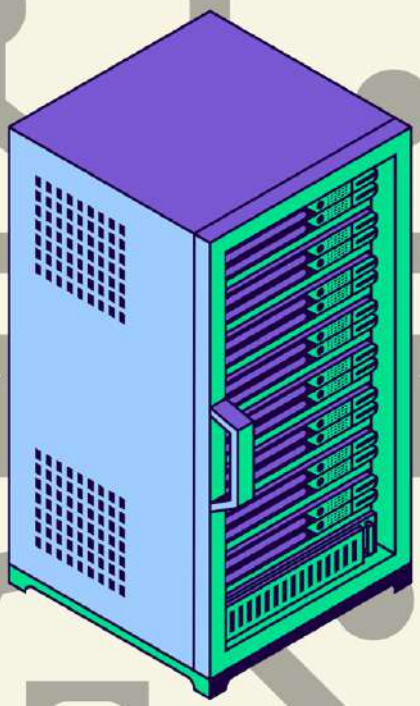
System Design Notes

(High Level Design)



By- Preksha Mahajan

System Design -

→ Software Development is inspired from Real World.

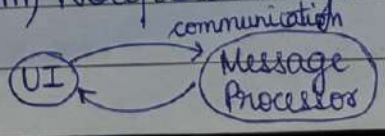
- Creating a City :-

High Level Plan - i) Schools
ii) Houses → Area, Rooms, Size, Materials
iii) Hospitals
iv) Roads
v) Gyms

→ In case of software - High Level Planning
↓
Low Level / Detailed Planning
↓
Execution
This is called System Design.

→ For building Software -
1. Requirements → customer
→ self-researched
2. Prioritization / Phases → P_1, P_2, P_3
3. Planning (with features in hand)
Resource Planning + Budget → infrastructures & estimations → ($)
storage, compute, network
4. Components of System

E.g: In Facebook Messenger, there are components like :
i) UI
ii) Message Processing System
iii) Notification System



(5.) Decide tech stack to build components and how they will communicate with each other.

These stand under HLD (High Level Design).

Requirements

May be conveyed by customer ← Functional

- i) Needed by customers
- ii) App functions

E.g: User → chat
User → send message

Non-functional → In-built / Expected

Qualities & characteristics

1) Performance
2) Latency
3) Security
4) Ease of use
5) Reliability
6) Scalability
7) Availability
8) Consistency

→ A good software project delivers both functional and non-functional requirements.

(6.) Detailing of each component

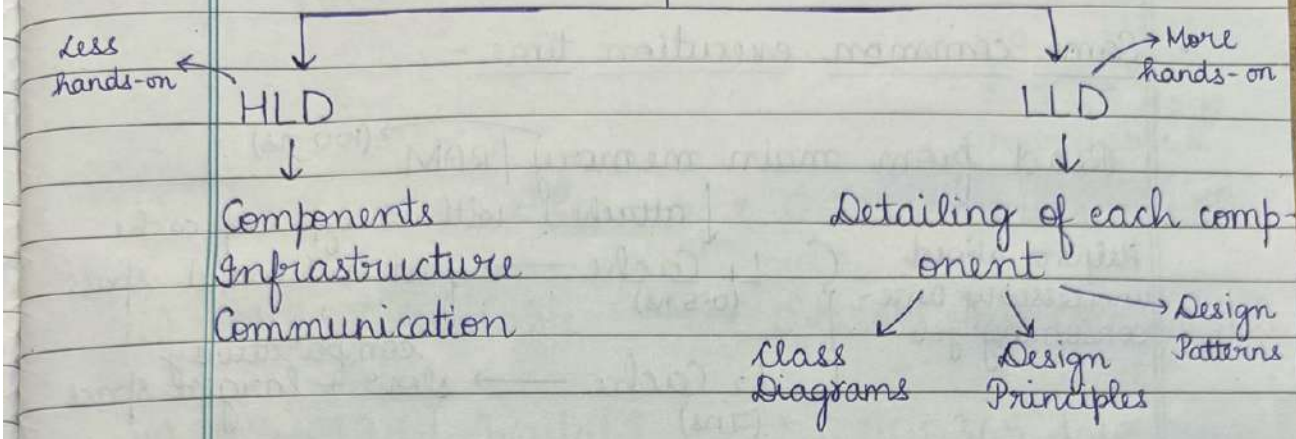
- SOLID Design Principles
- Design Patterns
- OOPS

E.g: Message Processing Service

- classes
- relationships
- behaviours

This comes under LLD (Low Level Design).

- System Design



→ Software Development is not just about writing code; you should also know about where the code is running, how many resources are needed and how much storage is needed, etc.
These type of computations are called as Back of the Envelope calculations.
↳ Infrastructure estimations

- 1 Byte = 8 bits
  1 ASCII character → 1 byte
  ↳ American Standard Code for Information Interchange
  (A type of coding system)
  converts character → numerical code
  E.g: 'A' = 65

1 KB = 1024 Bytes ≈ 10^3 Bytes → 1 Million Bytes
1 MB = 10^6 Bytes = 10^3 KB → 1 Billion Bytes
1 GB = 10^9 Bytes = 10^6 KB = 10^3 MB
1 TB = 10^{12} Bytes = 10^9 KB = 10^6 MB = 10^3 GB
1 PB = 10^{15} Bytes = 10^{12} KB = 10^9 MB = 10^6 GB = 10^3 TB

- Units of time - seconds → s
  milliseconds → ms
  microseconds → μs
  nanoseconds → ns

$1s = 10^3\ ms = 10^6\ \mu s = 10^9\ ns$

- Some common execution time -

Read from main memory / RAM → (100 ns)
↓ attached with 2 types of cache

Helps to avoid unnecessary time-consuming job

- L1 Cache (0.5 ns) → fast + limited space
- L2 Cache (7 ns) → comparatively slow + larger space

E.g: Arrays store elements in contiguous locations. So, they are fit to be stored in L1 cache.
⇒ Very fast read/write [Not in case of Linked List]
Elements are scattered in Linked List.

→ Reading Data - ① Main Memory
② Disk
③ Network

To read 1MB of data sequentially,
1) Main Memory → 250 μs :- Fastest
2) Disk → 30 ms :- Slowest
3) Network → 10 ms :- Slower

→ In case of Big Data, disk I/O, network I/O → expensive → slowest → Map-Reduced Operations
RAM/memory I/O → fastest
↳ Done by Spark : in-memory computation

- Availability - If system responds back at any time (even if it returns an error), it is said to be available.

i) Available → Downtime → Years
99.% 1.%
$\frac{1}{100} \times 365$
= 3.65 days unavailable

ii) 99.9 % Available → 0.365 days unavailable

iii) 99.99 % Available → 0.0365 days unavailable

iv) 99.9999 % Available → 0.000365 days unavailable

→ Companies have a complex mix of:
compute → faults
storage → faults
network → unreliable
→ Data Centre Failures
Electricity Issues
Natural Calamities
Human Error

- Distributed Systems / Computation -

data → storage → Databases
data → computation → Programming Languages

Data that we are creating is high in:
- volume
- velocity
- veracity

Thus, data is becoming utterly complex.

Initially, data is stored and computed by a single machine.
But data is getting complex day-by-day.

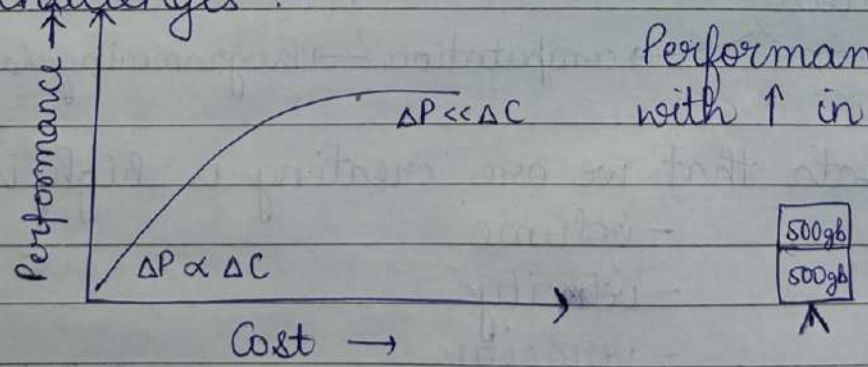
So, instead of a single machine, when data is stored in multiple machines to distribute work and storage and all these machines being connected to each other over a network, this type of arrangement is called distributed systems.

→ For an end user, it seems as if they are interacting with a single machine but internally the data is distributed on multiple machines.
→ This group of machines is also called as CLUSTER of machines.

- Vertical Scaling -

Increasing the size and configuration of a machine more and more is termed as vertical scaling. (Intuitive Solution)
→ It will involve some downtime.
→ It is done to compensate the load coming on the system.
→ Challenges :-

i)



ii) The single machine can become the single point of failure. You lose all the expertise and expenditure put on the machine.
iii) Hardware limitations can lead to stagnant performances.

Due to these challenges, the concept of distributed systems comes in.

- Horizontal Scaling -

→ It is referred to as the centre of Distributed Systems.

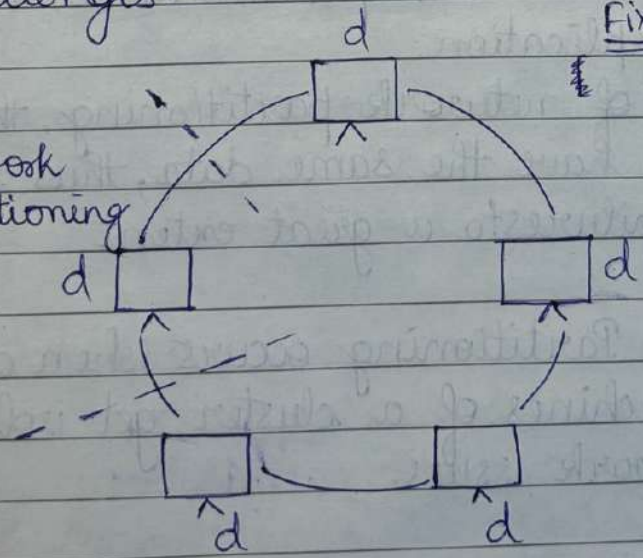
→ This strategy uses multiple machines. When a single machine is unavailable to sustain data and provide storage, we make use of several machines. This strategy is called as Horizontal Scaling.

→ If more load comes up, machines can be added and in case of less load, machines can be taken down from the cluster.

⇒ Distributed Systems help to achieve Horizontal Scaling.

→ Challenges :-

i) Network Partitioning



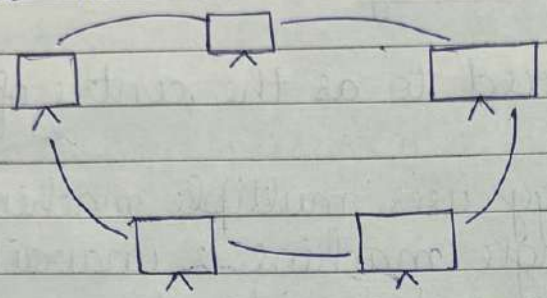
Fix - Replication

ii) Cheap Hardware

iii) Unreliable Communication
↳ network

- Two guided principles of building a Distributed System -

i) Multiple machines → commodity hardware.
ii) These machines are connected over network.



→ Commodity hardware is usually cheap, so probability of failure is high.
→ Network is an unreliable thing.

- Replication -

→ Keep back ups.
In a distributed system, when you don't store data on a single machine but you store the same data on multiple machines, this process is called replication.
→ In case of network partitioning, the machine will still have the same data, thus helping to handle failures to a great extent.

→ Network Partitioning occurs when a machine or some machines of a cluster get isolated due to some network issues.

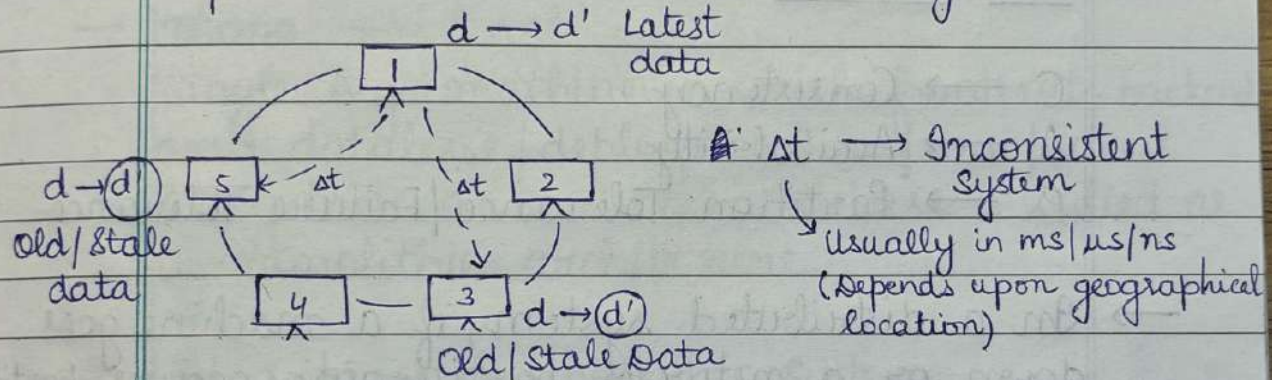
⇒ Replications create a new world of complexities.
E.g: Problem of consistency → Can be understood by learning about CAP Theorem.

- **Consistency –**

When you make multiple queries to a distributed system and everytime it returns same results, then the system is said to be consistent.

→ Data communication occurs via Gossip Protocol.
→ Data before updation is called as Stale Data.

Replication leads to inconsistency.



Distributed Systems are said to be Eventually Consistent because after Δt, the system will eventually become consistent as data changes from d to d'.

E.g: Cassandra is an eventually consistent database.

→ $\Delta t \neq 0$ (impossible but can be close to zero).

- **Availability –**

When you make query to an application and it responds everytime, the application is said to be available.

→ Higher availability, lower downtime.
→ It is usually mentioned in 9's.

can be reduced by fixing bugs faster

- **Reliability** -

Reliability = 1 - Probability of failures

→ If a system has higher reliability, it has a higher availability.
→ If a system has higher availability, it may/may not have a higher reliability.

- **CAP Theorem** -

C → Consistency
A → Availability
P → Partition Tolerance / Failure Tolerance

→ In a distributed system, if a machine goes down or a network partitioning occurs, partition tolerance refers to not losing any data in such situations.

According to CAP Theorem, you cannot have a system to be consistent or available or partition tolerance altogether.
In real-world applications, partition tolerance has to be there always. (Actual Inference of CAP Theorem)
The actual compromise lies between consistency and availability.

→ Social Networking Websites like : Facebook or Instagram are more available than consistent.
→ Applications involving transactions like : IRCTC or banking applications or BookMyShow are

more consistent than available.
↳ (Down for sometime).

E.g: i) Landing Page of Swiggy/Zomato : Available ↑
ii) Booking a place via Zomato : Consistent ↑
iii) Search Page of MakeMyTrip : Available ↑
iv) Messenger/Chat Application : Consistent ↑
↳ order of messages are very important

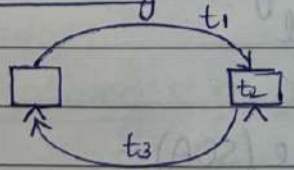
- Monolithic System/Architecture -

→ Mono → Single
Single big machine in which frontend, backend and database/deployment ~~is~~ are placed altogether as a single codebase is called as a monolithic architecture.

⇒ Advantages of Monolithic System -

i) When you're just starting with building an application to keep it easy and cost-efficient
→ Every tech giant started with a monolithic system.
→ When you don't know about your future.
ii) When an application is latency-sensitive.
↳ to avoid network hop that makes applications slow.

* Latency - It is also called as Round Trip Time.



Latency = Total Time = $t_1 + t_2 + t_3$
↳ Calculated in ns/ms/μs/s

iii) Easy and less expensive to secure as all resources are stored at one place.
→ Almost all intelligence agencies use monolithic architecture.

iv) Monolithic applications have fixed boundaries, so functional/integration testing is comparatively easy.
v) Less confused developers, better productivity and simplicity.

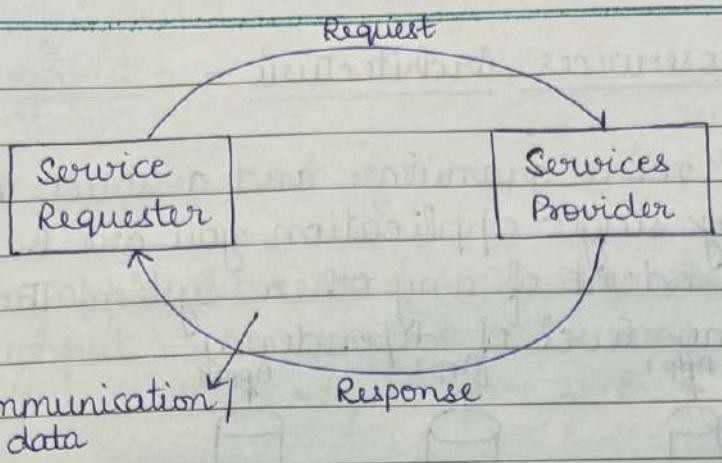
⇒ Challenges of Monolithic Architecture -

i) Scaling can have its own complexities.
Unnecessary scaling will add complexities and lead to wastage of resources.
Selective scaling can't be done.
ii) As the code is present in one single repository, this system is less flexible, it takes more time to production and there might be human conflicts.
iii) The code for frontend, backend and database should be written in same tech stack.
→ Forced to use single tech stack
iv) Deployment time is very high, leading to down-times.
v) Since everything is at a single place, if the system is not highly available, it will lead to single point of failure.

→ Mono Repo (→ Followed by Google) → Single Repo having all codes
└→ Part of Monolithic System
But vice-versa is not equal

• Service Oriented Architecture (SOA) -

There are 2 components:
i) Service Requester
ii) Service Provider



- Any tech stack can be used to communicate between each other.

Classic SOA - Make system as granular as possible.

→ Frontend, Backend and Database need to be made on different tech stacks.

Services talk to each other via communication protocols like : HTTP, socket io

⇒ Advantages of SOA -

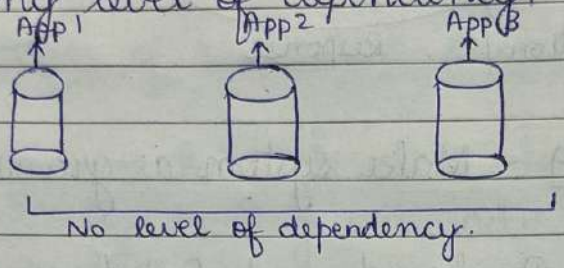
i) Selective Scaling can be performed.
ii) It is flexible to use multiple tech stacks.
iii) It is loosely coupled i.e. more flexible and time to production is lower.
iv) Change in one component doesn't impact other.
v) Deployment time is less.
vi) No single point of failure (SPOF) due to distributed work).

⇒ Challenges of SOA -

i) Latency will go up.
ii) Security will become complex and expensive.
iii) Testing SOA is a complex process.
iv) Confused developers

- **Microservices Architecture –**

It is more granular and a strict version of SOA. Every single application you are building is fully independent of any other system. There should not be any level of dependency.



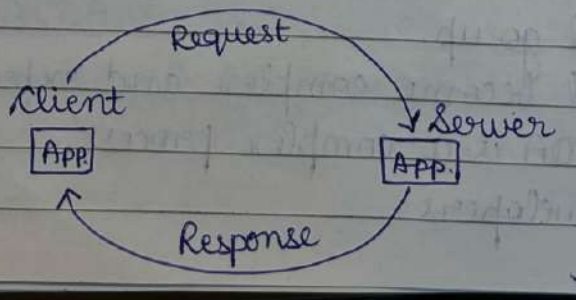
→ This type of architecture is expensive as resource sharing is not allowed.
→ Change in one component doesn't impact others.

- **Client and Server Architecture –**

Client requests for data.
E.g: A browser requesting for data is a client.

Server is a never stopping process running on a computer.
→ It has a port number.
→ It knows how to execute process and respond to a request.

Servers for: Node.js – Express
Python – Flask / nginx
Java – Tomcat



- **Throughput -**

Work done per unit of time is called throughput.

E.g: If an application sends 500 requests in 10 seconds,
throughput = 500 requests / 10 seconds
= 50 requests per second

→ It helps to :
i) Understand performance evaluation
ii) Achieve Capacity planning
iii) Identify bottlenecks

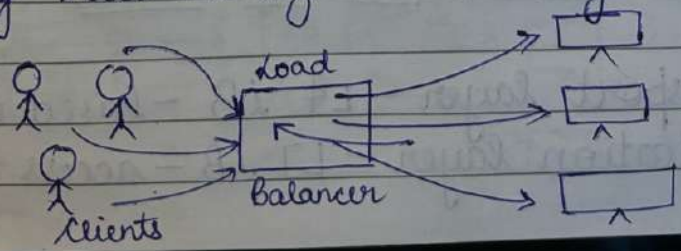
→ **Improvement of Throughput -**

i) Scaling
ii) More optimised code / write code with lower time complexity
iii) Caching / compressions
iv) Database optimisations
v) Network optimisations
vi) Parallel Processing

Throughput is related to latency. If latency is reduced, throughput increases.

- **Load Balancers -**

It balances the load across a machine by ensuring that horizontal scaling is working effectively.



There are 2 types of Load Balancers -

i) Active → Primary
ii) Passive → Secondary
} Leads to Redundancy

By default, every machine is connected with Active Load Balancer. In worst case, Passive one comes into play.

For duplicating a process / system, redundancy is used.
For creating multiple copies of same data, replication is used.
But redundancy and replication are used interchangeably.

- Types of Load Balancer -

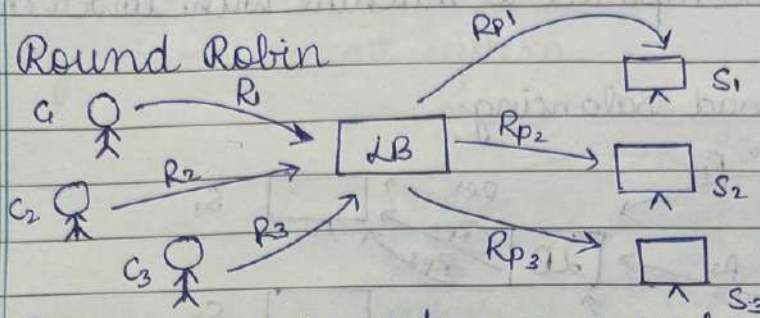
*	Hardware LB	Software LB
i)	Physical load balancer	i) Not a physical LB (virtual)
ii)	Dedicated resources	ii) Shared resources - Virtual Machines (VMs) - Docker images
iii)	Need-specific features - more secure - encryption	iii) Customizable and flexible
iv)	More load	iv) Less load
v)	Expensive	v) Cost-effective
vi)	Used by applications having high load.	vi) Used by applications with less traffic.
E.g:	AWS / GCP, Walmart	E.g: LinkedIn

* Network-based (In OSI model)

i) In transport layer - L4 LB - based on IP Address
ii) In application layer - L7 LB - access to application data

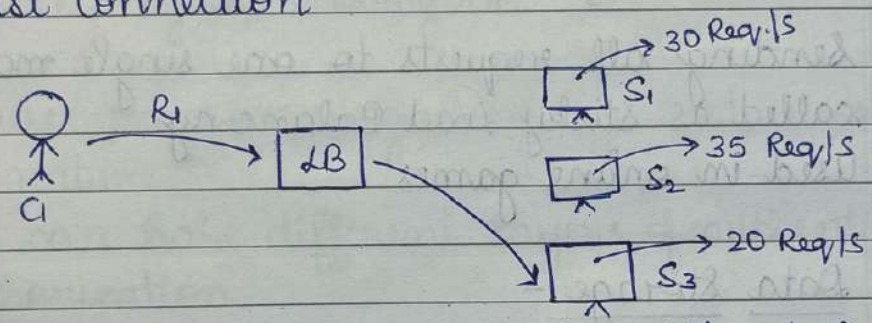
- Algorithms ~~Advantages~~ of LB -

(1) Round Robin



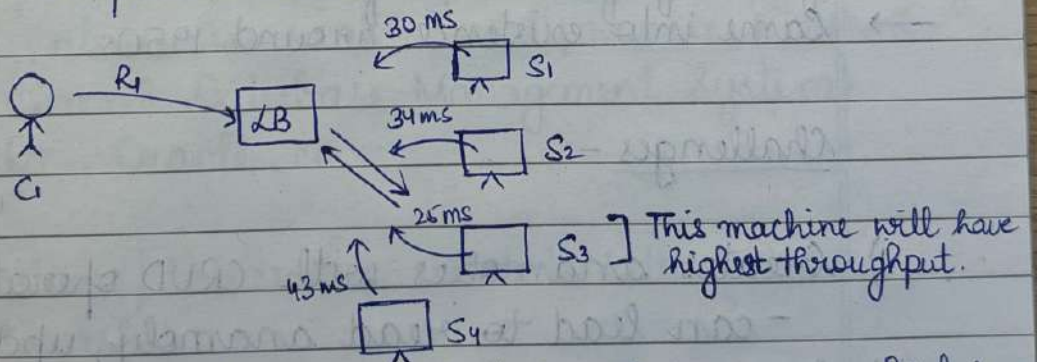
Every server gets an equal chance.

(2) Least Connection



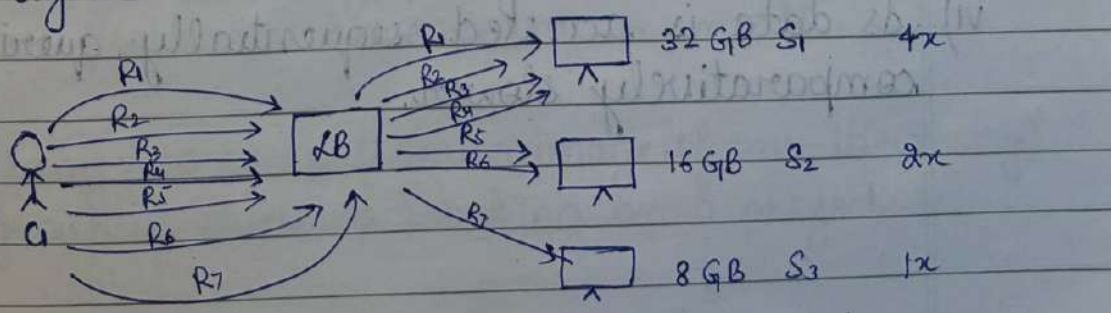
Server with least Requests/s will get the request.

(3) Least Response Time

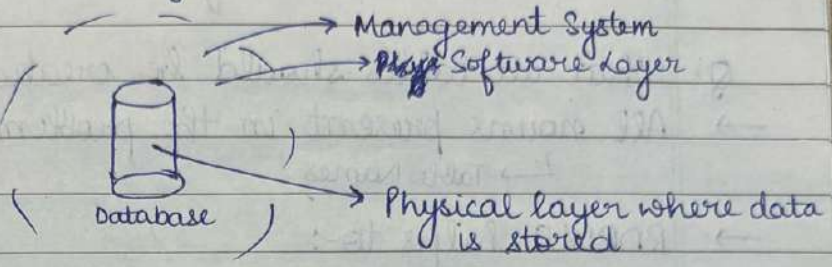


Request will be redirected to the server which has the least response time.

(4) Weighted Round Robin



② Database Management System (DBMS) – At its core, lies the database and it is surrounded by management system.



E.g: Oracle, MySQL, MongoDB

→ It uses programming language that is declarative in nature.
→ You can have different layers of authentication/ authorization.
→ It helps to remove redundancy.

• To represent relationships between different objects and entities in real-world, we use RDBMS (Relational Database Management System).

E.g: MySQL, Oracle, etc.

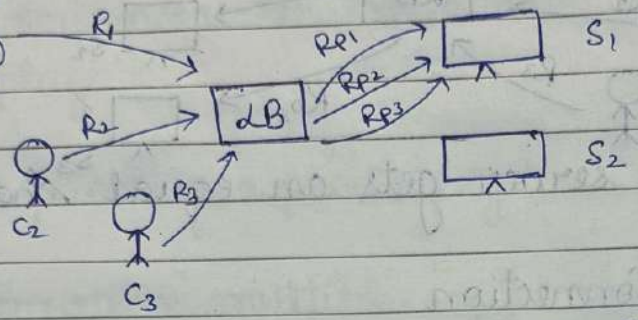
→ Relations / Tables are used to store data.
→ Relationships –
i) One-to-one
E.g: Every person has one Aadhar Card.

ii) One-to-many
E.g: One post can have many comments.

iii) Many-to-one (Mirror image of one-to-many)
E.g: Group of students working on a project

Machines with higher configurations get more no of requests compared to machine with lower configurations.

(5) Sticky Load Balancing



Sending all requests to one single machine is called as Sticky Load Balancing

→ Used in online games

- Data Storage :-

(1) File-Based Storage System - Storing data in the form of files and folders.

→ Came into existence around 1950s.

Challenges :-

i) Creates anamolies with CRUD operations
   - can lead to read anamoly, update anamoly, etc.
ii) Leads to redundancy
iii) Can't be used for scaling
iv) Has the security provided by the Operating System
v) Random access of data is not allowed.
vi) As data is accessed sequentially, queries are comparatively slower.

iv) Many-to-many
E.g.: One teacher can teach multiple students and one student can learn from multiple teachers.

Q. What all tables should be created?
→ All nouns present in the problem statement.
↳ Table Names

→ RDBMS helps to:
i) maintain relationships using foreign keys
ii) structure data
iii) maintain support of transactions
↳ Either all or nothing

→ It follows ACID properties.
A → Atomicity
C → Consistency
I → Isolation
D → Durability

→ RDBMS is a supporter of Normalisation that helps to avoid redundancy.
↳ 1NF
2NF
3NF
BCNF
4NF

Each table should represent single entity.

⇒ Challenges of RDBMS -

i) It makes use of tables which has structured data i.e. number of columns are fixed. RDBMS can't be used when we have a dynamic schema.

* ii) JOIN queries are used to fetch data from multiple tables. These queries are expensive and time-consuming. Network I/O is also very slow. So, it is difficult to scale

RDBMS.

iii) It can't be used for unstructured data or the DB where schema changes frequently.

- NoSQL Databases - Not-only SQL

NoSQL databases are not to replace SQL databases;- they are made to complement SQL databases.

→ It follows a principle called BASE.
i) Basically available
ii) Soft state → State changes on its own
iii) Eventually consistent

→ called as Natively Scalable

→ All NoSQL databases follow the principle of Horizontal Scaling. They are natively made to be built as Distributed Systems.

→ In RDBMS, when SQL queries are run and the state of the database is changed. (Hard State)
In NoSQL (already distributed), there are multiple machines that get updated if one machine's data is updated. (Soft State) These type of systems are eventually consistent.

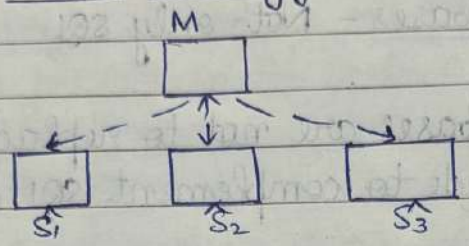
→ NoSQL databases store data in the form of key-value pairs (maps, dictionaries) using tuples, so they are called Key-Value Databases.
↳String ↳Any datatype

E.g: Redis
They are used as caching solutions.

→ NoSQL databases are the propagators of denormalised data i.e. single table has all information.

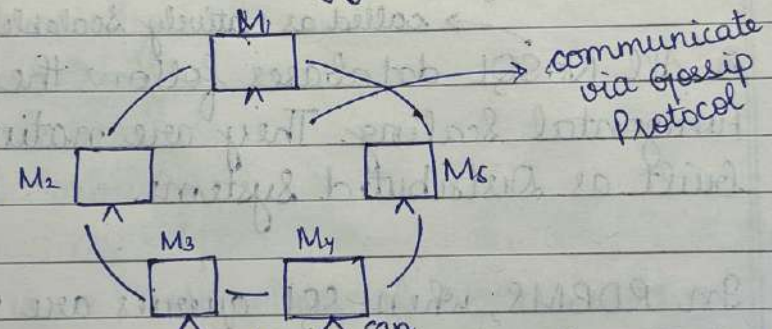
→ Partitioning - Partioning the big table into smaller ones.

Master - Slave Strategy -



All read-write operations occur at Master machine and then they are passed onto slave machines.

Master - Master Strategy -



Read-write operations occur at any machine.

⇒ Challenges of Key-Value Databases -

i) Supports data only in the form of key-value pairs.
ii) Queries are complex (based on keys)

→ Lazy/eager loading - On-demand loading (approach) → for denormalised data
↳ everything will not be present
↓ for normalised data.

- Document Databases -

Data is stored in the form of documents.
↳ JSON document

E.g: MongoDB

→ It tries to bring the best of both the worlds.
From RDBMS :- i) Structure
ii) Transaction
From NoSQL :- i) Dynamic Schema
ii) Scalability

→ Data Storage -

RDBMS	Document DB
- Database	- Database
- Table	- Collection
- Rows	- Document
- Columns	- Fields

• Columnar Database - (Not row-based)
↳ single query returns entire info. about entity

Data is stored in the columns together.
→ Used in the world of analysis for faster aggregation of data. → scale read/write
→ Better compression of data leading to storage of more data in less space

E.g: Apache Cassandra

• Graph Database -
↳ → collection of vertices & edges.

Data which is complex is stored and represented in the form of graphs
E.g: Modern applications like : LinkedIn, Facebook, Google Maps

- **Database Optimization** -

It improves the query speed.

(1.) **Indexing** - Improves the performance of read query.

→ B-trees (Balanced trees) are used for indexing
↳ Helps to store data in sorted order

Disadvantages - i) It takes a lot of space.
ii) It will reduce the write-speed and worsen the query performance if the database is write-heavy. This will lead to restructuring and sorting of B-trees which will take extra time.

Indexing should be done when -
i) database is read-heavy
ii) Identify the columns mostly used in "WHERE" clause.
↳ indexing only on these columns

(2.) **Partitioning / Sharding** - Breaking a larger table into smaller tables is called as partitioning.
↳ based on certain criteria

Partitioning is used for same machine/instance while sharding is used for multiple machines/instances.

Different criterias -
i) **Range-based partitioning** - Based on certain range data is partitioned.
ii) **List-based partitioning** - Based on a list of values, data is partitioned.
iii) **Hash-based partitioning** - Based on the value returned by hash function, data is partitioned.
↳ Most commonly used technique

Note: i) Write good query
ii) Go for indexing
iii) If indexing is not helping out, go for partitioning / sharding (hash-based partitioning).
↳ i) control on the no. of partitions
ii) based on IDs

→ Partitioning is independent of read-write operations

→ Caching avoids redundant reads.

• Types of Processing / Communication-

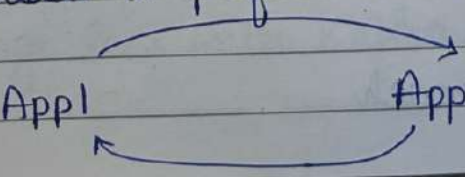
i) Synchronous
ii) Asynchronous

① Synchronous / Blocking - Step-by-step (waiting till step is completed) → Blocks process

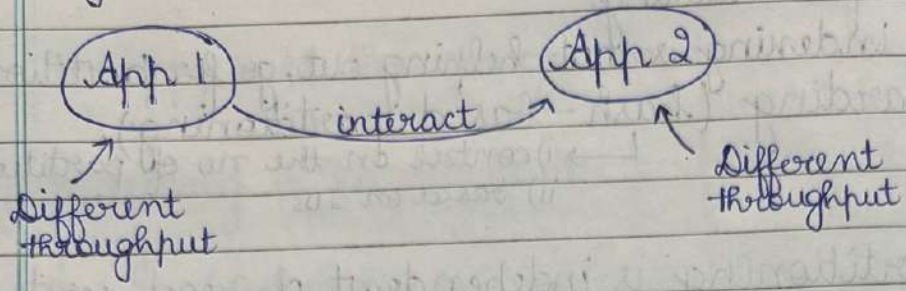
It is used when -
i) Rate of requests sent is compatible with rate of requests being accepted. (Almost equal)
ii) Transaction is being made.
iii) There is an order / sequence involved.
iv) Steps are dependent on previous ones.

⇒ Challenges -

i) It might take a lot of time. (Wait time increases)
→ If the upper limit of number of requests in the waiting queue reaches, the application crashes, leading to cascading failure.
ii) It also leads to performance issues.



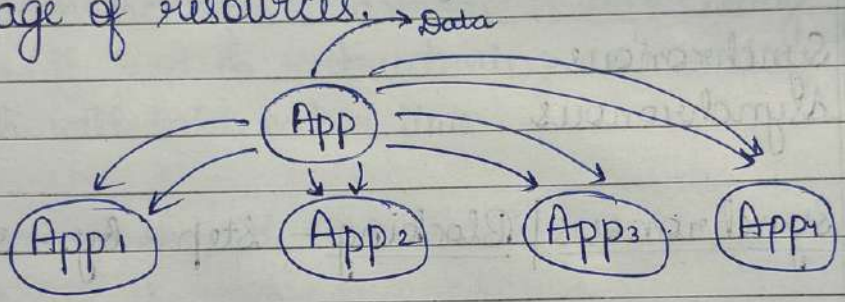
② Asynchronous - Scale the systems well



It is used when -

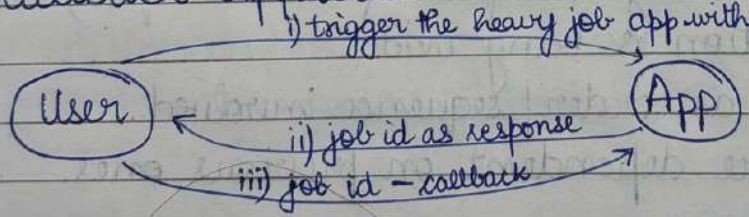
i) Two applications interacting have different throughputs.

ii) Same call has to be made which can lead to wastage of resources.

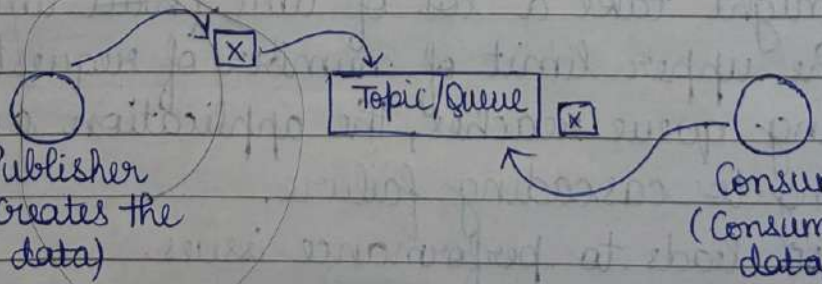


In practice, there are multiple approaches to do asynchronous Processing.

i) Callback Approach



ii) Messaging System Approach / Publisher Subscriber Approach

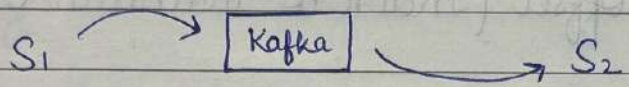


→ Pub/Sub Approach

→ Examples of Messaging Systems -

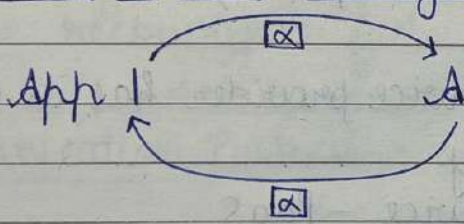
1. Java Tibco
2. AMQ
3. Rabbit MQ
4. SQS
5. Kafka — Market Leader
   → robust distributed messaging system

- Kafka -

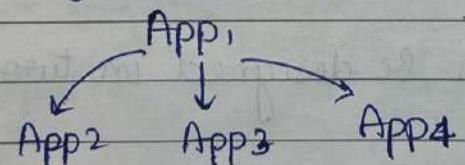


⇒ Types of Messaging :

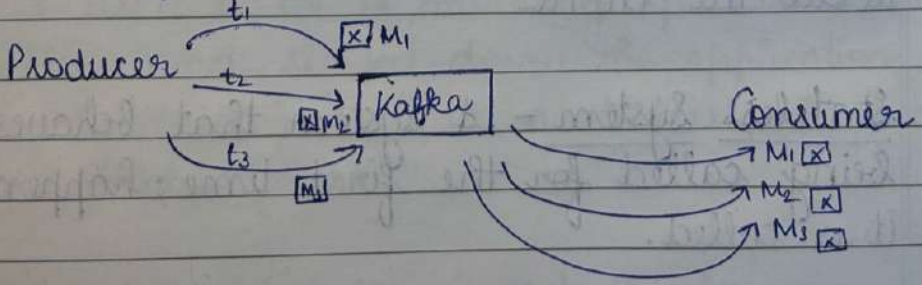
1. P2P (Peer to Peer) Messaging - One to one



2. Broadcasting Messaging - One to many



USP : i) Kafka can do both P2P and broadcasting.
ii) It also guarantees ordered message delivery.



iii) No message loss with Kafka.

[You can increase the duration for which data has to be persisted]
↳ Can also act like a database systems

→ TCP connections & HTTP calls

Producer - Creates messages
Consumer - Reads messages
Topic - Offset (Index to maintain order)

→ Since Kafka is a distributed system, it can pull a large amount of data. So, it is used in Big Data Processing. (Creates Pipelines)

→ Every cloud service provider has its own messaging system. Two ways:
i) Messaging service - PaaS
ii) Kafka cluster - IaaS

- System can be designed in two ways:

i) Stateful
ii) Stateless

i) Stateful System - A system that stores information about the client.

ii) Stateless System - A system that behaves as if it is being called for the first time; happens everytime it is called.

→ If sticky load balancing is being used, then the system is a stateful one.

⇒ Advantages of Stateful Systems -

i) Customized flow
ii) Better user experience

E.g: LinkedIn (login portion) → preserves info. about user/client.

⇒ Challenges of Stateful Systems -

i) Costly
ii) Failure recovery is expensive and slow.
iii) Scaling becomes tougher.

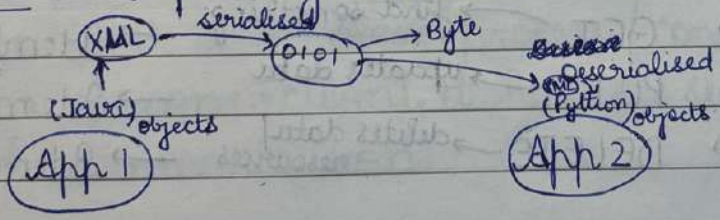
⇒ Advantage of Stateless Systems -

→ can be slow when calls are made to DB. So, caching layer is introduced.

i) Scalable system can be made.
ii) Failure recovery is fast.

• Communication Protocols - → Share | Tranfer data b|w Applications

① SOAP - Simple Object Access Protocol



XML - External Markup Language

→ Companies that used to use SOA, also used SOAP.

→ XML used a lot of memory space which made it bulky and slowed down the application. Stores more metadata and less actual data. SOAP is old school.

② REST - Representational State Transfer

→ More flexible than SOAP. (XML, Text, JSON)

→ Difference between PUT and PATCH - Both of them are used for UPDATION.

Patch is comparatively lighter than Put as partial updation/ the thing that needs to be updated is passed while Put leads to complete replacement.

→ PATCH - Idempotent

- TCP - Transmission Control Protocol

It provides credible information if connection is met or not. If not, it retries.

⇒ Data is broken in smaller packets so that it is easy to transport. Packets are received in right sequence. Damaged packets are retransmitted. Connection is secure. You get a confirmation

Since TCP is a 2-way process (connection, confirmation of connection), it has a slow connection and communication.

Queue window size of TCP needs to be optimally decided or it will lead to congestions.

E.g: www.facebook.com → Internally it makes a TCP connection (Transport Layer Protocol)

- UDP - User Datagram Protocol

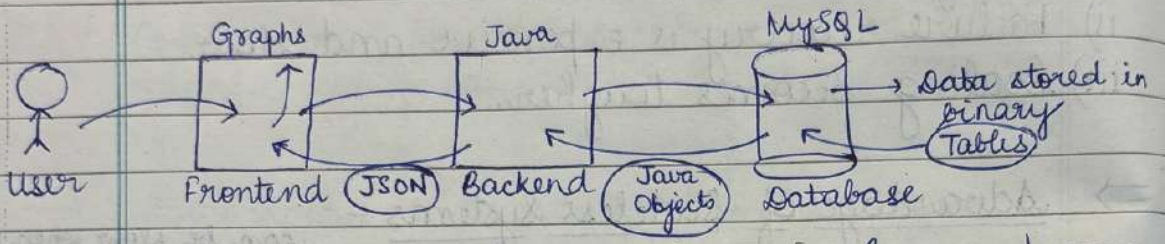
There are no checks to see if the data has been received or not.

→ Prefers to use JSON as it is light and helps in faster conversion

→ Built on top of http (Hyper Text Transfer Protocol) which is an Application Layer Protocol.

→ http is stateless, so it is easy to scale.

→ Web Application -



→ In case of SOAP, everything is focussed around Action/Verb but in case of REST, everything is focussed around resource/data.

→ Since REST is built on top of HTTP, it provides support for CRUD.

C → POST → submit data to server/create resources → Non-idempotent

R → GET → find something → Idempotent

U → PUT → updates data → Idempotent

D → DELETE → deletes data/resources → Idempotent

Idempotent - Either you call for a request once or multiple times, it should lead to same outcome (no side effects).

E.g: Getting OTP is non-idempotent.
↳ Same OTP doesn't come everytime.

Idempotent
PUT, GET, DELETE

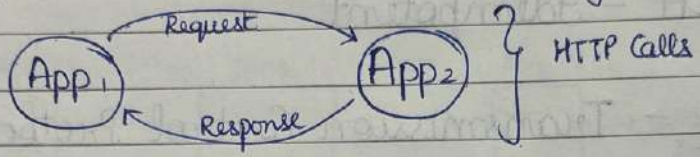
Non-idempotent
POST

No confirmation of delivery, so UDP is fast.

E.g: Video Call System / Zoom / System with Fast Transfer → Broadcast → Light Weight
UDP is used.

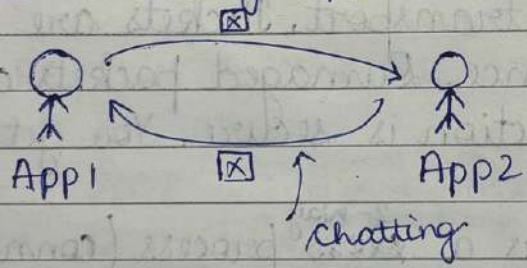
→ Data loss may be there.

- HTTP Connection - Application Layer Protocol



E.g: www.facebook.com
↳ At Application Layer, HTTP is called.
→ At Transport Layer, TCP is called.

E.g: Small Chatting Application -



Short term + for single communication

i) Making HTTP call is not right as TCP will be an expensive connection.
ii) Kafka can be one of the solutions.
iii) More streamlined solution - (Web sockets)
1) Long term connection
2) Two-way communication
↳ Connection tunnel is made

→ Web Sockets are also used to build applications like : Cric Info (where scores get updated live)

→ Web Sockets are used when we need to update content both ways without hard refresh.

→ Guesstimation -

Helps to estimate the scale of the problem and perform back of the hand calculations.

→ Assumption - Scale / Numbers

Total number of websites = 1.2 Billion
60% of websites are active.
$\frac{60}{100} \times 1.2$ Billion = 720 Million

Let's assume that average no. of web pages / website = 100

∴ total no. of webpages / URL = 720 x 100
= 72000 Million
= 72 Billion

So, web crawler has to crawl 72 billion URLs which is a huge number.
Let's assume that we have textual data (string) ignoring images and video files.
Average size of 1 webpage = 100 KB
No. of pages = 72 Billion
∴ total size of web pages to be stored = 72 billion x 100 KB
= 7.2 PBs

After applying some type of compression algorithm that compresses data by 50%, total size = 3.6 PB
This is again a huge amount of data which needs to be stored in a distributed manner.

⇒ When you crawl multiple times, it updates the data stored.

→ Algorithms -

→ some basic website/ seed URL

- Web Crawler - It is an application that scans a website for more links of other websites until you finish with all the websites of the world.

→ Google's search - It contains/ stores the data for all the websites.
It also employs distributed web crawling system.

⊛ Design a Web Crawler -
→ Hit URLs and read content

[1.2 More than Billion Websites] → in world.

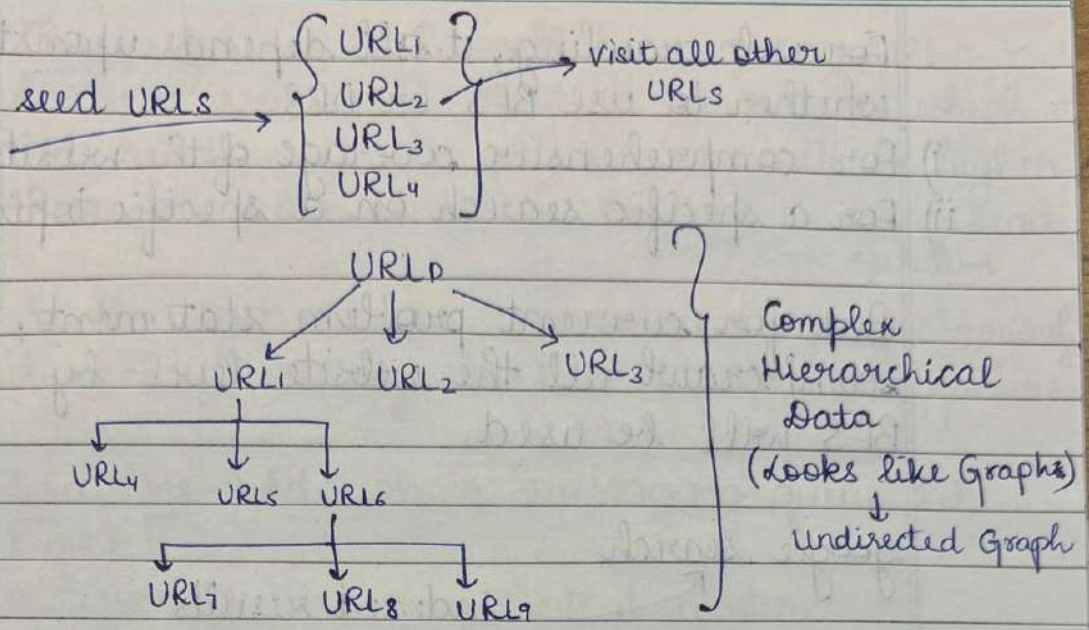
→ Used in :-
i) Search Engines like : Google, Bing, etc.
ii) Copyright Violation Detection / Plagiarism Detection
iii) Data Science for Data Searching
iv) Keyword-based Search
v) Malware Detection

→ Functional Requirements -

i) Scan and store data for all the websites.
ii) Don't store duplicate records.

→ Non-Functional Requirements -

i) Security
ii) Highly scalable
iii) Reliable
iv) Fast
v) Polite on websites → otherwise performance will decrease



This is a problem of Graph Traversal.

Algorithms to traverse Graph:

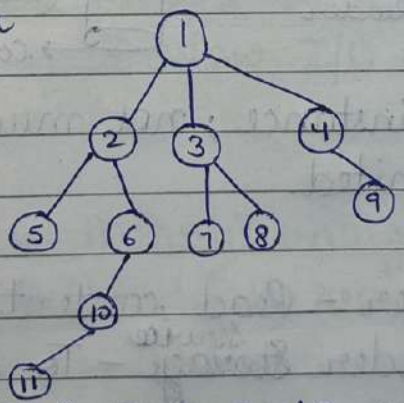
i) Depth First Search (DFS)
ii) Breadth First Search (BFS)

] Both of them return all possibilities eventually

BFS - Gives priority to neighbours.

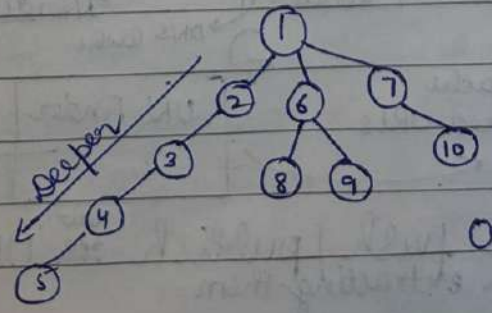
(For closer vicinity)

[Find the shortest Distance]



Order: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

DFS - Goes deeper [Result is at further distance]



Order: [1, 2, 3, 4, 5, 6, 8, 9, 7, 10]

For web crawling, it all depends upon the use case whether to use BFS or DFS.

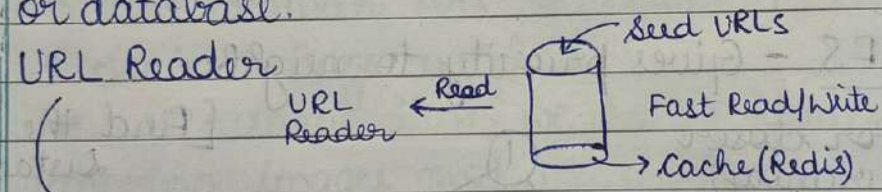
i) For comprehensive coverage of the website - BFS
ii) For a specific search on a specific topic - DFS

But for current problem statement, we need to scan/crawl all the website level-by-level. So, BFS will be used.

Google search ← indexed results ← stored data → Web Crawler (BFS)

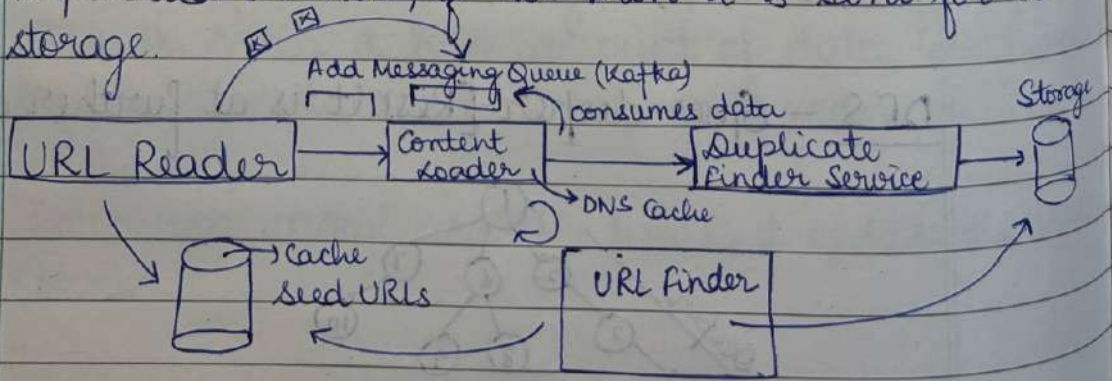
→ Components -

i) Provide URLs that are stored in the form of files or database.
ii) URL Reader



→ Single instance ; not much load as seed URLs are limited.

iii) Content Loader - Read content for each URL
iv) Duplicate Finder ~~Storage~~ Service - To see if content is duplicated or not ; if not then it is sent for stable storage.



v) URL Finder - To push/publish new URLs to cache after extracting them

→ Challenges – i) Cache will have many URLs.
ii) Scaling issues will arise as the rate at which URL reader will be getting URLs will be higher than the rate at which content is being read.
↳ slow operation
iii) As the systems are communicating synchronously, it will ultimately lead to a cascading failure.

→ Solutions – i) Add on a messaging queue like: Kafka
ii) Horizontal scaling of Content Loader
iii) Content Loader service consuming data from Kafka

** Lambda / Serverless – used when you have intermittent operations or low-compute services (when there is time to take rest)

Since data is huge, we need a distributed storage like: HDFS or cloud service provider but these have slow I/O operations to read/write.

→ Strategies :– i) Partitioning
ii) Caching (Redis) – fast read + store hot/latest data

Caching
- In-memory (cleaned after every restart) — E.g: memCache
- Consistent/Stable (Remains stable even after restart) — E.g: Redis

→ Scope of Improvement –
Biggest Player – Content Loader Service → loads data of URL
↳ DNS LookUp → URL → IP Address
↓
Load Cache → Improves speed of system

Daily, users/riders are increasing @ 50000/day
Storage requirement is going by 50000 × 1KB
= 50 MB/day

By next year, current storage = 500 GB
365 days = 365 × 50 = $\frac{365}{\cancel{2}1}^{182.5}$ × 100 MB

= 18250 MB
= 18.25 GB ≈ 18.2 GB

So, storage requirement in 1 year = 500 + 18.2
= 518.2 GB

Driver — Driver's Data
↳ 1KB/driver's data

Total storage needed for all the drivers = $3 \times 10^6 \times 1$
= 3 GB

Daily, drivers are increasing @ 500/day.
Storage requirement is going up by 500 × 1KB
= 500 KB/day = 5 MB/yr

By next year, current storage = 3 GB.
365 days = 365 × 5 MB + 3 GB
= 1825 MB + 3GB
= 1.8 GB + 3 GB
= 4.8 GB

Trips Data Storage —
In 1 day, about 20 million trips take place.
Data for 1 trip = 100 bytes
Storage reqd. for 1 day = $20 \times 10^6 \times 100$ bytes
= 2×10^9 bytes
= 2 GB
For 365 days, storage reqd. = 365 × 2
= 730 GB

⊛ Design a Ride Sharing App like : Uber / Ola -

→ Functional Requirements -

i) Book a cab
ii) ETA of the cab arrival
iii) Cancel the cab
iv) Details of the ongoing cab
} Perspective of User

v) Notification for the new booking
vi) Accept / Reject a new booking
vii) Update location very frequently
viii) Details of the ongoing trip
} Perspective of Driver

→ Non-functional Requirements -

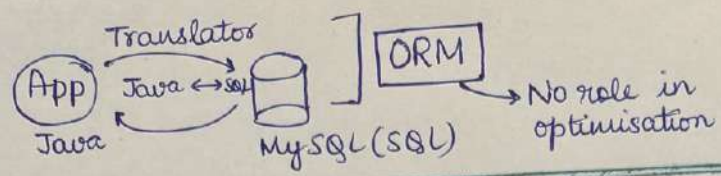
i) Real-time updates
ii) Highly available and eventual consistency
iii) Scalable (especially during peak hours)
iv) Reliable

→ Estimations - → for compute; storage; N/W bandwidth
→ make the infra ready

Assume that total customers / riders = 500 Million
Total no. of drivers = 5 Million
Daily Active → Riders = 20 Million
Drivers = 3 Million

Storage : Rider — Rider's Data
→ id
→ name
→ age, etc.
} 1 KB

Total storage needed for all the riders = 500 Million × 1 KB
$= 500 \times 10^6 \times 1\ kB$
$= 500\ GB$



- **Programming Paradigms** –

(1) **Procedural** – Procedure → Steps

$S_1 \xrightarrow{\text{Boolean decisions taken}} S_2 \longrightarrow S_3 \longrightarrow S_4 \longrightarrow$ Done

E.g: C Language

Used to build: Kernels / Drivers / Statistical or Mathematical calculations

(2) **Object – Oriented Programming** –

Show relationships between objects & entities with their behaviour

E.g: C++, Java, Python, JavaScript (→ trying to be)

Used to build modern applications and solve real-world problems

(3) **Functional** – Function is a first-class citizen.

It can be used as:

i) Variable
ii) Argument
iii) Can be assigned
iv) Can be returned

} Used like any other literal

Everything is immutable.

Principle – Write less, get more

Used to build fast and scalable systems / highly concurrent systems (Scala is one such application)

(4) **Declarative** – Says about what & not how

Trips result in queries or API calls, thus

In 24 hours = 20 Million

$= 20 \times 10^6$

In 1 second $= \frac{20 \times 10^6}{24 \times 60 \times 60} = 230$ trips/s → 100 Bytes

$= 230 \times 100$ bytes/s

$= 23$ KB/s

Driver needs to update his location frequently
Assume that it takes 36 bytes (lat/long details)
For 3 million drivers, storage $= 36 \times 3 \times 10^6$ bytes

$= 108 \times 10^6$ bytes

$= 108$ MB

Also let's assume that the driver updates his location by 4 seconds.

Throughput $= 108$ MB / 4 s

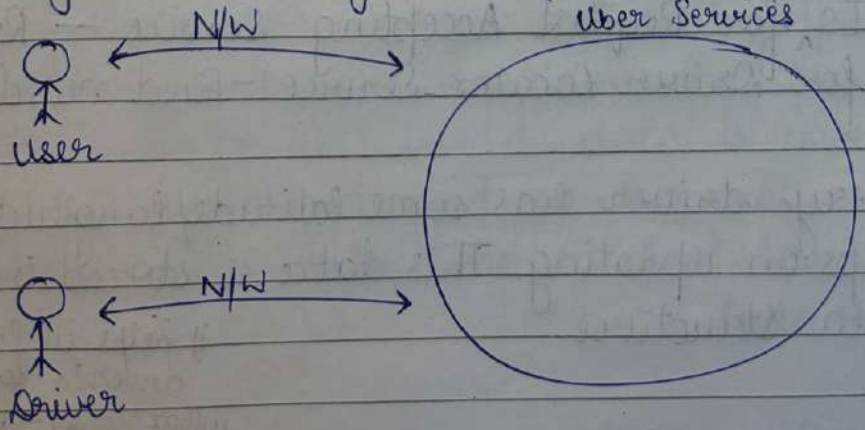
$= 24.5$ MB/s

In 1 day, 20 million rides take place.
If 1 server takes upto 8000 requests,

No. of Servers $= \frac{20 \text{ Million}}{8000}$

$= 2500$ servers

→ High Level Design + Components -



Traffic Data + Map Data → Routing Engines

Deep ETA ←

↓

predicts right ETA

→ source
→ destination
→ time of request

Google Maps – i) Satellite Snapshots
ii) Map of other users
iii) Installed the sensing units
iv) Machine Learning

⊛ Design a Chatting Application like : WhatsApp, Telegram, FB Messenger, Slack, MS Teams, Webex, etc

→ Functional Requirements –

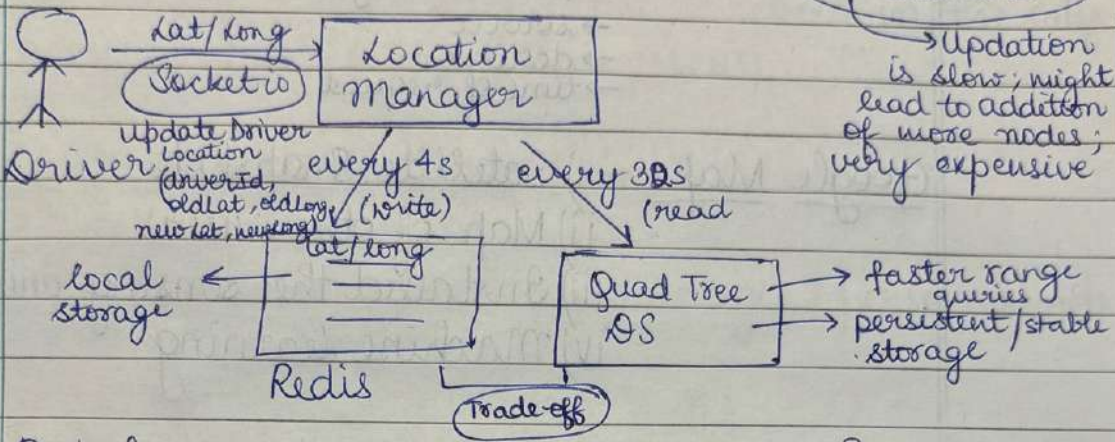
i) 1 : 1 conversation } send + receive messages
ii) Group messaging }
iii) User can see the old message (Store all messages)
iv) Messages should be sent in proper order.
v) Send Messages of type : Text, Images, Videos
vi) Notification about the message
vii) Message shouldn't be lost if recipient is temporarily unavailable.
viii) Message encryption
ix) Delivery and Read Status Checks } Additional Requirements

→ Non-functional Requirements –

i) Real time with good performance (low latency)
ii) Scalable
iii) Reliable + High Availability
iv) Secure (+ Encryption)

After every 4 seconds, driver → Socket.io → Location Manager → Quad Tree Data Structure

So, strategy is :



Updation is slow; might lead to addition of more nodes; very expensive

Database → Huge data (High read/write & i/o) → Scalable

One of the choices : Cassandra

(Recently, Uber has started using Spanner DB). → provides high throughput

API call by driver for Location Manager - updateDriverLocation

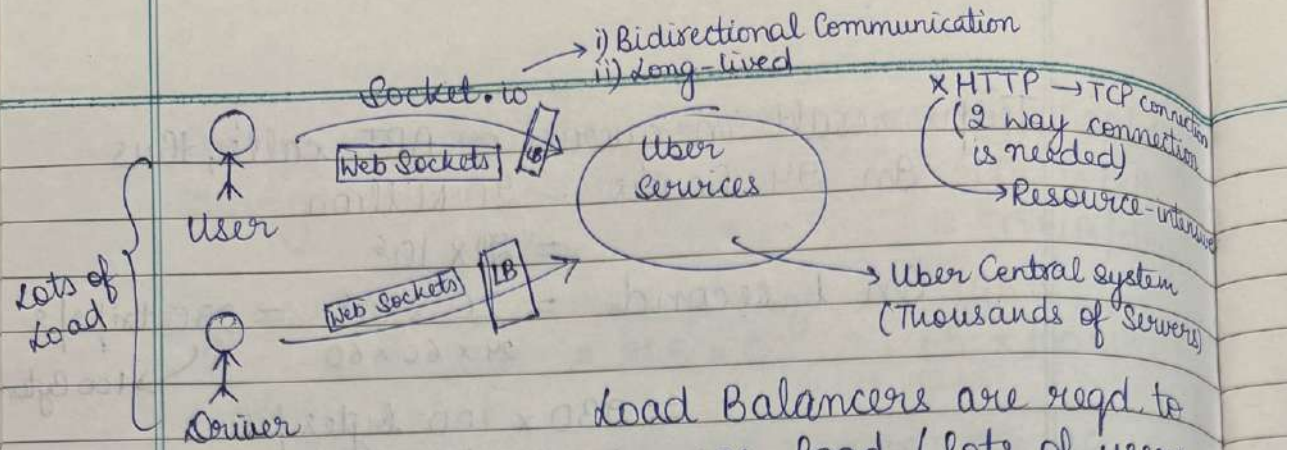
Driver uses Google Maps (complex graph) when a cab is booked.

For ETA, (1) Shortest path to be found
(2) Find time to cover that shortest path

Dijkstra's Algorithm on real-time data (that is huge) is utterly slow. To speed up, pre-calculations are done in background and graph is broken down into smaller partitions. But this data is stale as in real world, data is governed by traffics, accidents and ad-hoc traffic rules.

Modern ETA = Arithmetic Calculations + Multiple Data Informations

ML library to predict right ETA - DeepETA



Load Balancers are reqd. to manage load (lots of users + customers)

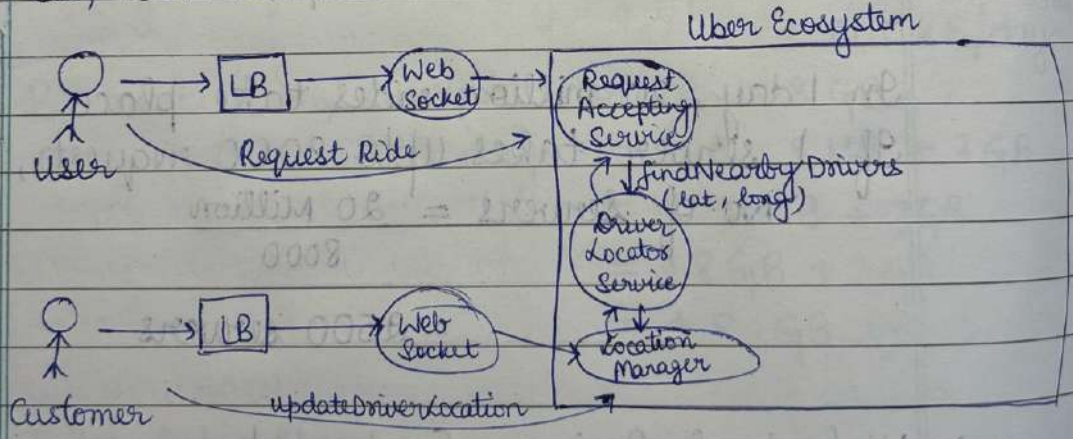
Uber Central System -

i) Driver Locator
ii) Trip Management
iii) Location Updation
iv) Driver Matching
v) ETA Calculations

Logic 1: All services clubbed together i.e. use of Monolithic Architecture

Ideally, should be separated due to different functionalities

⇒ So, SOA should be used.



APIs (to be called by user)
for Request Accepting Service - Request Ride
for Driver Locator Service - Find nearby drivers

Every driver has some latitude/longitude which keeps on updating. This data is stored in Quad Tree Data Structure.

i) helps in fast search of any driver's location
ii) (lat/long) - driver neighbours or nearby location

→ Scale of the system –

Assume that 2 billion people are using it, out of which 500 million people come in the category of DAU (Daily Active Users).

→ Infrastructure Estimations –

Let's assume that on average, 1 person sends 100 messages.
Size of 1 message = 100 bytes
Total data for storing messages = 500 million $\times_{\times 100}^{\times 100}$
$= 500 \times 10^6 \times 10^4$
$= 10^{12} \times 5$ bytes $= 5$ TB
Storage for messages in a year = 365×5
$= 1825$ TB
≈ 1.8 PB

Bandwidth Estimation – No. of reads ≈ No. of writes
Data generated in 24 hrs = 5 TB
Data generated in 1 s $= \frac{5 \text{ TB}}{24 \times 60 \times 60} \approx 460$ MB/s

No. of servers/machines required –
No. of machines $= \frac{\text{Total no. of connections}}{\text{No. of connections/saver}}$

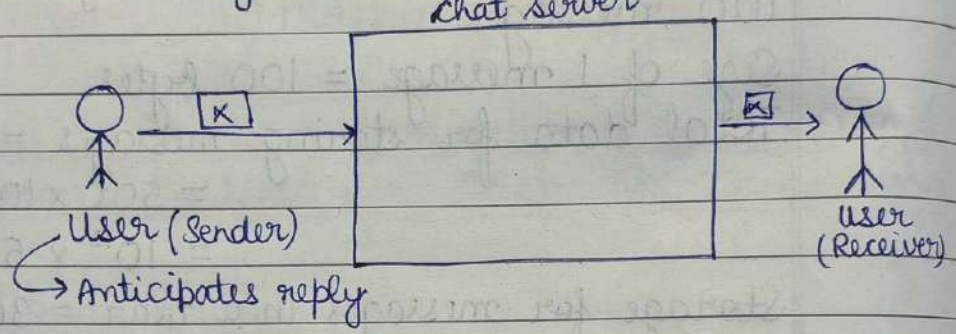
DAU = 5 Million
Assume that 1 connection takes 100 Bytes of data.
1 server has RAM of about 16 GB.
Out of 16 GB, 10 GB is available to make connection
No. of connections $= \frac{10 \times 10^9}{100} = 10^8$

No. of servers $= \frac{50 \times 10^6}{10^8} = \frac{50}{100} < 1$ } Single Machine

Note: Chatting is not a CPU-intensive task.
Server — i) accepts the message
ii) relays to another

Multiple Machines are used to : i) easily scale
ii) Higher Availability
iii) locality advantage → to achieve low latency

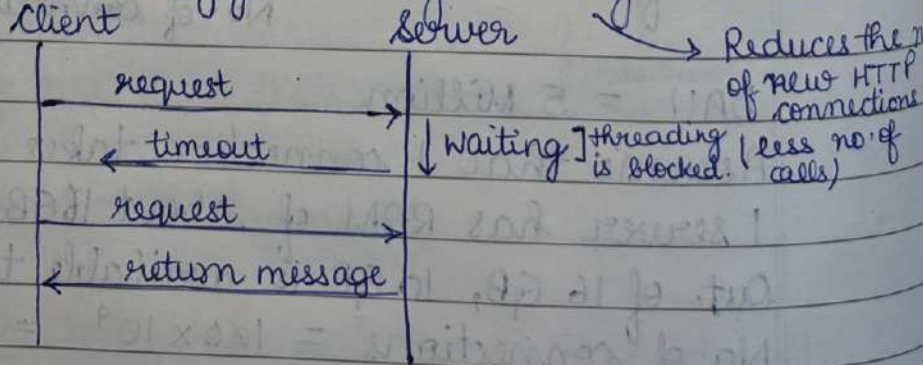
→ High Level Design & Components + Communication -



HTTP connection can't be made as it is expensive to create such multiple connections and it is short-lived too.

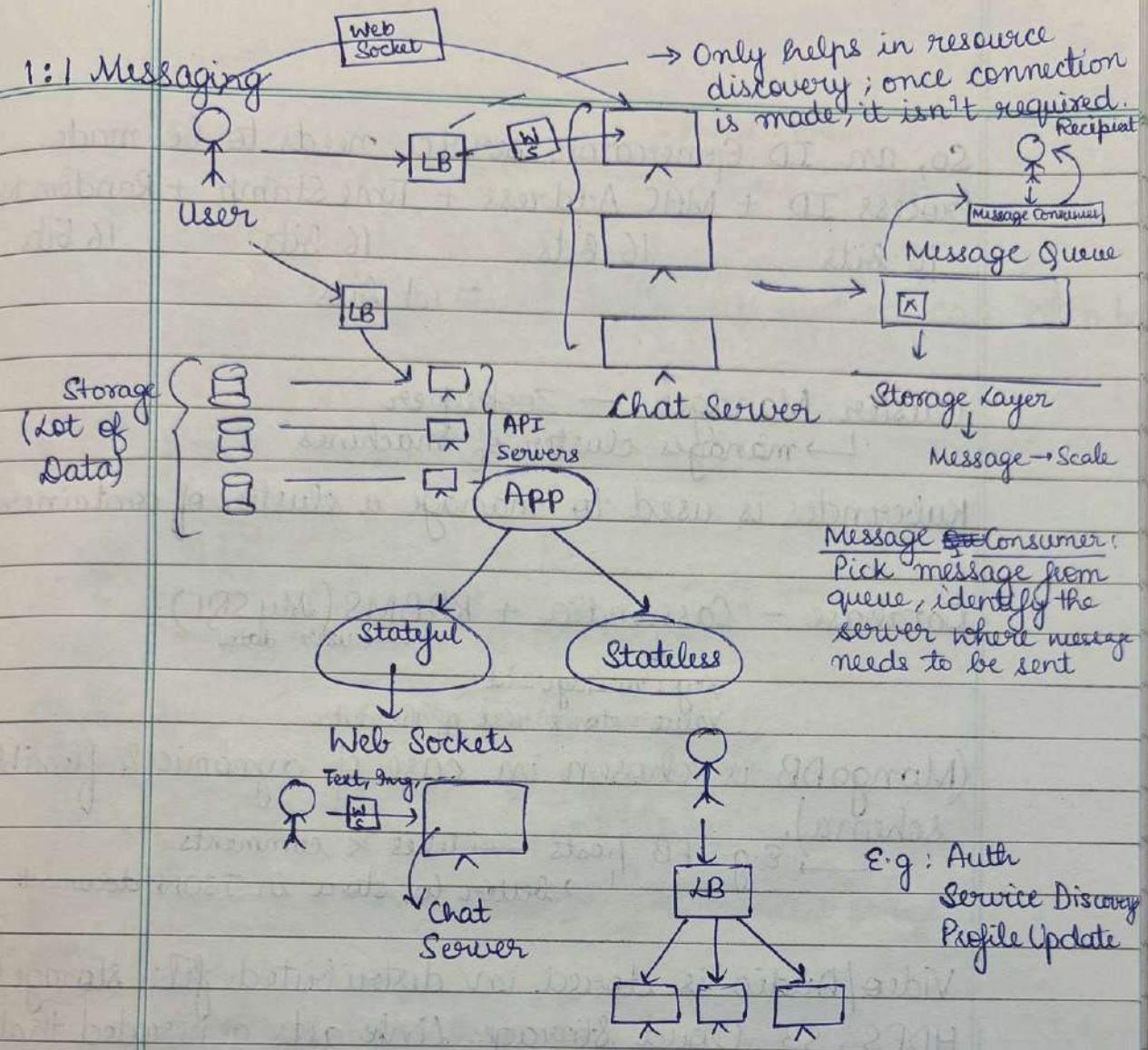
Note: Polling - The strategy of making multiple HTTP calls checking if server has data and returing the data from the server is called Polling.

Optimised strategy - Long Polling → not good for multi-tasking



More Streamlined Approach - Web-sockets
Help in 2-way connection + Stateful communication
↳ socket.io

1:1 Messaging



Message Consumer:
Pick message from queue, identify the server where message needs to be sent

E.g: Auth
Service Discovery
Profile Update

Database – i) Scalable
ii) Stores all the messages
iii) Historical messages → that can be loaded later
iv) People watch the latest messages

Message { → message-ID → unique identifier
i) has to be unique
ii) sorted based on time/chronologically sorted
timestamp
from
to
content

i) unique
ii) chronologically ordered

Creating unique ID is itself a problem statement.
Usually, RDBMS (MySQL) can generate IDs in sequence. But since data is huge, NoSQL is used and above isn't valid in case of NoSQL.

So, an ID Generator Service needs to be made.
Process ID + MAC Address + Time Stamp + Random No.
16 bits 16 bits 16 bits 16 bits
= 64 bits

Cluster Manager — Zookeeper
↳ manages cluster of machines
Kubernetes is used to manage a cluster of containers

Database - Cassandra + RDBMS (MySQL)
↓ ↳ user-data
Key: message_id
Value: stores rest of the data
(MongoDB is chosen in case of dynamic & flexible schema).
↳ E.g: FB posts → likes & comments
↳ better to store in JSON/document

Video/Audio is stored in distributed file storage/ HDFS, S3 Cloud Storage. Link gets generated that is stored in regular database.

Textual data ⟶ Transactional database
↳ OLTP

Video needs to be loaded from HDFS.

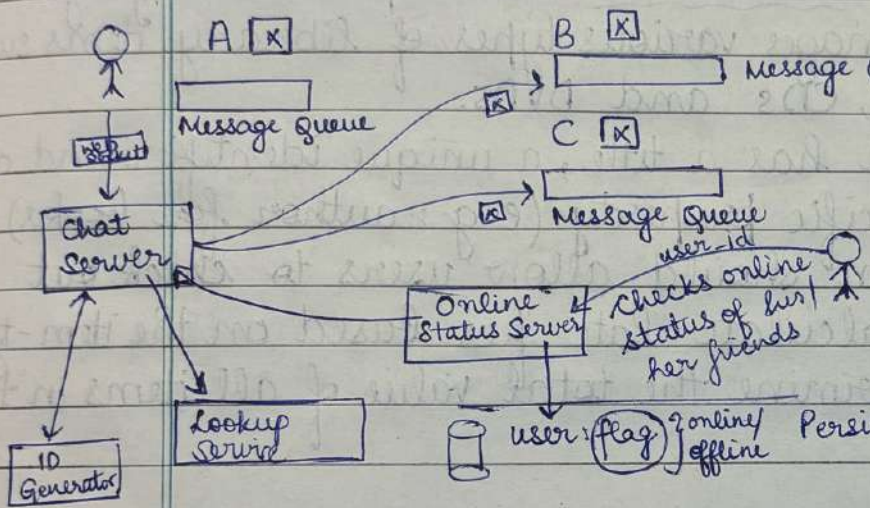
Challenges for message consumer -
i) consumes from Kafka ⟶ single topic (tough)
ii) finds which chat server to push message

If recipient is offline, message is sent in a waiting state.

.. Message sync-up - → Phone – online
→ Tablet
→ Laptop – on/off

Add a layer of caching.
Cloud storage and back-up system can also be used.

Group Chat -



No. of messages published ∝ Group size

G_1 G_2 G_3 G_4 G_5

↓ unique id

Constant time lookup. ← Map -
(Key) → Group Id
(Value) → List of user Ids.

Ordering of messages
↳ Taken care by unique message id.
↳ Chronologically sorted based on time

Online/Offline Status of Users - Online status server checks the active socket connections. User Id will be retrieved and a flag is set at the persistent storage. ↳ (used for boolean outcome)

Note - Heartbeat test is used to see if server is alive or not. In a distributed system, a health server is used to keep a check on the health of other systems. It might give stale information.