antoine: c
Hey Antoine
a
b
c
> Bot: 1
> Bot: 2
> Bot: 3
> Bot: 4
> Bot: 5
> Bot: 6
```

~/Documents/Content/Design_Fundamentals/Examples/polling_and_streaming — node client.js

```
Clements-MBP:polling_and_streaming clementmihailescu$ MODE=poll NAME=Antoine node client.js
> Chat Room: Welcome!
Hi Clement
How are you?
a
b
c
> Clement: Hey Antoine
> Clement: a
> Clement: b
> Clement: c
> Bot: 1
> Bot: 2
> Bot: 3
> Bot: 4
```


polling_and_streaming — node server.js — 93×24
~/Documents/Content/Design_Fundamentals/Examples/polling_and_streaming — node server.js
Clements-MBP:polling_and_streaming clementmihailescu$ node server.js
Listening on port 3001!
polling_and_streaming — node client.js — 93×24
~/Documents/Content/Design_Fundamentals/Examples/polling_and_streaming — node client.js
Clements-MBP:polling_and_streaming clementmihailescu$ (for i in `seq 1 10000`; do sleep 1; echo $i; done) | NAME=Bot node client.js
polling_and_streaming — node client.js — 93×24
~/Documents/Content/Design_Fundamentals/Examples/polling_and_streaming — node client.js
Clements-MBP:polling_and_streaming clementmihailescu$ MODE=stream NAME=Clement node client.js
> Chat Room: Welcome!
> Antoine: Hi Clement
> Antoine: How are you?
> Antoine: a
> Antoine: b
> Antoine: c
Hey Antoine
a
b
c
> Bot: 1
> Bot: 2
> Bot: 3
> Bot: 4
> Bot: 5
> Bot: 6
> Bot: 7
~/Documents/Content/Design_Fundamentals/Examples/polling_and_streaming — node client.js
Clements-MBP:polling_and_streaming clementmihailescu$ MODE=poll NAME=Antoine node client.js
> Chat Room: Welcome!
Hi Clement
How are you?
a
b
c
> Clement: Hey Antoine
> Clement: a
> Clement: b
> Clement: c
> Bot: 1
> Bot: 2
> Bot: 3
> Bot: 4
> Bot: 5
> Bot: 6
> Bot: 7
24:42
algoexpert.io

Logging & Monitoring

2 imp concepts that help debug issues, find improvement areas easily as your sys. keeps growin

LOGGING:

The act of collectin and storin logs (useful info abt. events in your sys).

Typically, prog.s output log msgs to its STDOUT (regular logs) or STDERR (for error logs) pipes.
eg:- console.log for JS, cout for C++, print() for Python.

The outputs recd at the end of these pipes automatically get aggregated into a 'centralized logging solⁿ' → like a DB we store all logs in for visibility into sys's f"ing and debug issues
eg:- Google Stackdriver.

Use case :- someone purchasin a product gets charged but doesn't gain prod. access
Logs related to that user's opn./behaviour, shows you what exactly happened durin this transacⁿ opn.

Gotta format these logs usin a special lib to ensure logs r hr useful info and are easy to read

2 popular opⁿ's → Syslog, JSON

Date:
M T W T F S S

MONITORING :

The process of givin visibility into a sys's key metrics. Creatin sys-s to monitor imp metrics on overall sys
Usually implemented by collectin imp events in a sys, and aggregatin em in human-readable statistic representa"s (eg:- charts)

Makes managin a sys, lookin at its performance stats, stats related to which part of sys is most imp to users (eg:- Google Sign In > sign in by email), stats related to gen health, status of sys. ⟶ a.k.a. gathering metrics

As sys grows, we wanna hv more and more visibility into a lotta opns. in it

eg: of metrics that are useful →
- e-commerce related (how many sales per week?)
- health, performance, gen. status of sys
- stats on what features are used most
- login (eg:- logins/day), authentica" stats (eg:- Google auth vs Github auth)

→ M-1 : Build / use a tool that scrapes logs and extracts data to create metrics

Problems → ltd. by data present in logs.
↳ if you ever decide to change your logs, you risk breakin the monitorin sys.

M-2 → Using a time series dB (TSDB) i.e. a specialised dB specifically tailored for data measured of time
eg of TSDB → InfluxDB, Prometheus, Graphite

Servers periodically send data to TSDB

Querying this dB (use query lang) for data

Use tools to graph values stored in dB
(eg of tool: Grafana)
↳ creates graphs out TSDBs

Can build robust monitoring sys.s to reduce dependency on just logs

ALERTING:

The process thru which sys. admins get notified when critical sys. issues occur.

Alerting can be set up by defining specific thresholds on monitoring charts, past which alerts are sent to a comm" channel (eg of channel → Slack)

eg:- alerts on abnormal ↑ in latency of sys.

Logging And Monitoring

CREATE METRIC | CREATE SINK | SAVE SEARCH | SHOW LIBRARY

Filter by label or text search

GKE Container, main-cluster, algoexpert | All logs | Any log level | Last hour | Jump to now

Showing logs from the last hour ending at 5:36 PM (PST) — Download logs | View Options

```
2020-01-13 17:35:53.473 PST {"service_name":"auth","response_code":200,"ts":1578965753.4734147,"elapsed_time_ms":0.165404…
2020-01-13 17:35:53.473 PST {"path":"/api/auth/describe","ts":1578965753.4737482,"level":"info","elapsed_time_ms":1.09223…
2020-01-13 17:35:53.481 PST {"msg":"Running code in csharp for user                    ,"ts":1578965753…
2020-01-13 17:35:53.481 PST {"msg":"Prepping user code on worker 0","ts":1578965753.4810894,"level":"info"}
2020-01-13 17:35:53.488 PST {"ts":1578965753.488276,"level":"info","msg":"running code"}
2020-01-13 17:35:53.526 PST {"msg":"Request completed in 153.196916ms (retry: 0)","ts":1578965753.526525,"level":"info"}
2020-01-13 17:35:53.526 PST {"ts":1578965753.5267758,"request_id":"86232398","elapsed_time_ms":875.431942,"request_path":…
2020-01-13 17:35:53.532 PST level=warn ts=2020-01-14T01:35:53.532Z caller=scrape.go:930 component="scrape manager" scrape…
2020-01-13 17:35:53.577 PST {"request_path":"/api/users/v1/meta","msg":"Request Served","level":"info","service_name":"us…
2020-01-13 17:35:53.582 PST {"request_path":"/api/users/v1/screen_lease/acquire","msg":"Request Served","level":"info","s…
2020-01-13 17:35:53.641 PST {"service_name":"auth","response_code":200,"ts":1578965753.6414318,"elapsed_time_ms":0.170159…
2020-01-13 17:35:53.642 PST {"msg":"Internal Request","path":"/api/auth/describe","ts":1578965753.641831,"level":"info","…
2020-01-13 17:35:53.667 PST {"ts":1578965753.6672525,"elapsed_time_ms":0.838036,"request_id":"36655995","request_path":"/…
2020-01-13 17:35:53.722 PST {"level":"info","service_name":"users","response_code":200,"ts":1578965753.7219486,"elapsed_t…
2020-01-13 17:35:53.728 PST {"service_name":"auth","response_code":200,"ts":1578965753.7285347,"request_id":"86939580","e…
2020-01-13 17:35:53.729 PST {"msg":"Internal Request","path":"/api/auth/describe","ts":1578965753.7290704,"level":"info",…
2020-01-13 17:35:53.734 PST {"request_path":"/api/events/v1/event","msg":"Request Served","level":"info","service_name":"…
2020-01-13 17:35:53.741 PST {"ts":1578965753.7413568,"elapsed_time_ms":100.670154,"request_id":"2719906","request_path":"…
2020-01-13 17:35:53.746 PST {"request_path":"/api/auth/describe","msg":"Request Served","level":"info","service_name":"au…
2020-01-13 17:35:53.747 PST {"path":"/api/auth/describe","ts":1578965753.7470586,"level":"info","elapsed_time_ms":1.13663…
2020-01-13 17:35:53.748 PST {"ts":1578965753.7483938,"level":"info","msg":"Running code in python for user              "
2020-01-13 17:35:53.748 PST {"ts":1578965753.7484279,"level":"info","msg":"Prepping user code on worker 1"}
2020-01-13 17:35:53.749 PST {"ts":1578965753.748842,"request_id":"46706538","elapsed_time_ms":0.158472,"request_path":"/a…
2020-01-13 17:35:53.749 PST {"path":"/api/auth/describe","ts":1578965753.7492566,"level":"info","elapsed_time_ms":1.03406…
2020-01-13 17:35:53.754 PST {"ts":1578965753.7547185,"level":"info","msg":"running code"}
2020-01-13 17:35:53.755 PST {"request_path":"/api/auth/describe","msg":"Request Served","level":"info","service_name":"au…
2020-01-13 17:35:53.756 PST {"msg":"Internal Request","path":"/api/auth/describe","ts":1578965753.7563138,"level":"info",…
2020-01-13 17:35:53.756 PST {"ts":1578965753.7565453,"level":"info","msg":"Running code in go for user testing|caee8e08-7…
2020-01-13 17:35:53.756 PST {"msg":"Prepping user code on worker 0","ts":1578965753.7565784,"level":"info"}
2020-01-13 17:35:53.763 PST {"ts":1578965753.7634645,"level":"info","msg":"running code"}
2020-01-13 17:35:53.885 PST level=warn ts=2020-01-14T01:35:53.884Z caller=scrape.go:930 component="scrape manager" scrape…
2020-01-13 17:35:53.902 PST level=warn ts=2020-01-14T01:35:53.902Z caller=scrape.go:930 component="scrape manager" scrape…
2020-01-13 17:35:53.951 PST {"request_path":"/api/users/v1/screen_lease/acquire","msg":"Request Served","level":"info","s…
2020-01-13 17:35:53.990 PST {"ts":1578965753.9900293,"request_id":"34391740","elapsed_time_ms":0.51834,"request_path":"/a…
2020-01-13 17:35:53.992 PST {"request_path":"/api/users/v1/screen_lease/acquire","msg":"Request Served","level":"info","s…
2020-01-13 17:35:54.021 PST level=warn ts=2020-01-14T01:35:54.020Z caller=scrape.go:930 component="scrape manager" scrape…
2020-01-13 17:35:54.030 PST {"ts":1578965754.0304697,"level":"error","msg":"run failed: exit status 1"}
2020-01-13 17:35:54.030 PST {"msg":"Done running user code on worker 1 at /shared/worker-1: [timeout 3 do-test /mnt/progr…
2020-01-13 17:35:54.031 PST {"msg":"worker ready check: true","ts":1578965754.031149,"level":"info"}
2020-01-13 17:35:54.031 PST {"service_name":"rce-python","response_code":200,"ts":1578965754.031206,"request_id":"6086324…
2020-01-13 17:35:54.031 PST {"msg":"worker 1 ready","ts":1578965754.0314157,"level":"info"}
2020-01-13 17:35:54.071 PST {"request_path":"/api/users/v1/screen_lease/acquire","msg":"Request Served","level":"info","s…
2020-01-13 17:35:54.085 PST level=warn ts=2020-01-14T01:35:54.085Z caller=scrape.go:930 component="scrape manager" scrape…
2020-01-13 17:35:54.286 PST level=warn ts=2020-01-14T01:35:54.286Z caller=scrape.go:930 component="scrape manager" scrape…
```

Date:
M T W T F S S

Publish/Subscribe Pattern

a.k.a. Pub/Sub model a.k.a. pub-sub

Recap : streaming → client-server long-lived connec° where client listens for data.

eg :- 1) chat app
2) like trading of stocks by stockbrokers - they gotta know latest updated stock prices as they gonna be bettin huge money on it.

Say now we wanna expand this sys into a large scale distri. sys.

Issues → handlin network partiⁿs (eg:- clients lose connecⁿ to servers)
→ servers die

In above cases, what happens to the data (eg:- txt msgs) when an issue is encountered? Is the data lost? Will clients be able to retrieve data upon reconnecⁿ?

Soluⁿ → The moment we think of expandin any sys into large scale distri sys, we gotta start thinkin abt persistant storage.

M-1 : Store data in a typical DB.

Problem with M-1 → Can't be used in all cases.
eg → Client sends async opn to server, opn takes

(C) (S) (dB)

some time in server to process and then resp is sent back to client. We don't use a typical dB for this as we gotta wait in req-resp cycle

M-2 → Storage soln. at server level. i.e. Server ≡ storage
eg:- Streamin

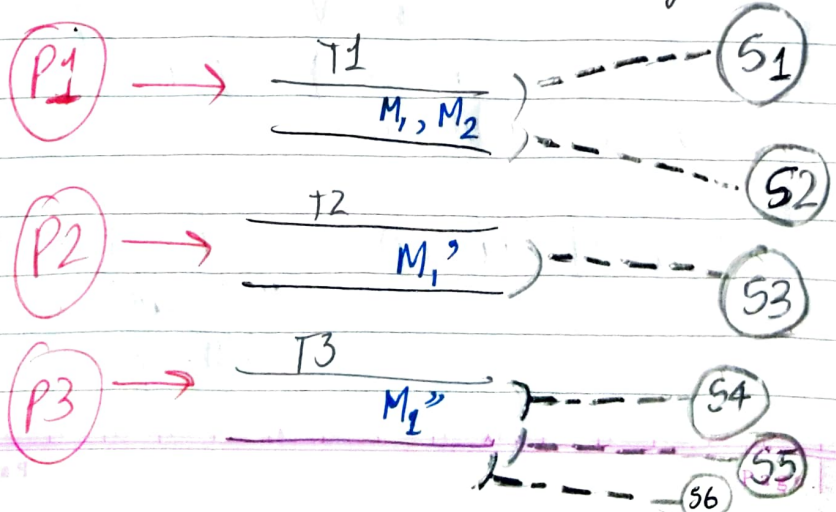
(C) (S≡dB)

Problem with M-2 : We don't wanna keep our business logic (usually in server) and our storage (usually in dB) ~~at the~~ separated

Pub-Sub System

1 A paradigm of 4 entities :

1) Publisher → servers that publish data to topic
2) Subscriber → Clients ~~as~~ listens to data from topics
3) Topic → Channels of specific info. Intermediary b/w publishers and subscribers (who don't directly comm. with each other
4) Msg → Represent → data / opn. of some form i.e. relevant to subscribers eg:- a chat txt, stock trade opn.



(2) Persistent storage in pub-sub pattern

Topics ≡ a dB soln.

∴ All msgs published to topic → effectively stored in a dB soln ≡ persistent

∴ Guarantee :- Msgs in a topic will delivered atleast once to subscribers

Overview logic - (1) Each msg keeps track of subscribers thru some sort of a UID.
(2) Upon receipt of msgs., subscribers send a 'ack' (acknowledgement) to the topic

(3) Idempotent opn.

Problem → (1) Msg. recd by subscriber but suddenly connec" b/w subscriber & topic fails
(2) Subscriber didn't send ack to topic
(3) Upon reconnec", topic thinks sub. doesn't hv msg so its gonna resend it.

Idempotent opn. → Opn. giving same ultimate outcome regardless of how many times it's performed

eg:- Click on 'Register' btn. to register for a webinar

eg of non-idempotent opn. → Like a facebook post (2nd time you press 'Like' btn, post is unliked)

Pub-sub sys has ~~drawbacks~~ drawbacks if the opn is non-idempotent cuz on reconnect of client the opn kinda does a diff thing than it did before.

(4) Ordering of msgs.

Order in which msgs published to topic from publisher = Order in which msgs pushed to sub. frm. topic

Queue (FIFO → 1st in 1st out)

use case: chat msgs appear chronologically, transac" opns. performed sequentially

(5) Replay msgs

Rewinding to a prev. msg or a prev. snapshot in time of the topic

eg:- Git revert

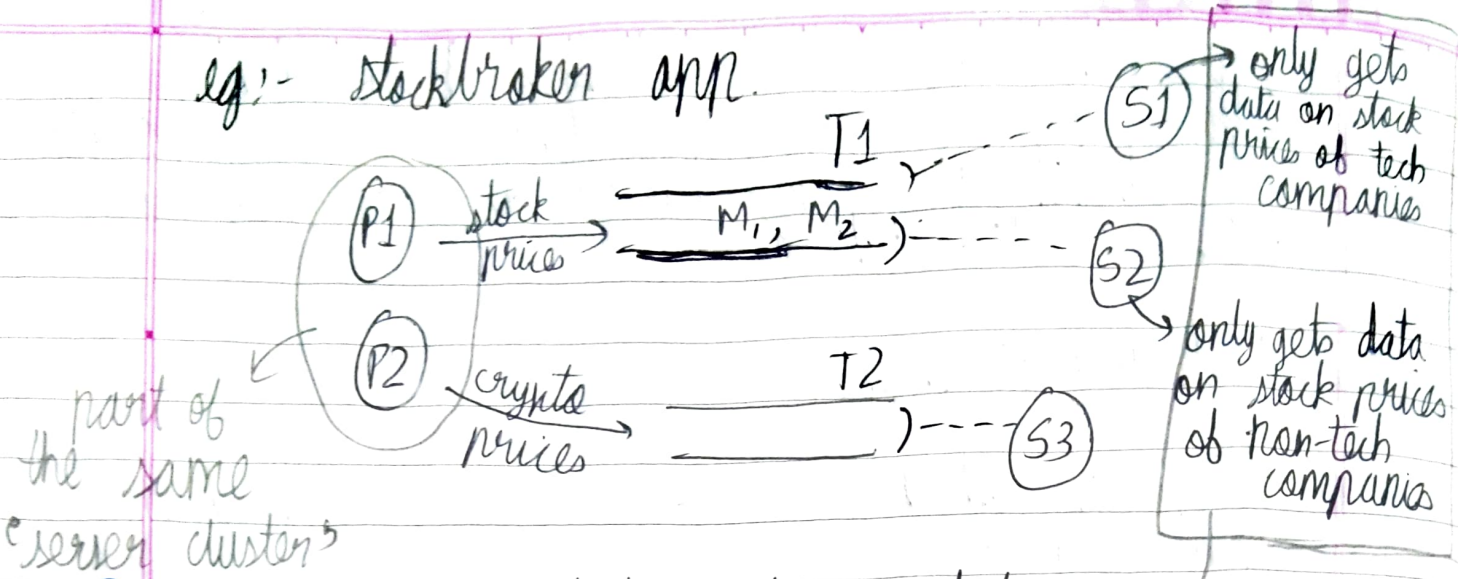
(6) Pub-sub Topics → Channels :: they channel specific data/opns. frm pub to sub.

All topics stored in ~~db~~ a common single dB
Each topic represents a diff type of data

This dB → bi-dir"al (pub. on 1 side, sub. on other)
→ clear separat" of data/opns. in topics

Date:
M T W T F S S

eg:- stockbroker app.



(7) Content based filtering at sub. lvl

Subs who have subscribed to the same topic but get pretty specific data amongst the data stored in the topic.

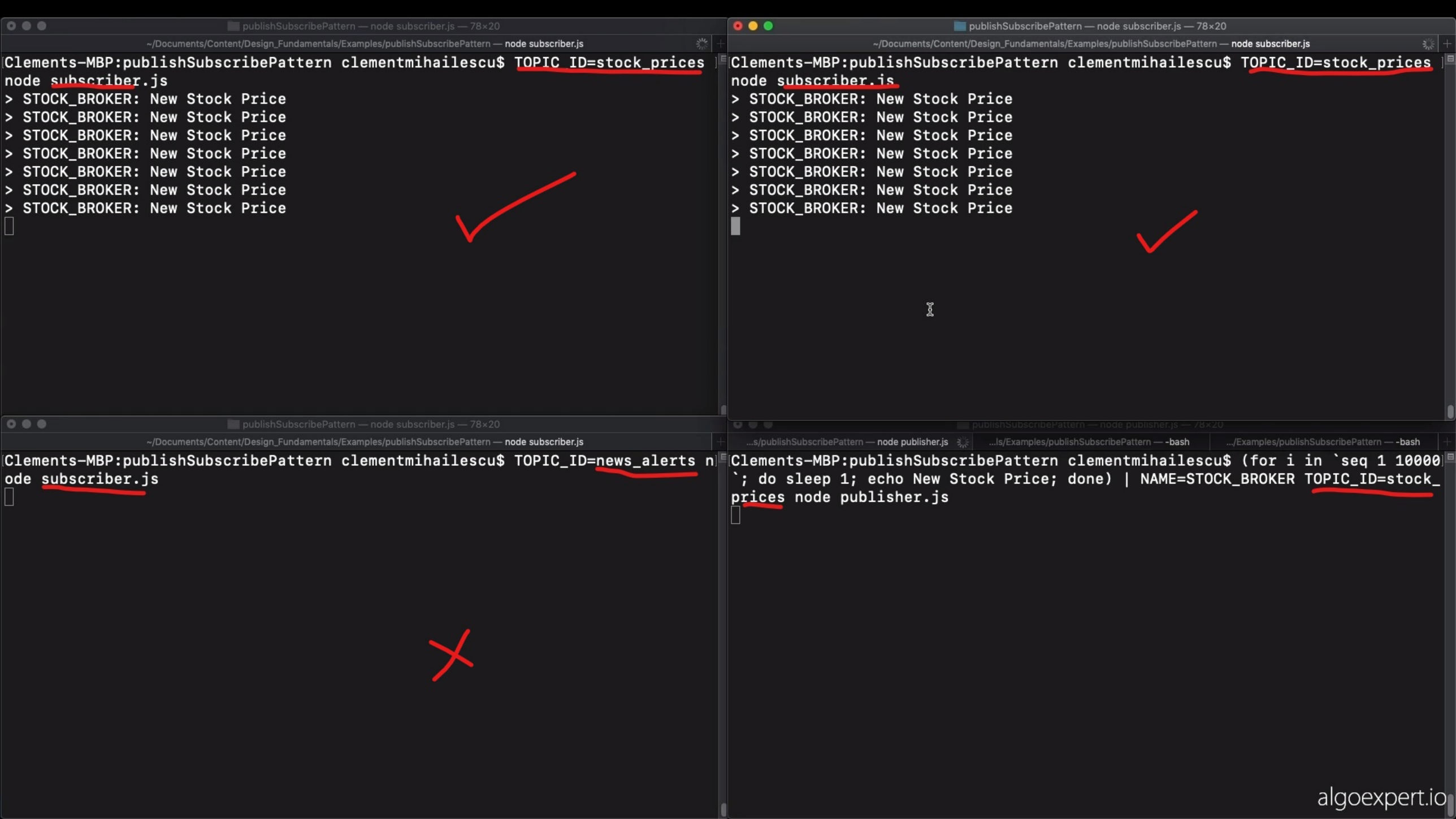
(8) Examples of pub-sub tools/solns

1) Google Cloud Pub-Sub - Everything auto-scales and you don't need to worry cuz this soln. will automatically take care of it by sharding topics

2) Google Cloud Platform, AWS - Provide end-to-end encryp^{n} of msgs s.t. msgs are encrypted over the network whilst inside topics and only subs know how to read em.
eg:- whatsapp msg encryp^{n}

3) Apache Kafka.

LINK TO CODE EXAMPLE OF PUB/SUB PATTERN


publishSubscribePattern — node subscriber.js — 78×20
~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js
Clements-MBP:publishSubscribePattern clementmihailescu$ TOPIC_ID=stock_prices node subscriber.js
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
publishSubscribePattern — node subscriber.js — 78×20
~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js
Clements-MBP:publishSubscribePattern clementmihailescu$ TOPIC_ID=stock_prices node subscriber.js
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
publishSubscribePattern — node subscriber.js — 78×20
~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js
Clements-MBP:publishSubscribePattern clementmihailescu$ TOPIC_ID=news_alerts node subscriber.js
publishSubscribePattern — node publisher.js — 78×20
...s/publishSubscribePattern — node publisher.js
...ls/Examples/publishSubscribePattern — -bash
.../Examples/publishSubscribePattern — -bash
Clements-MBP:publishSubscribePattern clementmihailescu$ (for i in `seq 1 10000`; do sleep 1; echo New Stock Price; done) | NAME=STOCK_BROKER TOPIC_ID=stock_prices node publisher.js
algoexpert.io

publishSubscribePattern — node subscriber.js — 78×20

~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js

```
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
```

publishSubscribePattern — node subscriber.js — 78×20

~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js

```
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
^C
Clements-MBP:publishSubscribePattern clementmihailescu$ TOPIC_ID=stock_prices
node subscriber.js
> STOCK_BROKER: New Stock Price
> STOCK_BROKER: New Stock Price
```

publishSubscribePattern — node subscriber.js — 78×20

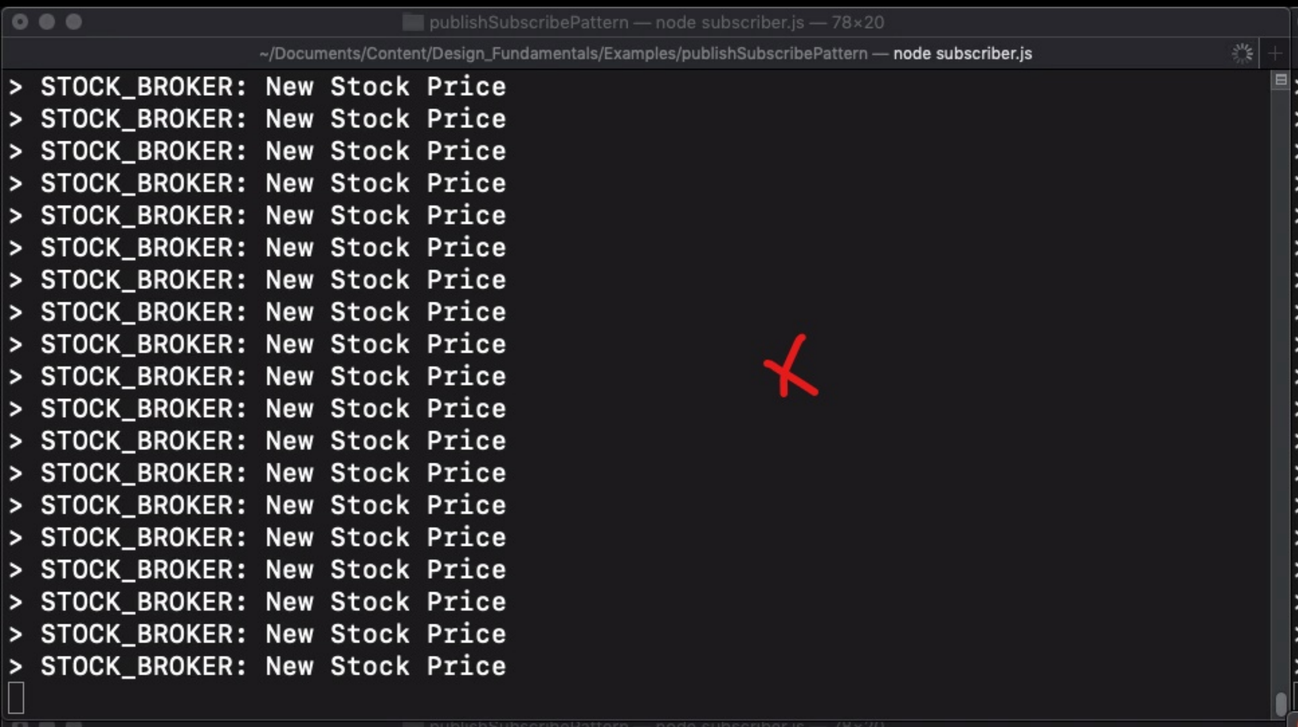
~/Documents/Content/Design_Fundamentals/Examples/publishSubscribePattern — node subscriber.js

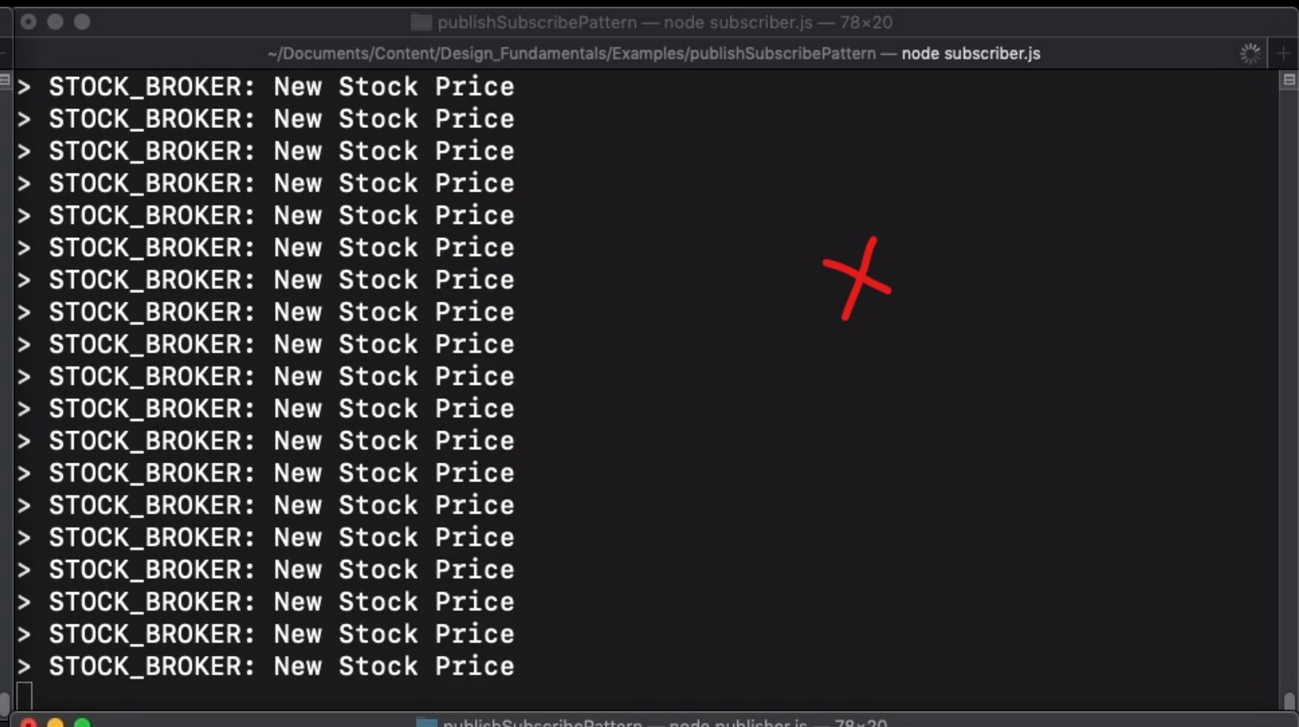
```
Clements-MBP:publishSubscribePattern clementmihailescu$ TOPIC_ID=news_alerts n
ode subscriber.js
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
> NEWS_STATION: Breaking News
```

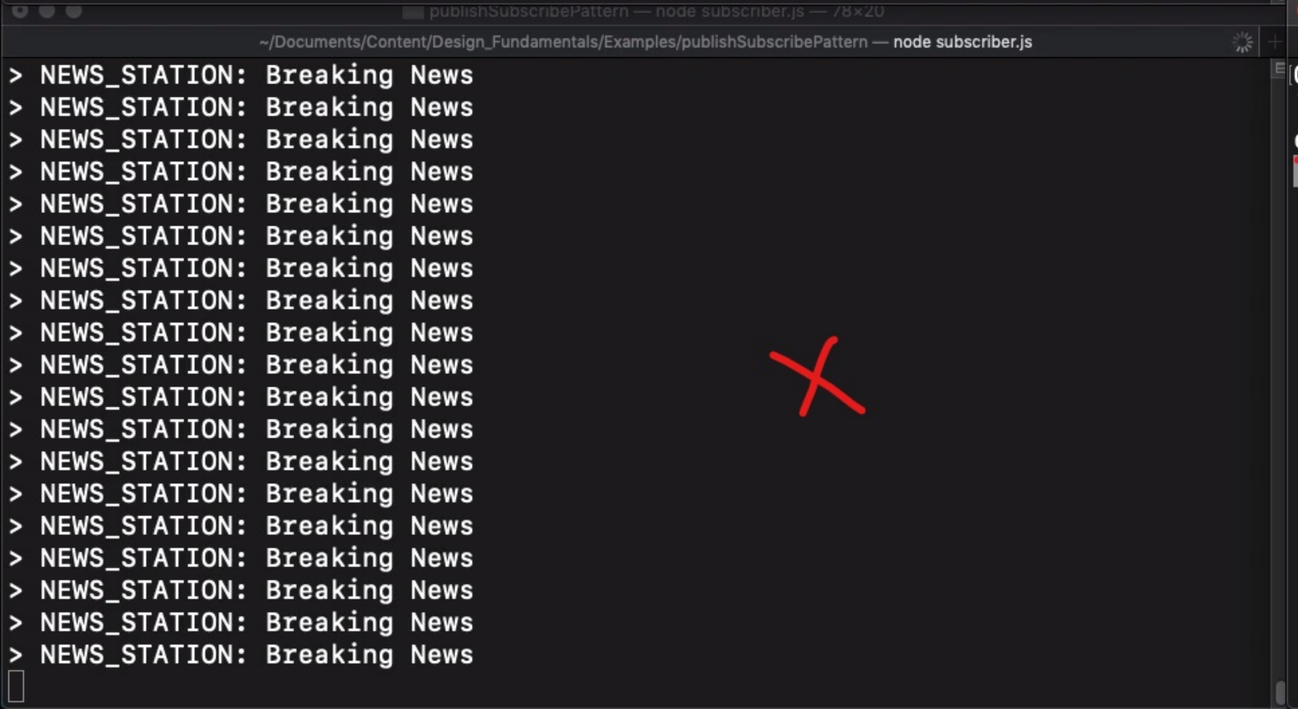
publishSubscribePattern — node publisher.js — 78×20

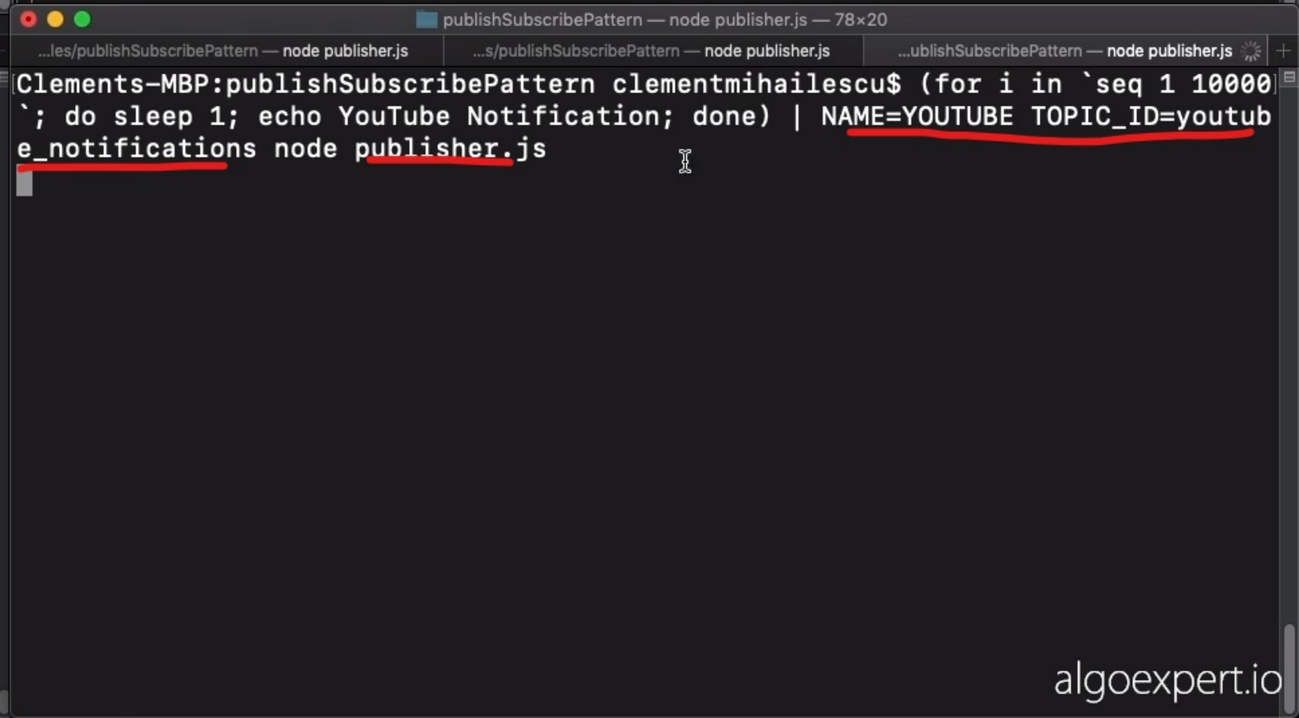
...les/publishSubscribePattern — node publisher.js | ...s/publishSubscribePattern — node publisher.js | .../Examples/publishSubscribePattern — -bash

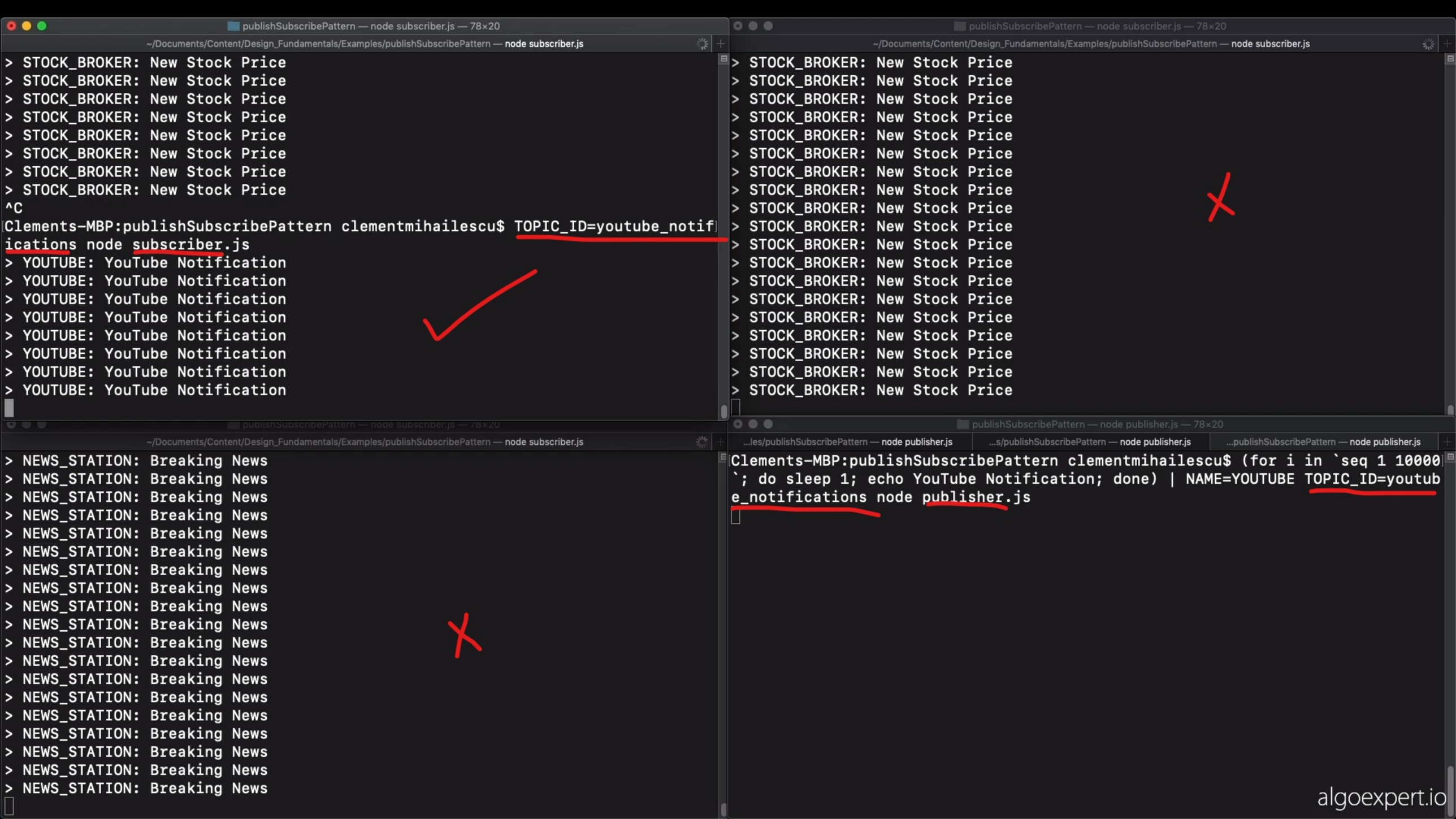
```
Clements-MBP:publishSubscribePattern clementmihailescu$ (for i in `seq 1 10000
`; do sleep 1; echo Breaking News; done) | NAME=NEWS_STATION TOPIC_ID=news_ale
rts node publisher.js
```











mapReduce

Problem: Processin large data sets stored across crapload (1000s) of machines → large scale distri. sys.
Processin such a data set stored across so many machines is difficult cuz you gotta:

- → paralyze the processin across these many machines
- → handle failures (eg:- network parti's, machine failures)

mapReduce

A framework/model (library) that allows sys admins to process v. large datasets spread across 100s / 1000s of machines. (distri. ed setting) in an efficient and fault-tolerant manner

Majority of data processin tasks split up into

- map step → transforms data across diff machines into k-v pairs
- reduce step → k-v pairs reduced into a final output eg:- a file to be used in a sys.

Date:
M T W T F S S

Key things abt the mapReduce model

1) When dealin with this model, we assume that we have a distributed file sys. i.e. data set split into chunks that are replicated and spread out across multip. machines (say 1000s of machines)

data set —split into→ chunks —replicated, spread→ across 1000s of machines

a central control plane is gonna be aware of everythin goin on in the mapReduce process

- → where each chunk resides
- → how to comm. with the various storage machines
- → comm.s with machines that perform map opns. a.k.a. worker machines.
- → comms. with reduce workers
- → where output resides

DISTRIBUTED FILE SYS

2)

2] Since dataset in each machine is too large to move, we send the map prog. to each such dataset.

3] k/v pairs struc. is imp. → when you 'reduce' data values that come from multip chunks of the same dataset, → you try to shuffle the k/v pairs struc. s.t. you get k-v pairs sharing common data, outputting to the same. → i.e. reduce the common data into a single meaningful value.

4] Handling faults

i) Model basically reperforms the 'map' or 'reduce' opn where the failure occured

ii) Hence, it is very imp. that the map and reduce opn. you choose for your sys., are idempotent

5] Sys. admin just needs to take care of 6 things

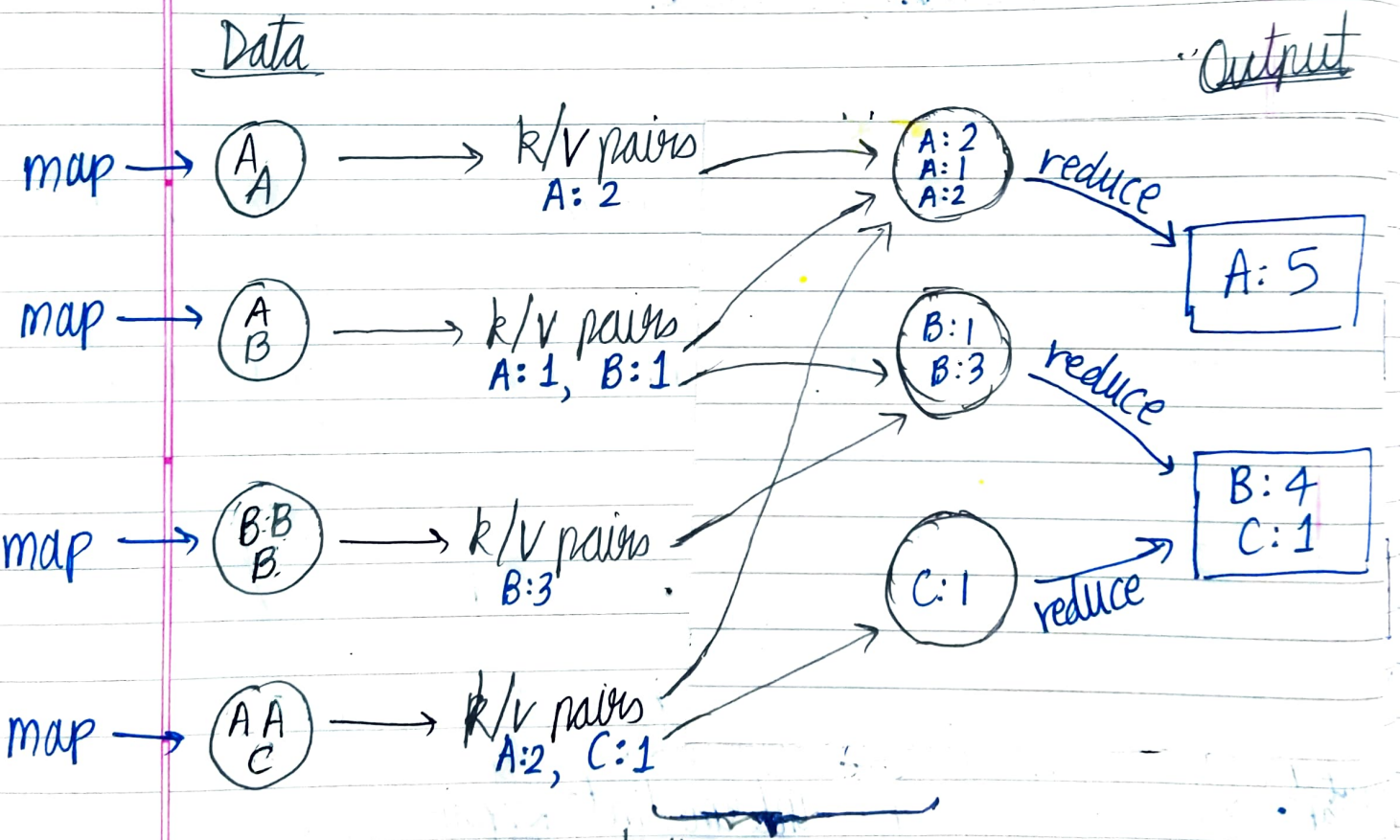
input of map → i) What the input data to map is?
ii) What f^n is specified
output of map → iii) What the intermediate k-v pairs (output of 'map')
input of reduce → iv) How these k-v pairs are shuffled (to optimize 'reduce')
v) What reduce f^n is is specified
output of reduce → vi) What the output of 'reduce' f^n (final output)

∴ No need to worry abt the intricacies of processin data in a distri-ed sys. cuz there's

gonna be libraries based on this model that you can use just by specifyin

- map fn
- reduce fn
- lookin at various I/O of both map and reduce

eg:- Countin no. of occurances of a letter in a large text file → aggrega^n of wikipedia pages, online eBooks



Date:
M T W T F S S

NOTE :- The kinda k-v pairs you generate depends on the type of map fn you choose
eg:- You could choose a map fn that outputs a stream of k-v pairs

Instead of outputtin A:2, we wouldve gotten A:1, A:1

In our eg:

Map fn → Some sorta loop that'll iterate thru txt file at each dataset and tally up the no. of occurances of a particular letter
All map opns run //ly

Reduce fn → Counts up all occurances in the input to it and writes final values to output file
All reduce ops also run //ly.

Shuffle step → Grouped all keys pertaining to a particular letter in 1 place to optimize reduce.

eg of use cases - ① A bunch of YouTube vids stored in a dataset i.e you have metadata abt this vids and you wanna get total no. of views or likes per vid

② Bunch of logs from various services (eg:- payments, authentication) and you wanna count total no. of logs in a given time interval

Date:
M T W T F S S

mapReduce →

→ Step 1 - Map
Runs a 'map' fn on the various chunks of the dataset and transforms these chunks into intermediate k-v pairs

→ Step 2 - Shuffle
Recognizes intermediate k-v pairs s.t. the pairs of the same key are routed to same machine in final step

→ Final step - Reduce
Runs a 'reduce' fn on the newly shuffled k-v pairs and transforms em into more meaningful data

Distributed file sys. (DFS)

DFS is an abstrac" over a (usually large) cluster of machines that allows em to act like 1 large file sys.

2 popular implementa" →
- Google File Sys (GFS)
- Hadoop DFS (HDFS)

Purpose : Extremely large scale persistent storage.

Diff. APIs implem"s hv diff APIs, semantics but purpose is same

Date:
M T W T F S S

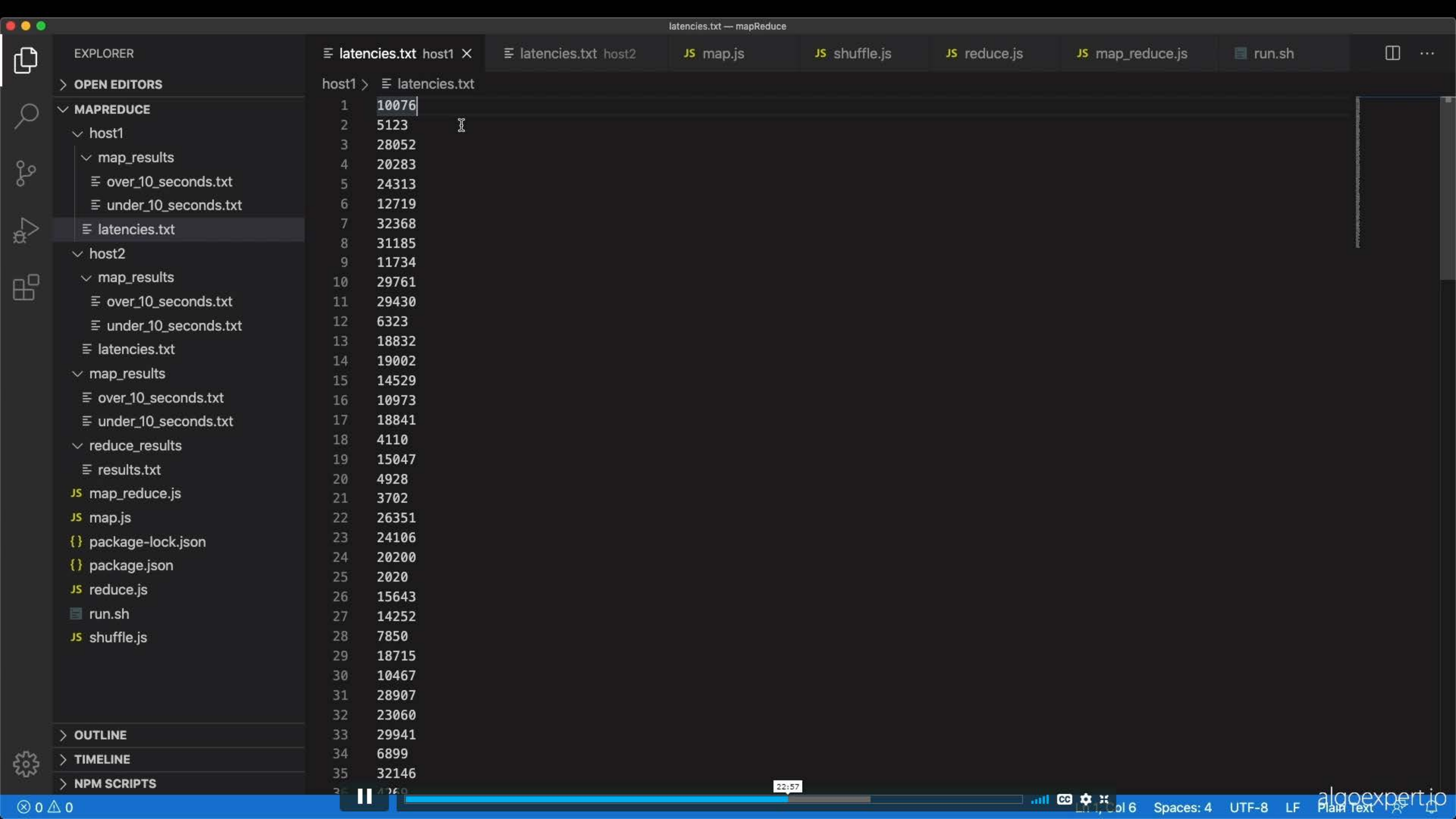
Salient features:

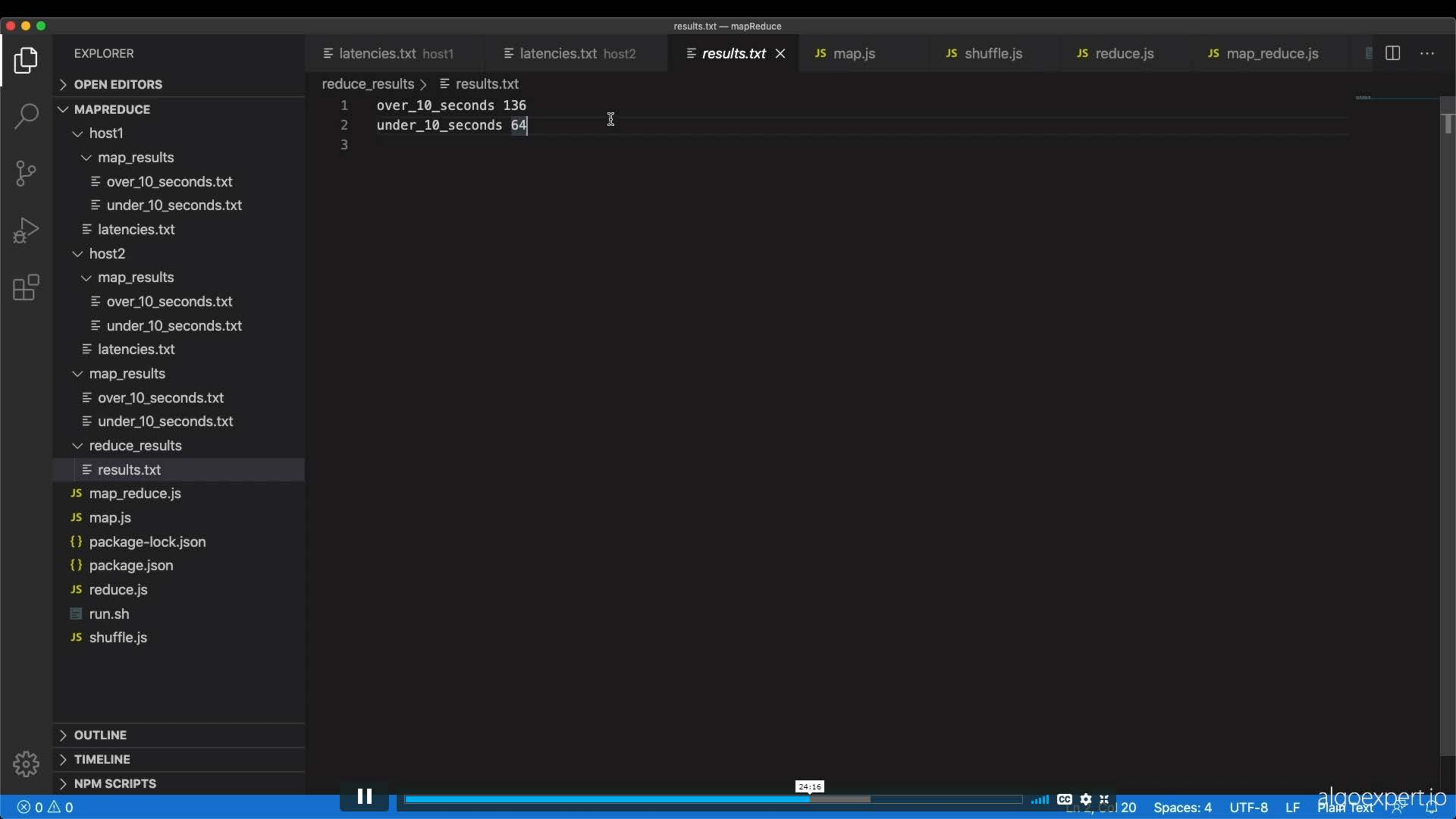
1) Provides availability and replica" guarantees that are tough to achieve in a distri. ed sys

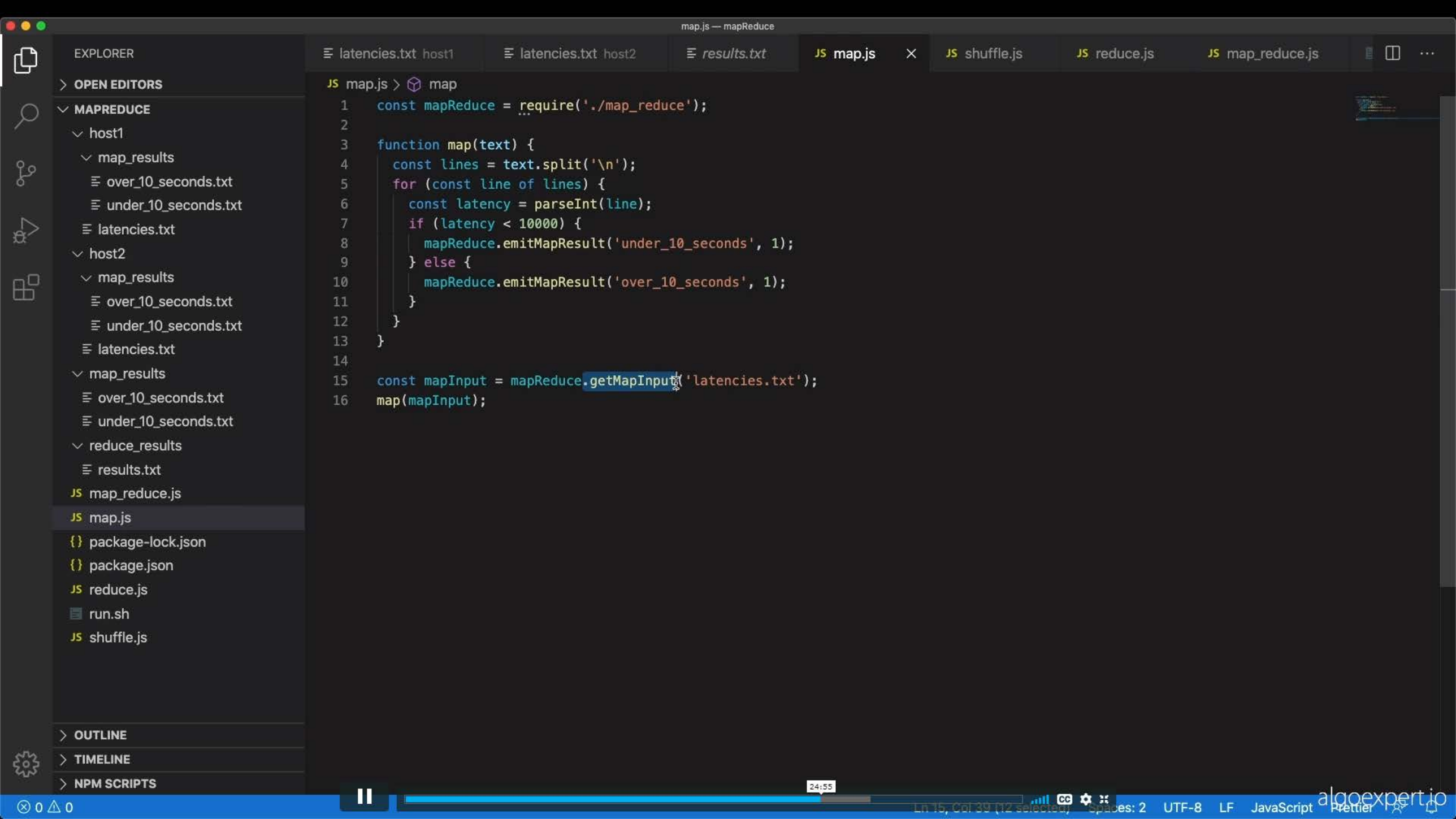
2) File —split→ chunks of certain size (4MB or 64MB) → chunks sharded across a large cluster of machines

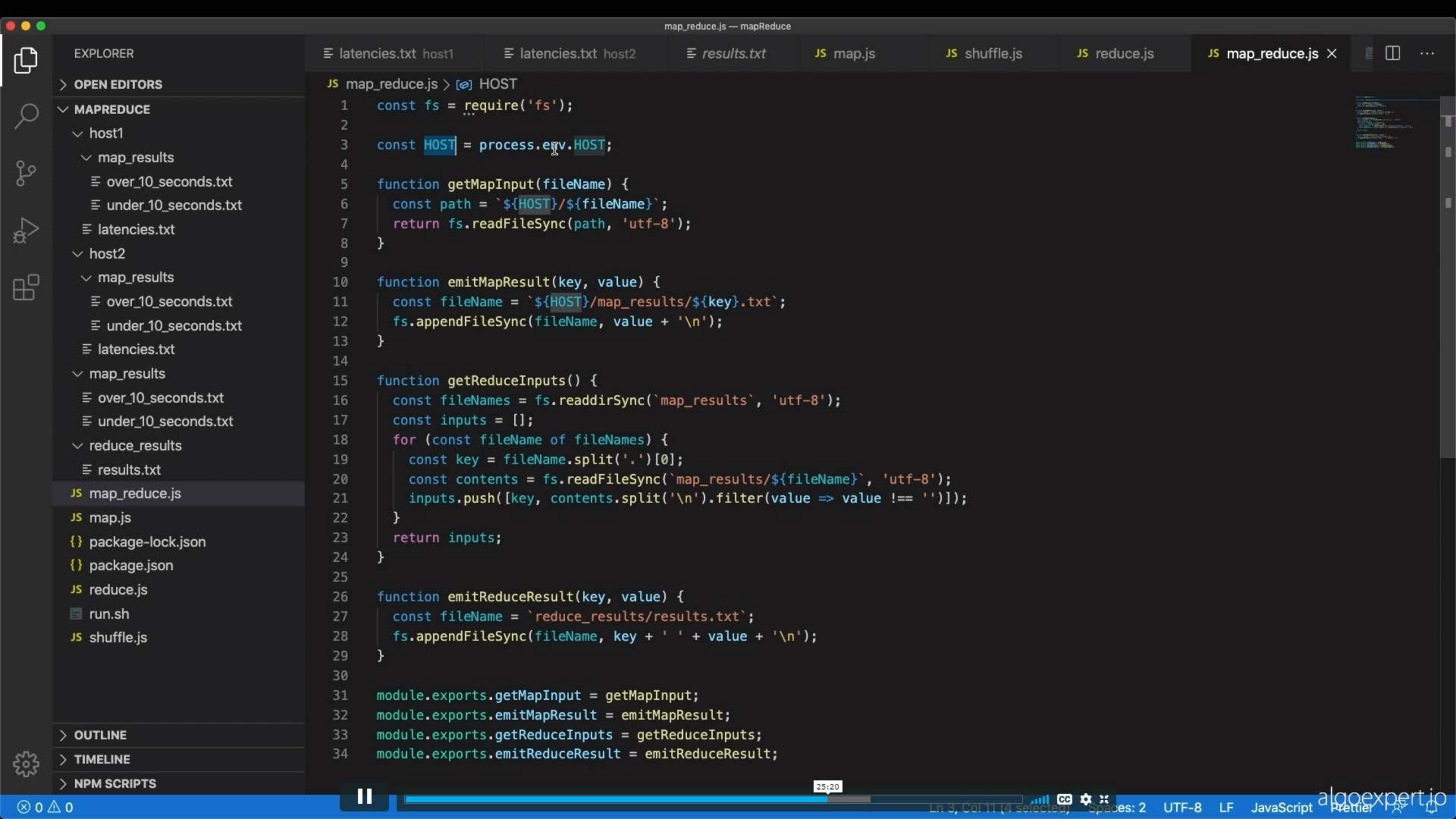
monitored by central control plane ←shards

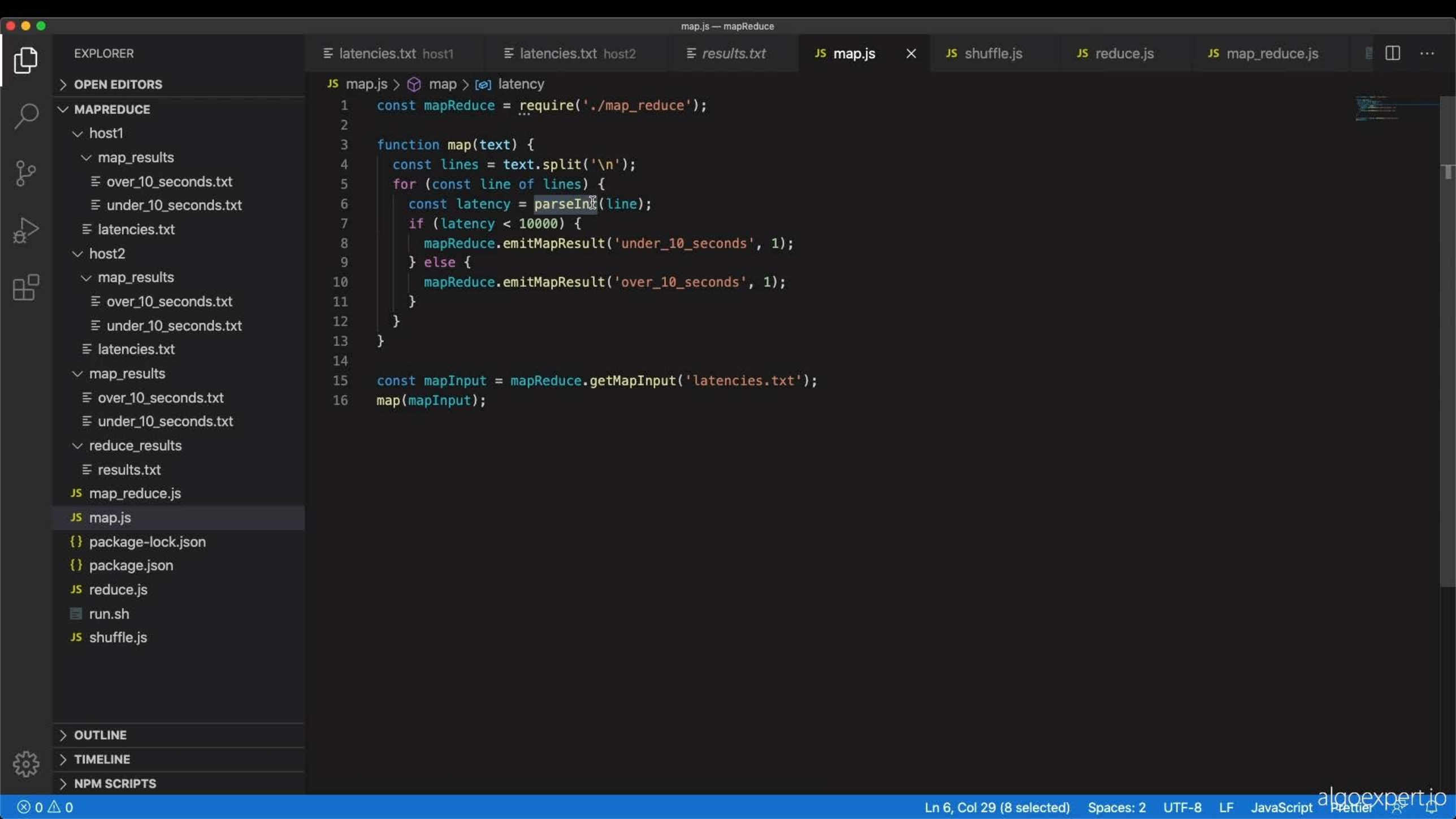
LINK TO mapReduce code example

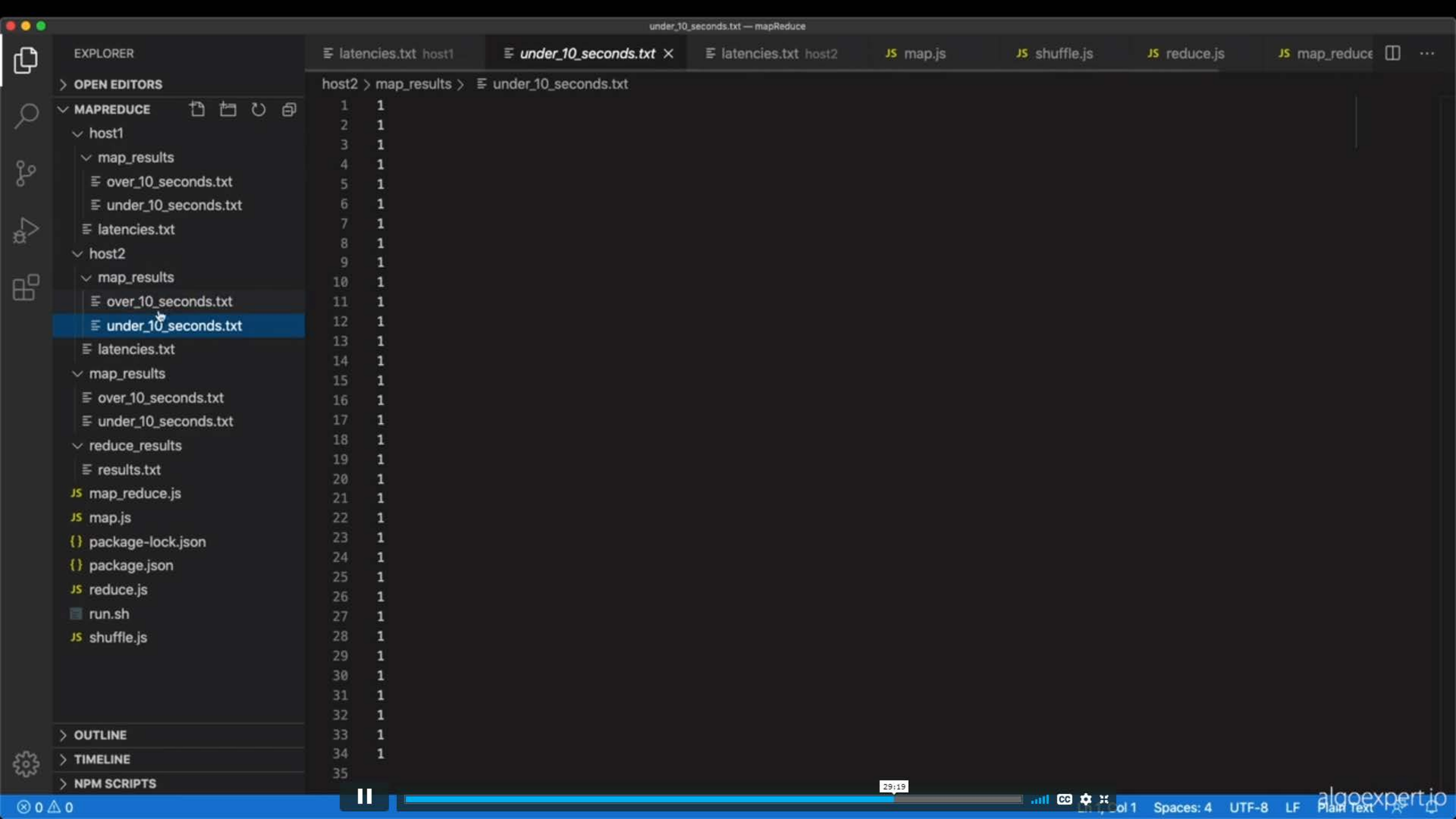

latencies.txt — mapReduce
EXPLORER
latencies.txt host1
latencies.txt host2
map.js
shuffle.js
reduce.js
map_reduce.js
run.sh
host1 > latencies.txt
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
10076
5123
28052
20283
24313
12719
32368
31185
11734
29761
29430
6323
18832
19002
14529
10973
18841
4110
15047
4928
3702
26351
24106
20200
2020
15643
14252
7850
18715
10467
28907
23060
29941
6899
32146
22:57
Ln 1, Col 6
Spaces: 4
UTF-8
LF
Plain Text
algoexpert.io

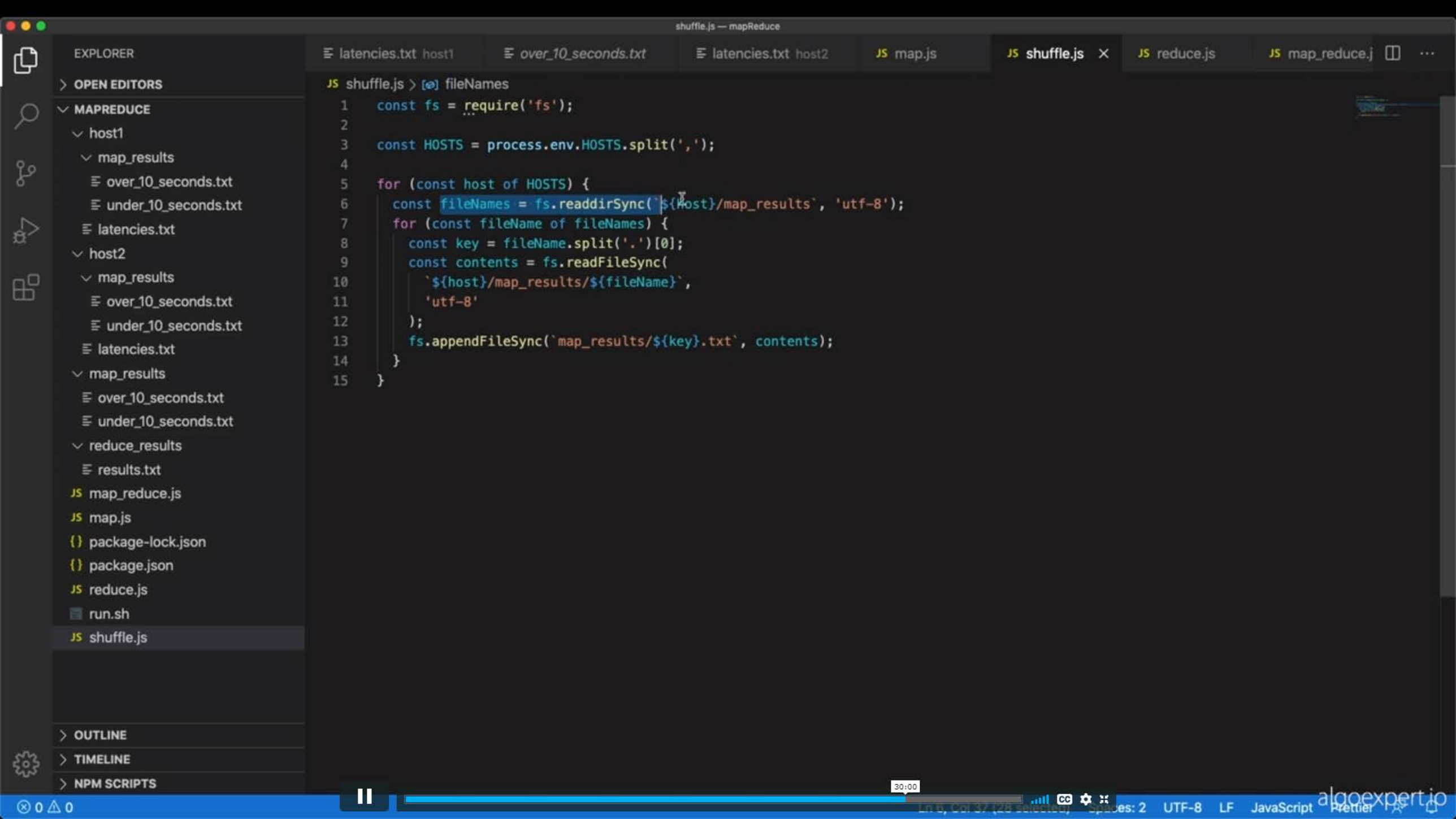

results.txt — mapReduce
EXPLORER
latencies.txt host1
latencies.txt host2
results.txt
map.js
shuffle.js
reduce.js
map_reduce.js
reduce_results > results.txt
1 over_10_seconds 136
2 under_10_seconds 64
3
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
24:16
Spaces: 4
UTF-8
LF
Plain Text
algoexpert.io

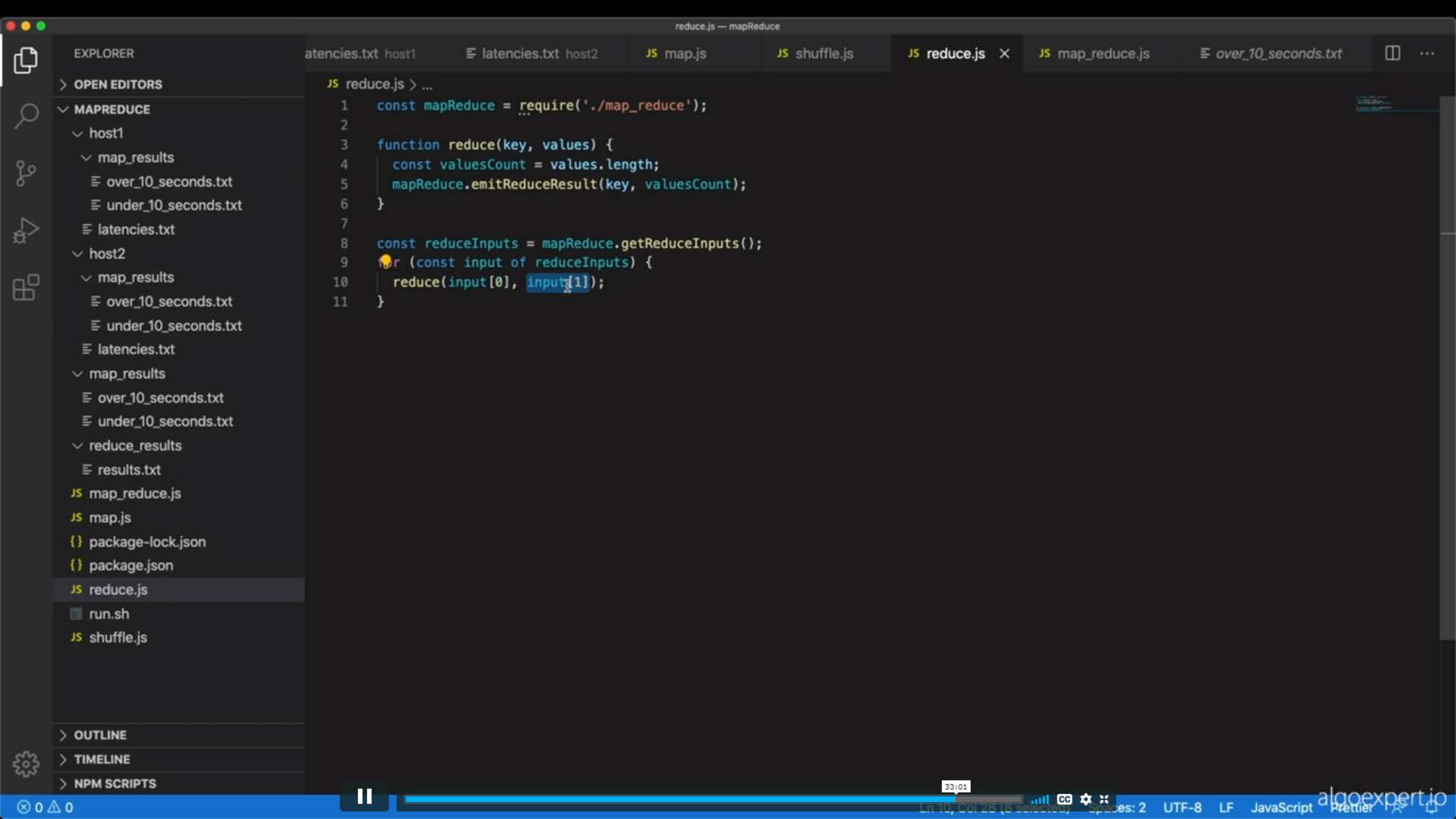

map.js — mapReduce
EXPLORER
latencies.txt host1
latencies.txt host2
results.txt
map.js
shuffle.js
reduce.js
map_reduce.js
map.js > map
1 const mapReduce = require('./map_reduce');
2
3 function map(text) {
4 const lines = text.split('\n');
5 for (const line of lines) {
6 const latency = parseInt(line);
7 if (latency < 10000) {
8 mapReduce.emitMapResult('under_10_seconds', 1);
9 } else {
10 mapReduce.emitMapResult('over_10_seconds', 1);
11 }
12 }
13 }
14
15 const mapInput = mapReduce.getMapInput('latencies.txt');
16 map(mapInput);
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
24:55
algoexpert.io

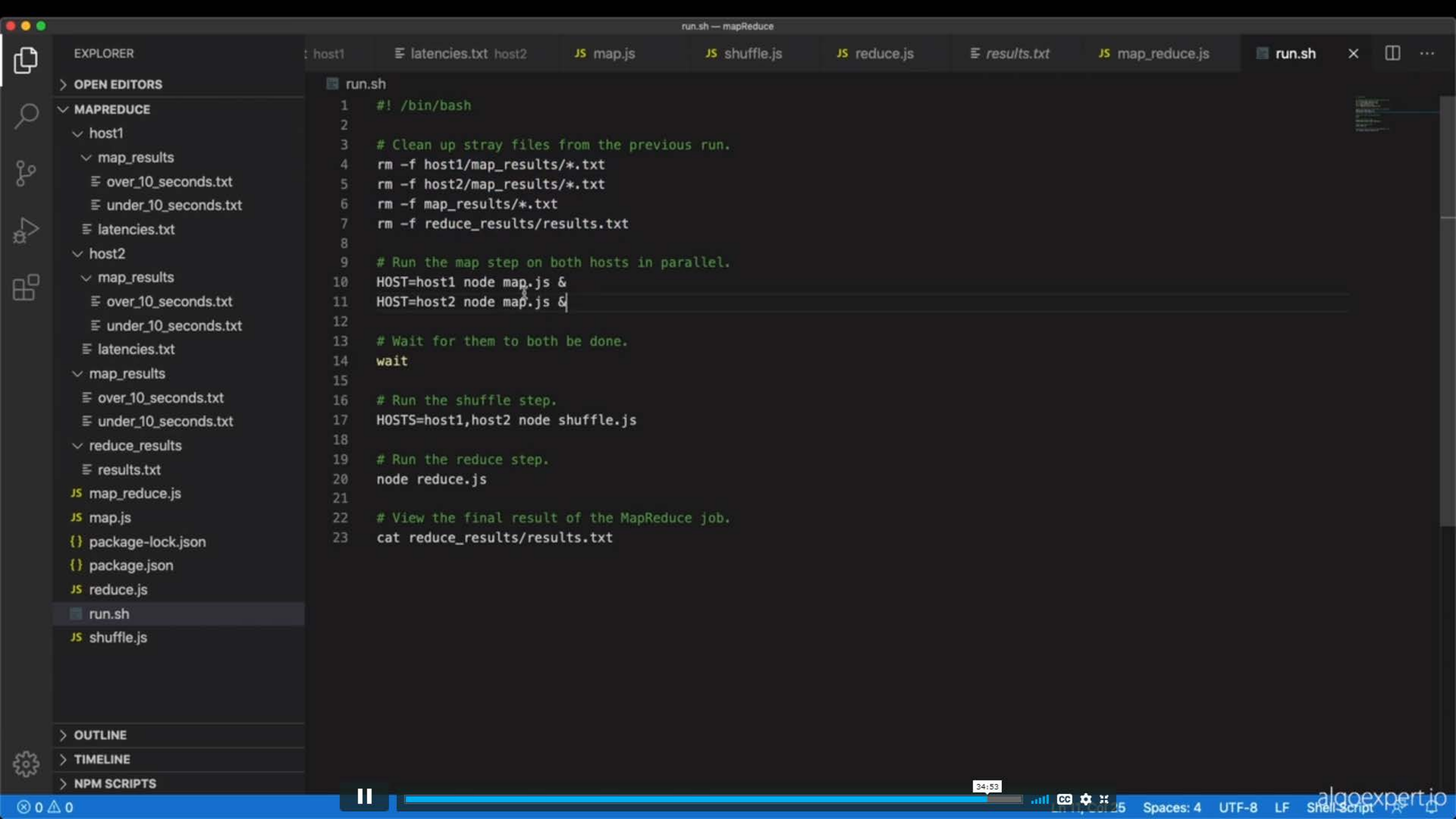

map_reduce.js — mapReduce
EXPLORER
latencies.txt host1
latencies.txt host2
results.txt
map.js
shuffle.js
reduce.js
map_reduce.js
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
map_reduce.js > HOST
1 const fs = require('fs');
2
3 const HOST = process.env.HOST;
4
5 function getMapInput(fileName) {
6 const path = `${HOST}/${fileName}`;
7 return fs.readFileSync(path, 'utf-8');
8 }
9
10 function emitMapResult(key, value) {
11 const fileName = `${HOST}/map_results/${key}.txt`;
12 fs.appendFileSync(fileName, value + '\n');
13 }
14
15 function getReduceInputs() {
16 const fileNames = fs.readdirSync(`map_results`, 'utf-8');
17 const inputs = [];
18 for (const fileName of fileNames) {
19 const key = fileName.split('.')[0];
20 const contents = fs.readFileSync(`map_results/${fileName}`, 'utf-8');
21 inputs.push([key, contents.split('\n').filter(value => value !== '')]);
22 }
23 return inputs;
24 }
25
26 function emitReduceResult(key, value) {
27 const fileName = `reduce_results/results.txt`;
28 fs.appendFileSync(fileName, key + ' ' + value + '\n');
29 }
30
31 module.exports.getMapInput = getMapInput;
32 module.exports.emitMapResult = emitMapResult;
33 module.exports.getReduceInputs = getReduceInputs;
34 module.exports.emitReduceResult = emitReduceResult;
25:20
Ln 3, Col 11 (4 selected)
Spaces: 2
UTF-8
LF
JavaScript
Prettier
algoexpert.io

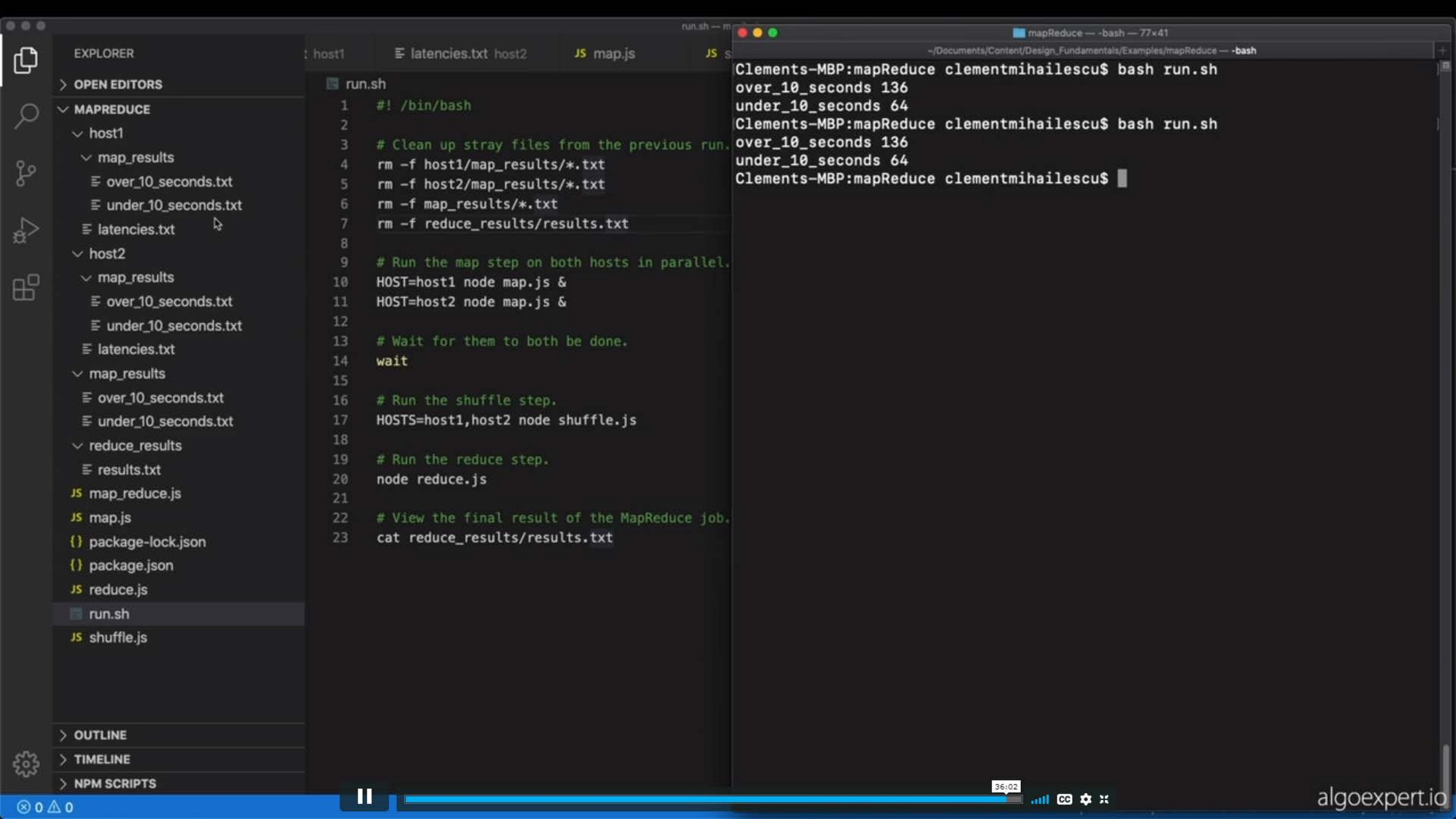



under_10_seconds.txt — mapReduce
EXPLORER
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
latencies.txt host1
under_10_seconds.txt
latencies.txt host2
map.js
shuffle.js
reduce.js
map_reduce
host2 > map_results > under_10_seconds.txt
1 1
2 1
3 1
4 1
5 1
6 1
7 1
8 1
9 1
10 1
11 1
12 1
13 1
14 1
15 1
16 1
17 1
18 1
19 1
20 1
21 1
22 1
23 1
24 1
25 1
26 1
27 1
28 1
29 1
30 1
31 1
32 1
33 1
34 1
35
29:19
Spaces: 4
UTF-8
LF
Plain Text
algoexpert.io


shuffle.js — mapReduce
EXPLORER
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
latencies.txt host1
over_10_seconds.txt
latencies.txt host2
map.js
shuffle.js
reduce.js
map_reduce.j
shuffle.js > fileNames
1 const fs = require('fs');
2
3 const HOSTS = process.env.HOSTS.split(',');
4
5 for (const host of HOSTS) {
6 const fileNames = fs.readdirSync(`${host}/map_results`, 'utf-8');
7 for (const fileName of fileNames) {
8 const key = fileName.split('.')[0];
9 const contents = fs.readFileSync(
10 `${host}/map_results/${fileName}`,
11 'utf-8'
12);
13 fs.appendFileSync(`map_results/${key}.txt`, contents);
14 }
15 }
OUTLINE
TIMELINE
NPM SCRIPTS
30:00
0 0
Spaces: 2
UTF-8
LF
JavaScript
Prettier
algoexpert.io


reduce.js — mapReduce
EXPLORER
latencies.txt host1
latencies.txt host2
map.js
shuffle.js
reduce.js
map_reduce.js
over_10_seconds.txt
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
reduce.js > ...
1 const mapReduce = require('./map_reduce');
2
3 function reduce(key, values) {
4 const valuesCount = values.length;
5 mapReduce.emitReduceResult(key, valuesCount);
6 }
7
8 const reduceInputs = mapReduce.getReduceInputs();
9 for (const input of reduceInputs) {
10 reduce(input[0], input[1]);
11 }
33:01
Spaces: 2
UTF-8
LF
JavaScript
Prettier
algoexpert.io


run.sh — mapReduce
EXPLORER
host1
latencies.txt host2
map.js
shuffle.js
reduce.js
results.txt
map_reduce.js
run.sh
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
run.sh
#! /bin/bash
Clean up stray files from the previous run.
rm -f host1/map_results/*.txt
rm -f host2/map_results/*.txt
rm -f map_results/*.txt
rm -f reduce_results/results.txt
# Run the map step on both hosts in parallel.
HOST=host1 node map.js &
HOST=host2 node map.js &
# Wait for them to both be done.
wait
# Run the shuffle step.
HOSTS=host1,host2 node shuffle.js
# Run the reduce step.
node reduce.js
View the final result of the MapReduce job.
cat reduce_results/results.txt
34:53
Spaces: 4
UTF-8
LF
Shell Script
algoexpert.io


EXPLORER
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
host1
latencies.txt host2
map.js
run.sh
#! /bin/bash
Clean up stray files from the previous run.
rm -f host1/map_results/*.txt
rm -f host2/map_results/*.txt
rm -f map_results/*.txt
rm -f reduce_results/results.txt
# Run the map step on both hosts in parallel.
HOST=host1 node map.js &
HOST=host2 node map.js &
# Wait for them to both be done.
wait
# Run the shuffle step.
HOSTS=host1,host2 node shuffle.js
# Run the reduce step.
node reduce.js
View the final result of the MapReduce job.
cat reduce_results/results.txt
mapReduce — -bash — 77×41
~/Documents/Content/Design_Fundamentals/Examples/mapReduce — -bash
Clements-MBP:mapReduce clementmihailescu$ bash run.sh
over_10_seconds 136
under_10_seconds 64
Clements-MBP:mapReduce clementmihailescu$ bash run.sh
over_10_seconds 136
under_10_seconds 64
Clements-MBP:mapReduce clementmihailescu$
36:02
algoexpert.io

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle$`URL`, `1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL`, `total count`$\rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle$`target`, `source`$\rangle$ pairs for each link to a `target` URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle$`target`, $list$(`source`)$\rangle$

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency \rangle$ pairs. The map function emits a $\langle$`hostname`, `term vector`$\rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle$`hostname`, `term vector`$\rangle$ pair.

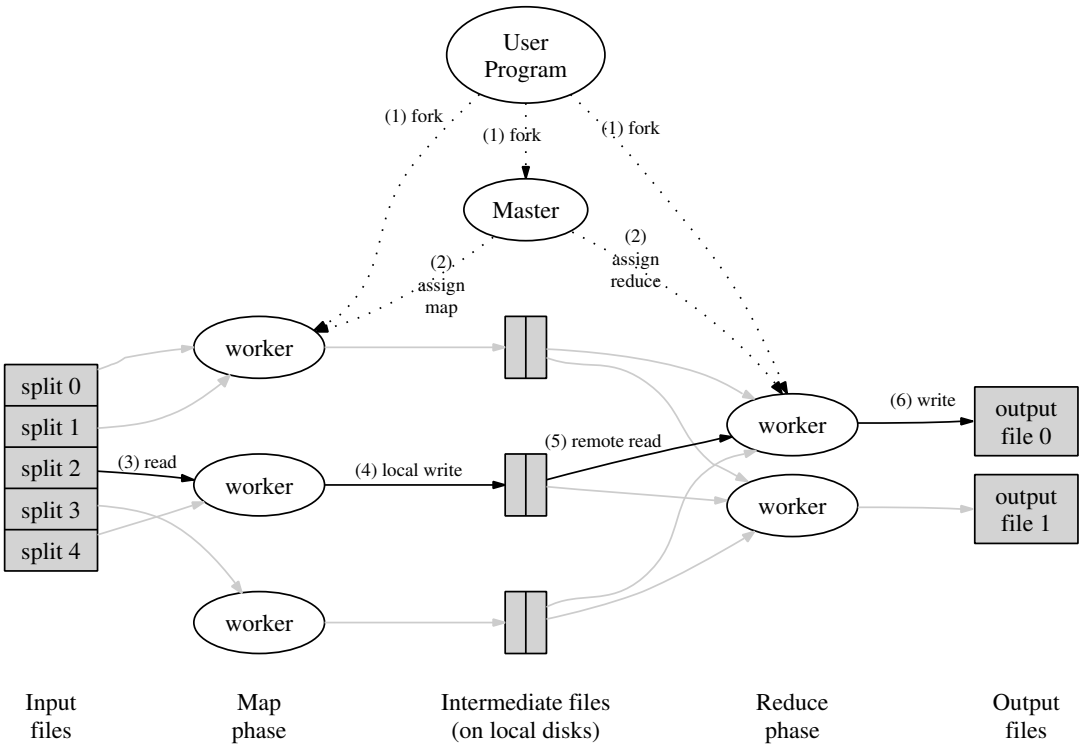


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of $\langle$`word, document ID`$\rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle$`word`, $list$(`document ID`)$\rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a $\langle$`key, record`$\rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

into a set of M *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) \bmod R$). The number of partitions (R) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the `MapReduce` call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces R such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of R files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task R_1 is equivalent to the output for R_1 produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution of the non-deterministic program.

Consider map task M and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_i that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of M and $e(R_2)$ may have read the output produced by a different execution of M.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large M and R can be in our implementation, since the master must make $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (The constant factors for memory usage are small however: the $O(M * R)$ piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "$hash(key) \bmod R$"). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey)) \bmod R$" as the partitioning function causes all URLs from the same host to end up in the same output file.

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text"

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

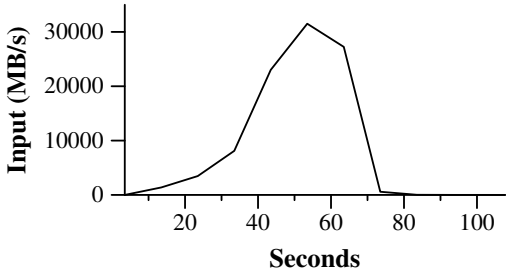


Figure 2: Data transfer rate over time

5.2 Grep

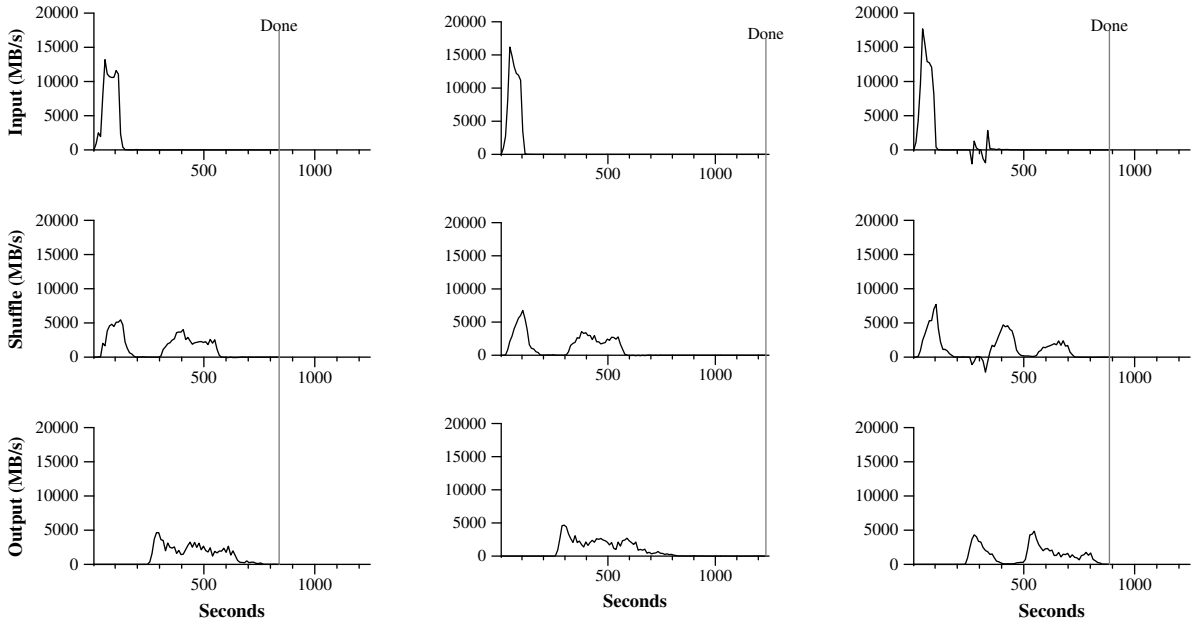
The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

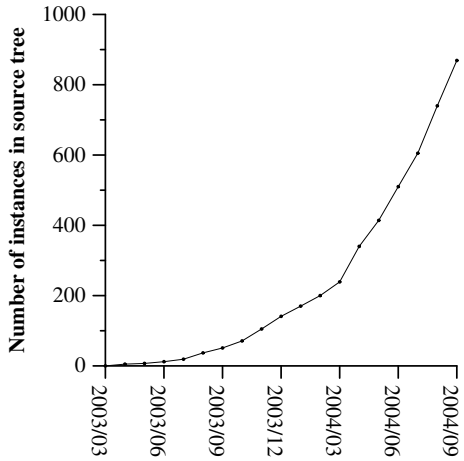


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an N element array in $\log N$ time on N processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of R reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

Date:
M T W T F S S

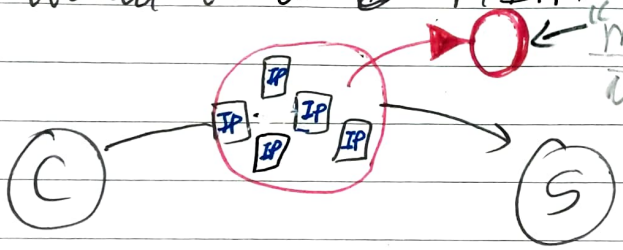
Security & HTTPS

Stuff you gotta know to understand HTTPS

I] Man In the Middle Attack (MITM)

i) An attack in which the attacker intercepts a line of comm" i.e. thought to be pvt by its 2 communicating parties

ii) A malicious actor intercepted and mutated an IP packet on its way from client to server, that would be a MITM attack



"malicious client" intercepts IP packets
① Goes thru sensitive info that they contain
② Manipulates data exchanged b/w client - server for some other purpose

iii) MITM attacks are the primary threat to privacy, cybersecurity and what encryp", HTTPS aim to defend against

2] Encryp": A way of securing data exchanged b/w parties by converting data into an unrecognizable string of random bytes and giving only concerned parties the power to reconvert it back into understandable data.

2 types of encryp^n

(1) Symmetric encryp^n

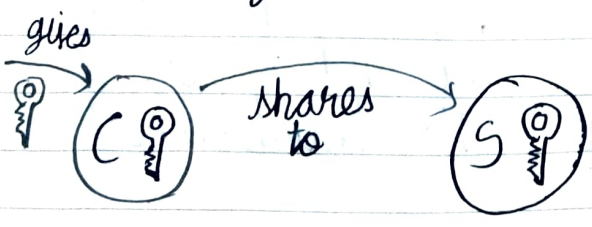
i) Type of encryp^n that relies on a single key (🔑) to both encrypt, decrypt data

ii) Key must be known to all parties involved in comm^n and must ∴ typically be shared b/w parties at 1 point or another → this is a vulnerable pt. as sharing can be intercepted by a MITM attack.

iii) Faster

iv) Symmetric key algos are (mostly part of the AES) used to generate the key.

symm. key algo gives



Use : for comm^n after secure connec^n b/w client and server has been established

(2) Asymmetric encryp^n

The type of encryp^n that relies on 2 keys → a public key (🔑) and a prvt. key (🔑) to encrypt, decrypt

The party involved should only know the key it is concerned with i.e. if it is concerned with
a) decryptin → prvt. key
b) encryptin → public key.

Slower

Keys generated usin cryptographic algos and are mathematically connected s.t. the data encrypted with public key can only be decrypted with prvt. key.

Use : to establish a secure connec^n b/w client-server

3] AES (Advanced Encrypⁿ Std.)

i) A widely used encrypⁿ std. that has 3 symm. key algos
- AES-128
- AES-192
- AES-256

ii) "Gold std" of encrypⁿ

iii) Use → U.S. National Security Agency to encrypt top secret info.

4] HTTPS (Hyper Text Transfer Protocol)

i) Extension of HTTP that runs on top of TLS (Transport Layer security). It a.k.a. HTTP on top of TLS

ii) Used for secure commⁿ online

iii) Requires servers to pr trusted certificates (SSL certificates) and uses the TLS, a security protocol built on top of TCP to encrypt data communicated b/w a client and a server. This process is a.k.a. "TLS Handshake"

5] TLS

The Transport Layer Security (TSL) is a protocol over which HTTP runs in order to achieve secure commⁿ online. HTTP over TLS is a.k.a. HTTPS

6) SSL certificate (Secure socket layer)

1) A digital certificate granted to a server by a certification authority (CA) → contains

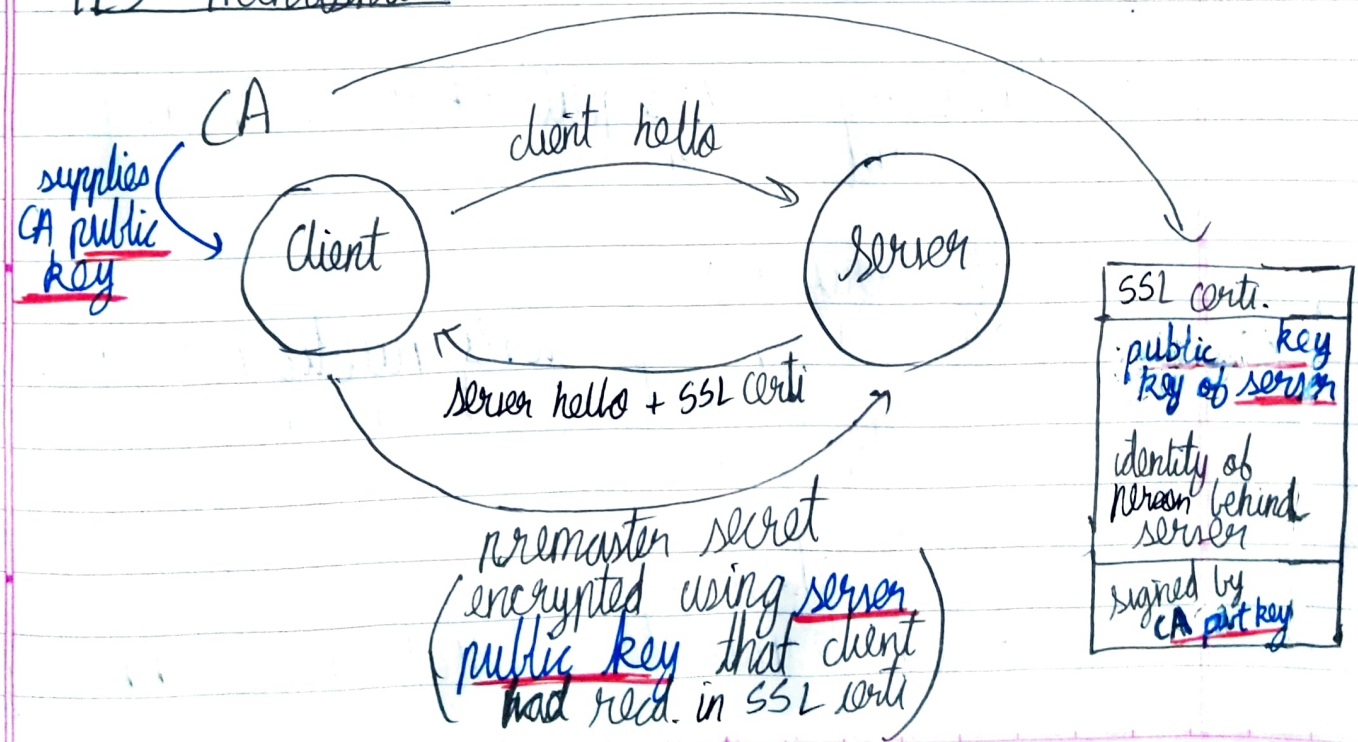
- (1) server's public key
- (2) name of entity owning public-pvt key pair i.e. the company whose server this is
- signed by CA's pvt key

2) Server's public key is used as part of the TLS handshake process in an HTTPS connection

3) An SSL certi effectively confirms that the public key belongs to the server claiming that it's their key.

4) SSL certi are a crucial step in defending against MITM attacks as they
   - ↳ provide identity of server
   - ↳ check and verify this identity

7) TLS Handshake



CA (Certificate Authority) → A trusted entity that signs digital certs namely SSL certs that are relied on in HTTPS connec^n's

TLS Handshake steps:

1) Client sends "client hello" – str. of random bytes – to server to start handshake process

2) Server responds by generation server's public-pvt key pair and sendin back 2 things:
   i) "Server Hello" – str of random bytes acknowledgin initia^n of HTTPS connec^n by client
   ii) SSL certi. contains → server's public key
                           → CA's pvt. key
                           → server owner's identity

Meanwhile, client obtains CA's public key from CA itself.

iii) Client uses CA's public key to verify SSL certi. sent by server
   a) If verified ✓
      client accepts server's public key given in SSL certi and uses it in the next step of the process
   b) If not verified X
      client terminate's handshake, connec^n fails

iv) Client generates a "premaster secret" encrypted using the server's public key it just recd.

v) Server uses its own pvt. key i.e. server's pvt. key to gain access to 'premaster secret'

vi) Now client and server both hv access to 3 things:

- client hello
- server hello
- premaster secret.

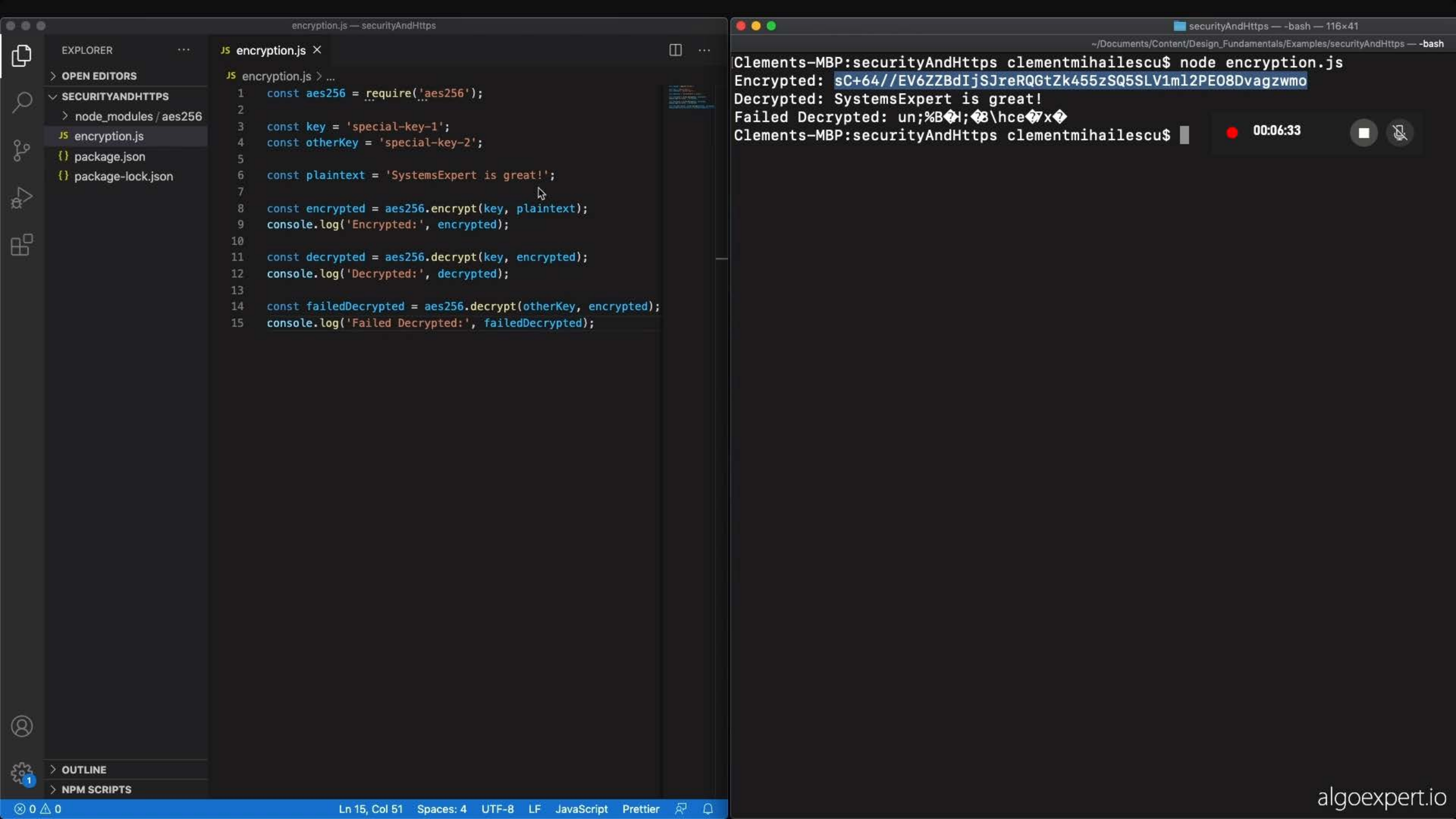
} + key algo = session key

HTTPS connec" established

vii) Using these 3 things and a symm. key algo, they generate a symmetric "session key" that they'll both store and use to decrypt/encrypt and data exchange in that session.

HTTPS connec" terminated

viii) Once connec" is terminated, curr. session key is disposed off and a new one will be generated by the "TLS Handshake" process next time around.


encryption.js — securityAndHttps
EXPLORER
OPEN EDITORS
SECURITYANDHTTPS
node_modules / aes256
encryption.js
package.json
package-lock.json
encryption.js
const aes256 = require('aes256');
const key = 'special-key-1';
const otherKey = 'special-key-2';
const plaintext = 'SystemsExpert is great!';
const encrypted = aes256.encrypt(key, plaintext);
console.log('Encrypted:', encrypted);
const decrypted = aes256.decrypt(key, encrypted);
console.log('Decrypted:', decrypted);
const failedDecrypted = aes256.decrypt(otherKey, encrypted);
console.log('Failed Decrypted:', failedDecrypted);
OUTLINE
NPM SCRIPTS
Ln 15, Col 51
Spaces: 4
UTF-8
LF
JavaScript
Prettier
securityAndHttps — -bash — 116×41
~/Documents/Content/Design_Fundamentals/Examples/securityAndHttps — -bash
Clements-MBP:securityAndHttps clementmihailescu$ node encryption.js
Encrypted: sC+64//EV6ZZBdIjSJreRQGtZk455zSQ5SLV1ml2PEO8Dvagzwmo
Decrypted: SystemsExpert is great!
Failed Decrypted: un;%B?H;?8\hce??x?
Clements-MBP:securityAndHttps clementmihailescu$
00:06:33
algoexpert.io

Date:
M T W T F S S

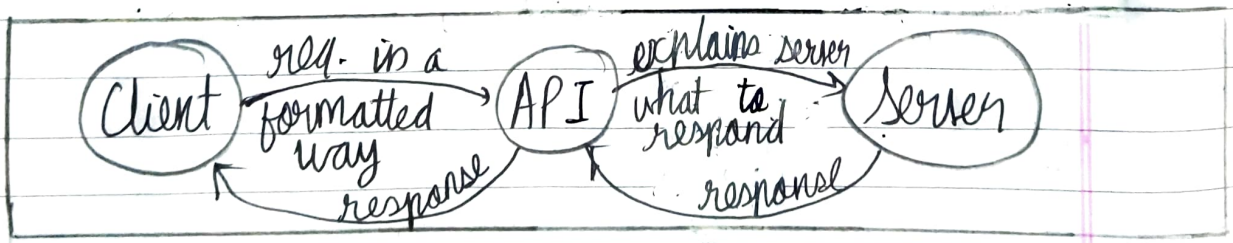
API Design

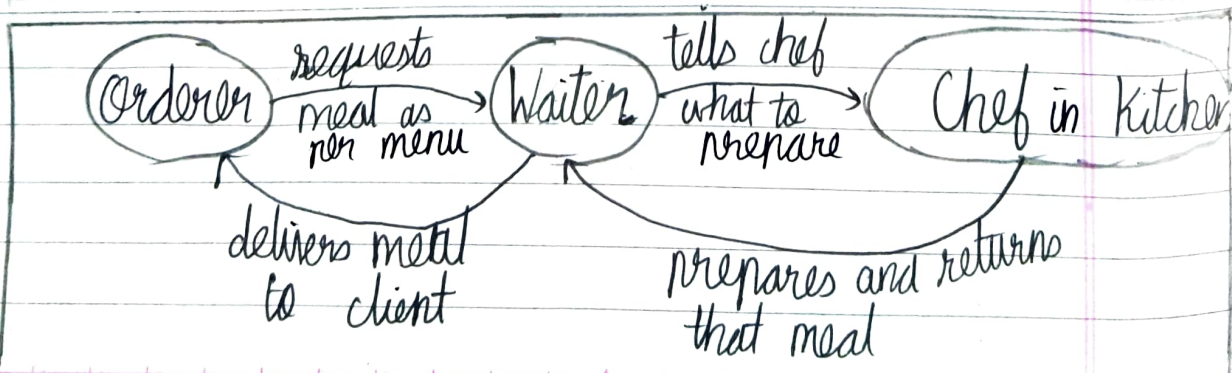
IMP: A "sibling" of sys. design and NOT a "subset"

Importance of API design:

API → Applica^n Program Interface

- A contract provided by 1 piece of software to another piece of software
- Consists of structured req., res.
  One piece of software says "gimme this info formatted in this way and I'll return this f^n / data / whatever reqd. res. is."
- Analogy: Ordering food in a restaurant





API is at the core of a service's backend.

Also certain software products like Stripe (payment transac" authentica" and conduc" service) have the API as the core/main service (enablin payments)

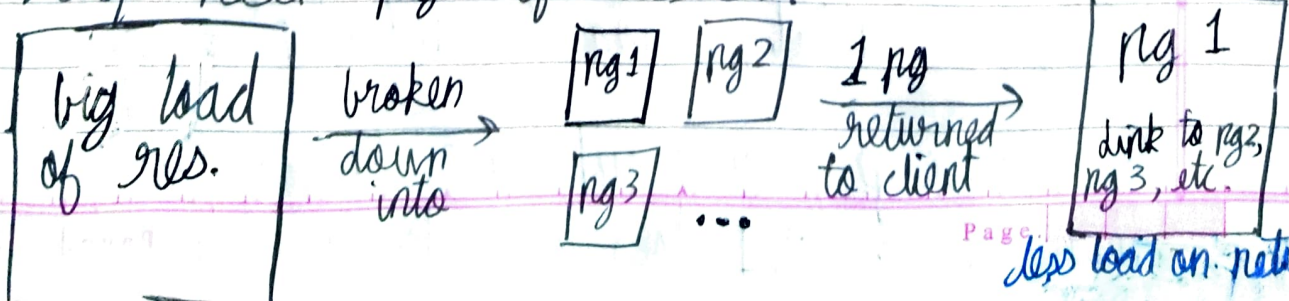
API Design Importance

1. A lotta customers or services, gonna use your API / depend on your API
2. Any decision in its designing (eg:- name of param. in an endpt.) has huge impacts on users
3. ∴ When a new API is developed, its gotta go thru a rigorous design & review process to make it easy to use and update

Some salient API design features

Pagina":

When a network req potentially warrants a really large res., the relevant API might be designed to return only a single or couple page(s) of that res. (i.e. a "de nor" of the res. accompanied by the identifier/token for client to req next pg. if desired.



less load on network

Date:
M T W T F S S

Pagination is often used to list endpts.

eg:- Endpt. to list videos on the YouTube trendin page

i) Cannot expect just 1 endpt. to return a huge list of vids.

ii) Reason - It wouldn't perform very well on mobile devices due to lower network speeds (unoptimal)

Also, unoptimal cuz most users only scroll thru 1st 10 or 20 vids.

iii) API is designed to respond with only the 1st few vids. of that list. i.e. API res. is paginated.

CRUD oprns.

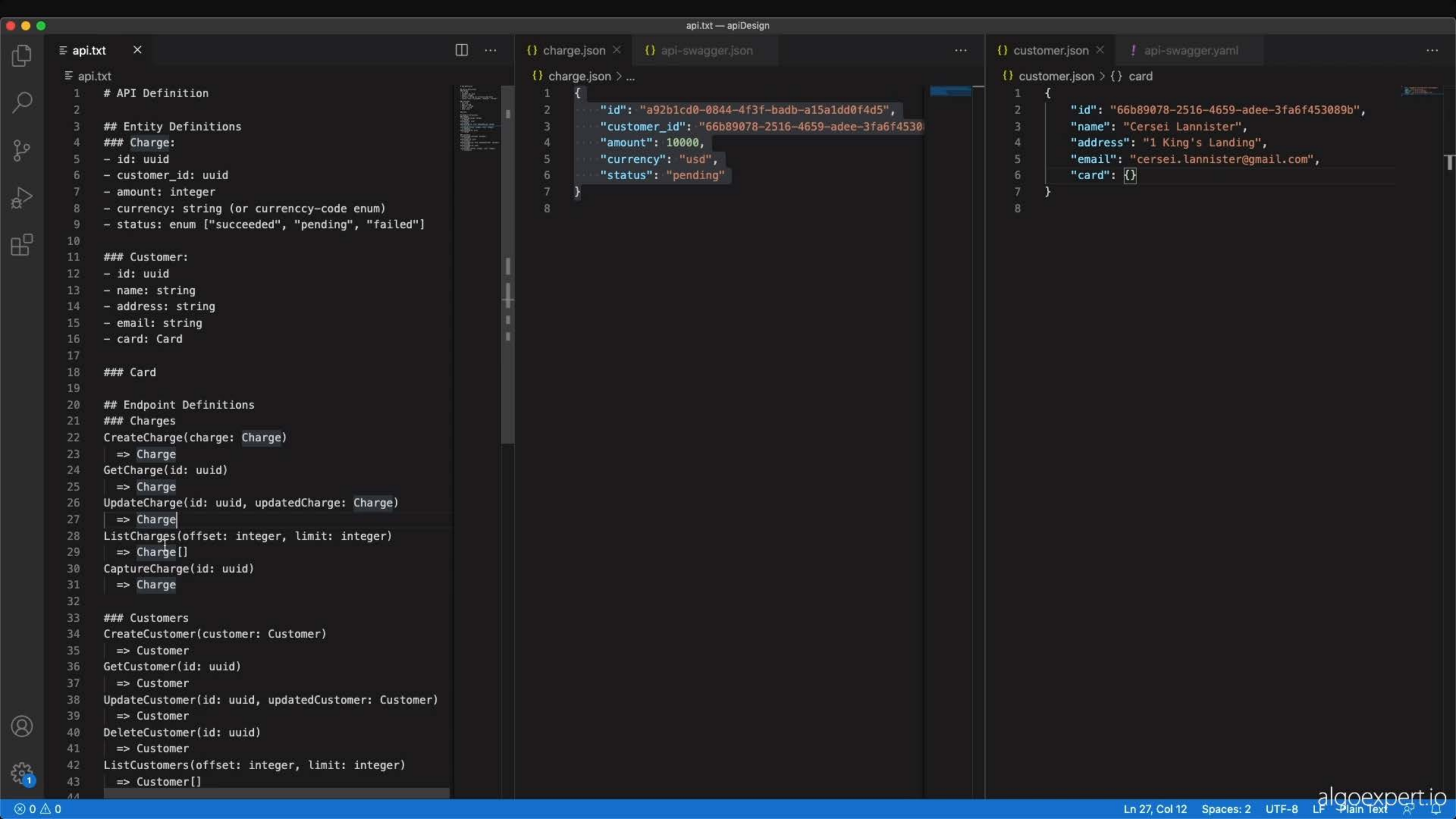
C - Create
R - Read
U - Update
D - Delete

} 4 oprns. servin as a bedrock of a f^n in sys and ∴ find themselves at the core of a lotta APIs.

PRO-TIPS:

(1) Read up APIs of some popular services. eg:- GitHub, Google Cloud IOT

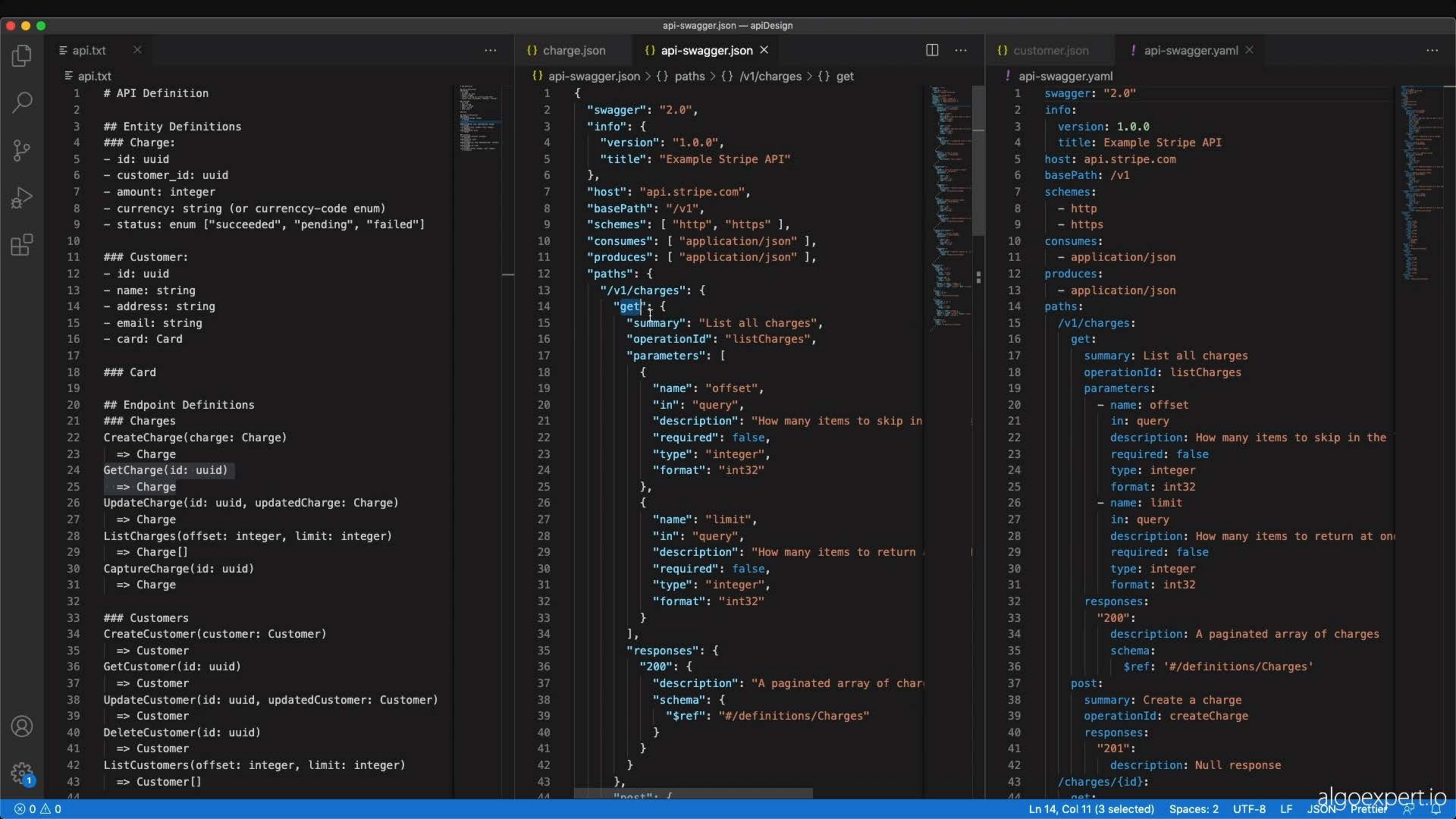
(2) Try to figure out API design for a service, then see its API docs to understand how close you are.




```
# API Definition

## Entity Definitions
### Charge:
- id: uuid
- customer_id: uuid
- amount: integer
- currency: string (or currenccy-code enum)
- status: enum ["succeeded", "pending", "failed"]

### Customer:
- id: uuid
- name: string
- address: string
- email: string
- card: Card

### Card

## Endpoint Definitions
### Charges
CreateCharge(charge: Charge)
  => Charge
GetCharge(id: uuid)
  => Charge
UpdateCharge(id: uuid, updatedCharge: Charge)
  => Charge
ListCharges(offset: integer, limit: integer)
  => Charge[]
CaptureCharge(id: uuid)
  => Charge

### Customers
CreateCustomer(customer: Customer)
  => Customer
GetCustomer(id: uuid)
  => Customer
UpdateCustomer(id: uuid, updatedCustomer: Customer)
  => Customer
DeleteCustomer(id: uuid)
  => Customer
ListCustomers(offset: integer, limit: integer)
  => Customer[]
```

```
{
    "id": "a92b1cd0-0844-4f3f-badb-a15a1dd0f4d5",
    "customer_id": "66b89078-2516-4659-adee-3fa6f4530
    "amount": 10000,
    "currency": "usd",
    "status": "pending"
}
```

```
{
    "id": "66b89078-2516-4659-adee-3fa6f453089b",
    "name": "Cersei Lannister",
    "address": "1 King's Landing",
    "email": "cersei.lannister@gmail.com",
    "card": {}
}
```


api-swagger.json — apiDesign
api.txt
charge.json
api-swagger.json
customer.json
api-swagger.yaml
api-swagger.json > { } paths > { } /v1/charges > { } get
API Definition
Entity Definitions
Charge:
- id: uuid
- customer_id: uuid
- amount: integer
- currency: string (or currenccy-code enum)
- status: enum ["succeeded", "pending", "failed"]
Customer:
- id: uuid
- name: string
- address: string
- email: string
- card: Card
Card
Endpoint Definitions
Charges
CreateCharge(charge: Charge)
=> Charge
GetCharge(id: uuid)
=> Charge
UpdateCharge(id: uuid, updatedCharge: Charge)
=> Charge
ListCharges(offset: integer, limit: integer)
=> Charge[]
CaptureCharge(id: uuid)
=> Charge
Customers
CreateCustomer(customer: Customer)
=> Customer
GetCustomer(id: uuid)
=> Customer
UpdateCustomer(id: uuid, updatedCustomer: Customer)
=> Customer
DeleteCustomer(id: uuid)
=> Customer
ListCustomers(offset: integer, limit: integer)
=> Customer[]
{
"swagger": "2.0",
"info": {
"version": "1.0.0",
"title": "Example Stripe API"
},
"host": "api.stripe.com",
"basePath": "/v1",
"schemes": ["http", "https"],
"consumes": ["application/json"],
"produces": ["application/json"],
"paths": {
"/v1/charges": {
"get": {
"summary": "List all charges",
"operationId": "listCharges",
"parameters": [
{
"name": "offset",
"in": "query",
"description": "How many items to skip in
"required": false,
"type": "integer",
"format": "int32"
},
{
"name": "limit",
"in": "query",
"description": "How many items to return
"required": false,
"type": "integer",
"format": "int32"
}
],
"responses": {
"200": {
"description": "A paginated array of char
"schema": {
"$ref": "#/definitions/Charges"
}
}
}
},
swagger: "2.0"
info:
version: 1.0.0
title: Example Stripe API
host: api.stripe.com
basePath: /v1
schemes:
- http
- https
consumes:
- application/json
produces:
- application/json
paths:
/v1/charges:
get:
summary: List all charges
operationId: listCharges
parameters:
- name: offset
in: query
description: How many items to skip in the
required: false
type: integer
format: int32
- name: limit
in: query
description: How many items to return at on
required: false
type: integer
format: int32
responses:
"200":
description: A paginated array of charges
schema:
$ref: '#/definitions/Charges'
post:
summary: Create a charge
operationId: createCharge
responses:
"201":
description: Null response
/charges/{id}:
Ln 14, Col 11 (3 selected)
Spaces: 2
UTF-8
LF
JSON
Prettier
algoexpert.io