SYSTEM DESIGN TOPICS PART 2

Read about **Sharding , Reverse proxy and memcached** today !

Sharding is basically a hierarchical way to index databases.
One problem is that you have to split the database somehow. What do you split on?
You only shard shards when the shard grows too big.
When shard fails you use the master/slave architecture. Writes always go to the master and reads are distributed across the slaves. When the master fails, one of the slaves becomes master.

Sharding is the horizontal partitioning of data according to a shard key. This shard key determines which database the entry to be persisted is sent to. And reverse proxy is just used for that ,a server that sits in front of one or more backend servers, and handles incoming client requests on their behalf. The reverse proxy checks the **shard key**, figures out which server holds the data, and forwards the request there.
Users never directly see or interact with the shards.

And Memcached is,
A high-speed in-memory cache system used to store frequently accessed data temporarily to reduce database load and increase speed. It Reduces load on shards, improves speed. So instead of going to the database/shard every time, the system serves from memory — much faster and reduces load on shards.

Better relate all this with consistent hashing and load balancing !

1. **Sharding** breaks a big database into smaller parts to manage it better.
2. **Shard key** decides which data goes into which shard.
3. **Consistent Hashing** spreads data evenly to avoid overload.(mainly fixed degree k around ye spread krta hai)
4. **Reverse Proxy** sends user requests to the right shard/server.(direct m risk h na kyoki)
5. **Memcached** stores frequently used data in memory for faster access.
6. **Master/Slave model** keeps data safe and available even if one fails.
7. **Hierarchical Sharding** means dividing data in multiple smart levels.(shards k andr shards banana)

Hmm and also ,Shards ki size equal nahi hoti hai but they are rather balanced based on query loads , writes and reads , and storage pressure wagera.

EOT

Master-Slave , Distributed Consensus , Data replication !

Ek distributed system mein, **master-slave architecture** ek control mechanism hota hai jahan ek central **master server** saare decisions aur commands issue karta hai, aur multiple **slave servers** un commands ko follow karte hain, jaise data ko read karna, ya replicate karna, jisse ek centralized aur ordered system maintain hota hai. Is architecture ka ek popular type hai **single-master multiple-slave**, jahan sirf master write karta hai aur slaves sirf read, jabki doosra type hota hai **multi-master**, jahan multiple servers write kar sakte hain, lekin conflict resolution zaroori ho jata hai.

Ab jab ye servers alag-alag jagah pe ho aur ek hi update ko accept karna ho, tab aata hai **distributed consensus**, jiska kaam hai ye ensure karna ki system ke mostly servers kisi bhi ek update ya transaction par agree karein — bina iske data consistent nahi rahega. Iske liye algorithms use hote hain jaise **Paxos**, jo thoda complex hota hai but reliable; **Raft**, jo simple aur easy-to-implement algorithm hai jisme ek **leader** elect hota hai, jo proposal deta hai aur majority ke accept karne pe update commit hota hai; aur **Viewstamped Replication**, jo backup leaders use karta hai taaki agar current leader fail ho jaye to system rukhe nahi. Ye consensus tabhi valid hota hai jab **majority nodes**, jaise agar 10 servers hain to kam se kam 6, ek update ko accept kar lein. Jo baaki servers thoda delay se respond karte hain, wo baad mein update le lete hain — ise hum **eventual consistency** kehte hain.

Jab update successful ho jata hai, tab system **data replication** karta hai, jiska matlab hai ki us update ki copies multiple servers pe store hoti hain, taaki agar koi server down ho jaye to data fir bhi accessible rahe. **Replication ke types** mein hota hai **synchronous replication**, jisme har server ek saath update hota hai — safe but slow; aur **asynchronous replication**, jisme pehle master update hota hai aur baaki later, jo fast hota hai par data loss ka risk bhi hota hai. Ek aur type hota hai **quorum-based replication**, jisme read ya write operation tabhi hota hai jab ek certain number of servers (jaise 3 out of 5) se approval milta hai, jisse performance aur consistency ka balance maintain hota hai.

In teeno concepts — master-slave control for coordination, distributed consensus for agreement, aur data replication for fault-tolerance — ka combined use Google, Amazon jaise large-scale systems ya blockchains mein hota hai, jahan data har waqt reliable, consistent aur available hona zaroori hota hai.

EXAMPLE :

Socho ki tum ek **online shopping platform** Amazon ki tarah bana rahi ho jahan poore India mein servers lage hain — Delhi, Mumbai, Bangalore, Hyderabad jaise cities mein. Ab jab koi user Indore se mobile order karta hai, to sabse pehle ye request **Delhi ke master server** ke paas jaati hai, jo order ko process karta hai aur apne **slave servers** (jaise Mumbai aur Bangalore) ko instruction deta hai ki "ye order update karo." Yaha pe master-slave architecture kaam karta hai jahan ek master control mein hai aur baaki uska follow karte hain. Fir system ensure karta hai ki **Distributed Consensus** ho — matlab baaki servers se confirm karta hai ki "kya sabne ye order update receive kiya?" Agar 6 out of 8 servers bolte hain ki "haan update ho gaya," to system decide karta hai ki ye transaction final hai, aur 2 servers jo network slow hone ki wajah se delay mein the, wo baad mein update le lenge — isse data consistency bani rahti hai. Ab jab order successful ho gaya, to ye update **replicate** kiya jaata hai across multiple servers, jisse agar kal ko Delhi ka main server crash ho jaye to bhi Mumbai ya Hyderabad ke server se user ka order history access ho sake. Ye replication **synchronous bhi ho sakti hai** (real-time sab servers pe copy ban jaaye) ya **asynchronous** (baad mein copy ho). Is tarah se tumhara pura system centralized control ke saath, sab servers ke agreement ke through consistent rehta hai, aur data ki multiple copies hone ki wajah se kabhi fail nahi hota — jo kisi bhi large-scale, real-time, high-availability system ke liye bohot zaroori hota hai.

Question : Agar Mumbai ke orders sirf Mumbai ka master handle kare, aur Delhi ka master sirf Delhi ke orders — to wo kyun nahi hota? Aur agar ho sakta hai, to kaise?

Yahaa aayaa sharding

Socho ek company hai jiska business poore India mein hai, par usne decide kiya:

- Mumbai ke users ke orders sirf **Mumbai master server** handle karega
- Delhi ke users ka data sirf **Delhi master server** ke paas rahega
- Bangalore ke liye Bangalore, Hyderabad ke liye Hyderabad

Matlab:

> **Ek server master hai, lekin sirf apne area (shard/partition) ke liye.**
> Ye sab alag-alag **independent clusters** jaise kaam karte hain.

Phir dikkat kya ????
OVERLAPPING ki dikkat smjhrhe ho
User A Mumbai mein hai, but Delhi mein jaake order place karta hai.
Tab kis master ko control lena chahiye?

Split Brain Problem

Tumhare distributed system mein **do ya zyada servers** hain, jo **normally ek master-slave** architecture mein kaam karte hain.
Lekin kisi network failure ya communication issue ke wajah se, **do alag servers ek dusre ko dekh nahi pa rahe** — aur **dono sochne lagte hain ki main master hoon.**

Iska Solution:

1. **Consensus Algorithms (like Raft or Paxos):**
   Ye ensure karte hain ki **ek hi leader** elected ho at a time

2. **Quorum-based Voting:**
   Update tabhi valid hoga jab majority nodes agree karein

3. **Fencing Tokens / Epoch Numbers:**
   Har leader ke paas ek version number hota hai — jo purana hoga, wo ignore hoga

4. **Network Partition Detection:**
   Agar system detect kare ki network split hua hai, to ek side ko passive bana diya jaata hai (read-only)

!!! Now we will go through leader electing algorithm lalala 👍

1. Raft Algorithm

- **Type:** Consensus protocol
- **How it works:** All nodes start as followers. If a follower doesn't hear from a leader within a timeout, it becomes a candidate and requests votes from other nodes. If it gets a majority, it becomes the leader. The leader sends regular heartbeats to maintain leadership.
- **Use cases:** Kubernetes, etcd, Consul
- **Strengths:** Simpler and more understandable than Paxos; ensures strong consistency

2. Paxos Algorithm

- **Type:** Consensus protocol
- **How it works:** Nodes propose values to each other. A proposer must get a majority of nodes to accept a proposal to consider it committed. It doesn't have a fixed leader but elects one temporarily for decisions.
- **Use cases:** Google Chubby, Microsoft Azure

- **Strengths:** Very fault-tolerant; can handle message loss and duplication
- **Weaknesses:** Complex to implement and understand

(values kuchh bhi ho skti hai koi database update , client request , new leader yaa log entry etc)

3. Viewstamped Replication (VSR)

- **Type:** Leader-based replication
- **How it works:** The system is divided into views. Each view has one primary (leader) and backups. If the leader fails, a view change occurs to elect a new leader and continue operations.
- **Use cases:** Replicated databases, VMWare systems
- **Strengths:** Structured and more readable than Paxos; fault-tolerant

(View = A period of time when one leader is in charge, Jab tak leader sahi kaam kar raha hai, **view change nahi hoti**. Agar leader fail ho jaye ya delay ho jaayetb
“view badalte hain” aur ek **nayi view** start hoti hai.)

4. Bully Algorithm

- **Type:** Election-based algorithm
- **How it works:** Each node has a unique ID. If a node notices the leader has failed, it sends an election message to nodes with higher IDs. The highest ID node becomes the leader.
- **Use cases:** Small-scale distributed systems
- **Strengths:** Simple and deterministic
- **Weaknesses:** Not ideal for large-scale or failure-prone environments

(ye id manually assign kri jaa skti hai yaa , ip address ho skti ahi yaa system start hone pr uuid generate ho skti hai , MAC address bhi ho skta)

5. Chang and Roberts Algorithm

- **Type:** Ring-based election
- **How it works:** Nodes are organized in a logical ring. A node starts an election by sending its ID around the ring. Each node forwards the higher ID. The highest ID is elected as the leader.
- **Strengths:** Efficient for ring topologies
- **Weaknesses:** Assumes a fixed ring structure

(**Physical ring nahi**, balki **logical ring** hoti hai — bas message-passing ka order ring jaisa hota hai.Sab nodes agar randomly kisi ko bhi message bhej den → system becomes chaotic.Ring allow karta hai ki **election message ek fixed route follow kare**
Har banda apni value agle ko deta hai, agla check karta hai aur best value aage pass karta hai.Jab sab ghoom ke value wapas pahuchti hai, original banda bolta hai:“Aha! Sabse bada main hoon. I’m the leader!”)

6. LCR (Le Lann-Chang-Roberts) Algorithm

- **Type:** Simplified ring-based
- **How it works:** Similar to Chang and Roberts. Each node sends its ID clockwise. When a node receives its own ID back, it becomes the leader.
- **Strengths:** Very simple
- **Weaknesses:** Not robust in dynamic systems

(Each node **only forwards** the ID if it's greater than its own.lcr m saare node election start kr skte hai pr uppr m koi ek hi krta hai aur ye bss khud ki id se compare krta agr khud ki se bada hota tohi forward krta hai aage , ye old version h little chaotic)

7. FloodMax Algorithm

- **Type:** Broadcast-based
- **How it works:** Each node floods its ID to all neighbors. In each round, nodes keep the maximum ID seen. After log(n) rounds, the node with the highest ID is elected as leader.
- **Use cases:** Sensor networks, arbitrary topologies
- **Strengths:** Works without knowing the full network
- **Weaknesses:** High message overhead

(isme topology structured nahi hoti hai, Har node apna **unique ID** rakhta hai.
In each round, node sends **maximum ID it knows** to **all its neighbors**.
Neighbors compare that ID with their current maximum, and **update if higher**.)

8. ZAB (ZooKeeper Atomic Broadcast)

- **Type:** Total order broadcast
- **How it works:** ZooKeeper uses the ZAB protocol to elect a leader who coordinates updates and broadcasts them in total order. If leader fails, followers elect a new one via voting.
- **Use cases:** Apache ZooKeeper
- **Strengths:** Ensures total order and consistency in coordination services

(Ye ek consensus + leader election + replication protocol hai, specially designed for Apache ZooKeeper (ye big data and hadoop se relate krna mainly for coordination hive hbase wagera jo hota hai aur ek pura system hai iska)**ZAB is ZooKeeper ka leader+data controller** — jo make sure karta hai ki leader election bhi ho, sabhi followers ke paas same data ho, aur system kabhi confuse na ho.

9. Multi-Raft / RAFT++

- **Type:** Raft variants
- **How it works:** These are improvements on the base Raft protocol to allow multiple independent Raft groups or better performance under partitioning.
- **Use cases:** High-scale multi-cluster systems
- **Strengths:** Distributed load, better scalability

(raft jesa hi hai bss ye multi raft grp banata hai jo parallelly kaam kr rhe hote hai , kaam aata hai jb bht saari requests ho jaaye)

EOT

Optimizing Writes in DB , Compaction , LSM tree

Imagine Twitter.
If thousands of users tweet at once, and the database tries to **write every tweet instantly**, it will slow down, or **break under pressure**.

Condense Multiple Queries into One

Combine many insert/update operations into a **single batch**.
Yes, it needs more memory, but I/O becomes way less → **faster overall**.(manje kam i/o operations used)

But here's the REAL question:

Which data structure does the fastest write?

Linked List

- Writing (inserting at end) = `0(1)`
- Just add at the end → very fast.

This is the core idea behind something called a **log**.

What is a log?

> A log is just an **append-only list**
> Like a **journal/diary** – you only write new entries at the end.

Fast write
Very slow read (because you have to search line by line)

So…what's the solution?

We want:

Fast writes (like a log/linked list)
Fast reads (like a sorted array/tree)

But how can we mix these? (btw , b-tree k read aur writes ki time complexity log(n) h)
Collect the writes in memory → like a linked list/log

1. **Sort them before saving to database** → like a sorted array

2. Then **persist** them all in one go
→ Batch + Sorted = Good Write + Good Read

So the trick is:
Sort the data before persisting it.

But sorting large data is costly?

Yes. So we optimize that too:

Break it into small chunks
Sort small batches
Merge only when needed

Pr pr pr humesha merge nahi kr skte hai unnecessarily right ,,,,
So we use smart merging called as COMPACTION

Iske baare m padhte hai hum thodi derr se uske pehle ….

Abhi tk humne bacth krna aur log dekha
Breifing : Instead of writing one row at a time, we send a group (batch) of them together.
And for log, Just write and don't worry about sorting or checking.
And magic was to combine them that's we call LSM trees , will see
Other than that we have for writes issssssss

Async Writes (Write in the background)

- Imagine someone fills a feedback form, and instead of saving it immediately, we **drop it in a box**, and someone saves it later.

This is what companies do:
User action → Goes into a **queue** → Saved in DB **later**

Used in systems like: Kafka and RabbitMQ

How to Optimize **Reads** (Faster Viewing of Data)

1. Indexing (Like bookmarks) , They keep indexes on important columns (like username or email) to **find things faster**.

2. Caching (Remembering answers) , We **save the result in memory** (like Redis or Memcached). If someone asks again → just give from memory.

3. Read Replicas (Twins for help) , In databases, we can create **read-only copies** (replicas). All the reads go there → main database is free to handle writes.

4. Materialized Views (Pre-written answers) , They store **results of complex queries** as ready-made tables.

5. Denormalization (Keeping all info together) , to avoid time-consuming joins, we just **store related info together**.

Ab jo thoda gaur krne waali baat uppr kri thi compaction aur lsm , basic to we got it right there , let's see it again

Pehle batadu lsm ki dawa hai compaction heheh

LSM Tree (Log-Structured Merge Tree)

Write jaldi karo (log ki tarah), baad mein sort karo (tree ki tarah)

Step 1: MemTable

MemTable ek temporary **in-memory table** hai jo sabse pehle likha jaata hai jab user kuch write karta hai (jaise post, comment, data update).

Important Point:
Ye **sorted hoti hai** (jaise TreeMap in Java or Python's `SortedDict`).

Why Sorted?

- Taaki **jab isse disk par likhen**, to data already sorted ho.
- Iska fayda ye hota hai ki SSTable banate waqt **sorting mein time waste nahi hota**.

Step 2: MemTable → SSTable

Jab MemTable **full ho jaata hai** (ya kuch time ke baad), to:

> Uska data uthake ek **immutable file** banai jaati hai → jise hum kehte hain **SSTable (Sorted String Table)**

SSTable = Permanent Sorted File on Disk

Socho MemTable ek **rough page** hai, jahan data sorted likha gaya hai.
SSTable ek **register** hai jisme wohi data **neatly** copy karke likha gaya ho, **par ab usse change nahi kiya jaa sakta**.

Toh agar MemTable sorted hai, SSTable ka kaam kya?

Very smart question! Dekho:

MemTable:

- **Temporary** hai
- RAM mein hoti hai
- Jab tak power on hai tab tak kaam karti hai
- **Fast writes** ke liye bani hoti hai
- Size limited hoti hai

SSTable:

- **Permanent** hai
- Disk par hoti hai
- **Crash ke baad bhi safe**
- **Fast reads** aur future merges ke liye hoti hai
- Sorted hone ki wajah se binary search possible hoti hai

Agar SSTable hi final hai, toh direct wahi use kyun nahi kar lete?

1. Disk Bahut Slow Hoti Hai (Compared to RAM)
2. Agar hum **har write** SSTable mein karne lagen, toh har baar **poori file rewrite** karni padegi
3. Agar har choti si write SSTable mein save ho, toh **hazaaron choti files** ban jaengi,Isiliye hum ek MemTable banate hain → usme likhte hain Aur jab enough data ho → tab usse ek SSTable mein convert karte hain
4. MemTable ke sath ek **Write-Ahead Log (WAL)** bhi hoti hai, Har write pehle WAL mein jaata hai (disk par), Agar crash ho jaaye → MemTable ja chuki hoti, par WAL se **data wapas mil jaata hai**

Problem?

Jab kai saari SSTables ban jaati hain, to:

- Read karna mushkil ho jaata hai (data multiple files mein scattered hota hai)
- Space bhi zyada lagta hai (duplicate keys ho sakti hain)
- Old values delete nahi hoti properly

Yahi problem solve krne k liye to compaction hai …

Compaction is the process of:

- **Merging** multiple **SSTables** (sorted files on disk)

- **Removing duplicates or outdated data**

- **Rewriting** a clean, more efficient, sorted file

Think of it as a **clean-up + merge** job that keeps the database fast and organized.

Imagine this situation:

- You write data → it goes to **MemTable**
- When full → it's flushed to disk as a new **SSTable**
- This happens **again and again**, so now you have: 1000ssssss of ss filessss
  Each of these has some data – and maybe some **updated or deleted versions** of the same key.Reading now becomes **slow**, and storage is **wasted**.

1. Minor Compaction (Small and frequent cleanups)

- Happens **automatically and often**
- Takes a few small, recent SSTables and **merges them into one**
- Keeps the system running smoothly without big effort

Think of it like tidying up your desk every evening
Quick clean-up → less mess tomorrow

2. Major Compaction (Full deep clean)

- Takes **all SSTables** and merges them into one big file
- Removes **all duplicates, all old data**
- Leaves a single clean SSTable

But:

- This takes **more time and CPU**
- Can **slow down other operations**, so it's done rarely

Like a full house clean-up on a Sunday

3. Leveled Compaction (used in LevelDB, RocksDB)

SSTables are stored in "levels", and merged step-by-step

- New SSTables start in **Level 0**
- As files grow, they are moved and merged into **higher levels** (L1, L2, L3…)
- Each level has its **size limit**
- Merging is done in smaller, more **controlled** steps

Balances read/write performance
Keeps number of files low
Reduces read amplification (fewer files to check)

Like organizing books in racks: small rack → medium → big

EOT

Location Based Databases , Spatial Index

What Makes a Database *Location-Based*?

1. **Store Geospatial Data** – like coordinates (latitude, longitude), geometries (points, lines, polygons).
2. **Support Geospatial Queries** – like:
   - "Find all restaurants within 5 km of me"
   - "Get all users in this city"
3. **Efficient Indexing** – such as R-trees, Quad-trees, Geohashing, or spatial indexes.

Requirements : (for basic start)
1. measurable distance: scalable granularity and Uniform assignment
2. Proximity

1. Measurable Distance

Your system should be able to **calculate the distance** between two locations **accurately and consistently**, regardless of scale (local or global).

- **Scalable Granularity**: Should work well whether you're tracking meters (like inside a building) or kilometers (like between cities).
- **Uniform Assignment**: All coordinates or locations must be stored and processed in a **standardized format** (usually latitude and longitude, in degrees).

How it's Done:

- Use the **Haversine formula** or **Vincenty's formula** to calculate the **great-circle distance** between two lat/lon points.
- Or use database-built-in functions:
  - `ST_Distance()` in **PostGIS**
  - `$geoNear` in **MongoDB**
  - `GEODIST` in **RedisGeo**

Example (Python):

```
from geopy.distance import geodesic

loc1 = (22.7196, 75.8577)  # Indore
loc2 = (28.6139, 77.2090)  # Delhi

distance_km = geodesic(loc1, loc2).kilometers
print(f"Distance: {distance_km:.2f} km")
```

2. Proximity Search

Your system should support queries like:

- “Find all users within 5 km of me”
- “Show nearby hospitals within 2 km radius”

This is essential for **location-based filtering**.

Sub-requirements:

- Efficient **spatial indexing** (R-Tree, QuadTree, Geohash)
- Ability to perform **range queries** (e.g., within radius)
- Fast performance even with **large datasets**

How it’s Done:

- **PostGIS**: `ST_DWithin()`, `ST_DistanceSphere()`
- **MongoDB**: `$near`, `$geoWithin`, 2dsphere index
- **RedisGeo**: `GEORADIUS`, `GEOSEARCH`

Example (MongoDB Query):

```
db.users.createIndex({ location: "2dsphere" });

db.users.find({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [75.8577, 22.7196]  // Indore
      },
      $maxDistance: 5000  // in meters (5 km)
    }
  }
});
```

Haversine Formula (Distance between two lat/lon points) (half the verse of sine of theta)

It calculates the **shortest distance over the earth's surface** (like how a crow flies) between **two geographic coordinates** (latitude/longitude).

Latitude measures how far north or south a location is from the Equator.

Longitude measures how far east or west a location is from the prime meridian at Greenwich, England.

Both latitudes and longitudes are measured in degrees and minutes.

Why not just use Euclidean distance?

Because the Earth is **round**, not flat. Straight-line distance won't work over long distances.

Formula:
$d = 2r * \arcsin(\sqrt{\sin^2((\Delta\varphi)/2) + \cos(\varphi 1) * \cos(\varphi 2) * \sin^2((\Delta\lambda)/2)})$
https://youtu.be/t6NkBRQ2Fz0 (link for understanding the origin of this formula)

When do we use it?

Whenever you're dealing with real-world **GPS coordinates** (lat/lon), like:

- Finding the distance between two cities
- Checking how far a user is from a delivery center
- Mapping locations on a globe

Vincenty's Formula

📍 More Accurate Than Haversine

Vincenty's formula accounts for the Earth being an **ellipsoid** (not a perfect sphere), so it's more precise — especially for long distances.

Use it when **precision matters**, like **aeronautics or GIS mapping**. (The main functions of GIS include creating geographic data, managing it in a database, analyzing patterns, and displaying the results on maps.)

there is no single neat formula like Haversine for Vincenty.

Instead, **Vincenty's method** is a set of **iterative equations** — meaning it uses a **loop** to keep refining the result until it's accurate enough.

on an ellipsoid:

- The surface curvature **changes** depending on where you are (equator vs pole).
- The shortest path (geodesic) isn't a simple arc anymore — it requires solving **nonlinear equations**.

So, instead of one clean formula, Vincenty needs:

- Starting guesses
- Trigonometric equations
- Iterative refinement until convergence (e.g., difference between two λ values becomes very small)

https://youtu.be/YT3YI2TLRgQ (referring to a specific detailed answer given by a user named *Kurt* who explained Vincenty's algorithm very well.)

What are Spatial Indexes?

Just like a **book index** helps you quickly find a topic in a book, a **spatial index** helps the database quickly **locate geographic data** (points, lines, areas) — such as:

- "Find all users within 5 km"
- "List restaurants inside this polygon"
- "Search locations near me"

Without spatial indexing, these queries would be **very slow**, especially with millions of records.

Why Normal Indexes Don't Work?

A normal index (like B-Tree) is **1D** — it works for sorted values like numbers or strings.

But **location data is 2D or even 3D** (latitude, longitude, sometimes altitude), so we need a way to index and search in **multi-dimensional space**.

Types

1. R-Tree – (Rectangle Tree)

Tumne India ke map ko **multiple rectangles** (boxes) mein divide kar diya.

- Har box ke andar kuch locations (restaurants, shops) hain.
- Agar user Delhi mein hai, to tum sirf **Delhi waale boxes** check karogi.
- **Baaki boxes ignore** → Query fast.

Used in: PostGIS, GIS software

Example:

> “Find all pizza places inside this rectangular area (box).”

2. QuadTree – (4 parts mein divide)

Poore map ko 4 parts mein divide kiya –
phir unke bhi 4 parts –
fir unke bhi 4...

- Har part ke andar sirf nearby places hain.
- Jis part mein user hai, **usme se hi results lao**.

Used in: Google Maps, Zoom features

Example:

> "Map zoom karte waqt sirf nearby tiles load karo."

3. Geohash – (Location ko string mein badal do)

Lat-Long ko ek **short code** mein convert kar diya:

- Jaise: `"ezs42"` = Indore ka ek area
- Nearby locations = almost same geohash
- Is code pe index bana diya → search fast

Used in: Uber, RedisGeo, Firebase

Example:

> "User ka geohash = `ezs42` → sabhi cabs with same geohash dikhado."

4. Hilbert Curve – (2D ko 1D bana ke nearby data paas paas rakho)

Ek **zigzag line** poore map ke andar aise chali jaati hai ki har area ko touch kare...

- Nearby jagahen = is line ke sequence mein **close** hoti hain
- So data ko 1D index mein daal ke fast search

Used in: BigQuery, GeoMesa, Cassandra

Example:

> "10 crore location points mein se fast search chahiye → Hilbert use hota hai."

EOT

Data Migration , Database Migration

You need to pack your items (data), decide what to take (filtering), transport it safely (consistency), and unpack it correctly (format).

Data migration means **moving data from one system to another**.

- From **one database to another** (e.g., MySQL → PostgreSQL)
- Restructuring schema (e.g., normalize → denormalize)
- From **on-premise to cloud** (e.g., local server → AWS RDS)
- From **monolith to microservices**
- Moving from legacy system to modern stack
- During a **schema change**, **sharding**, or **application redesign**

(A monolith is a single big application that contains everything together — frontend, backend, database logic, etc.)
(Microservices is an architecture where the application is broken into many small independent services, each doing one thing well.)

Key Challenges

1. **Downtime** – How to avoid or reduce it
2. **Data Integrity** – Ensure nothing is lost or corrupted
3. **Data Consistency** – Old and new data match
4. **Performance** – Migration shouldn't crash the system
5. **Live Traffic Handling** – What if users are still using the system?

Types

Type of Migration	What It Means	Tools Used	Real-Life Example	Why We Do It
Storage Migration	Moving files from one storage system to another (like hard disk → cloud)	`rsync`, `robocopy`, AWS S3 CLI	Move videos from a company server to Google Drive or Amazon S3	Save cost, improve speed, or increase space

Database Migration	Moving data from one database to another (like MySQL → PostgreSQL)	AWS DMS, pgloader, SQL tools	A company changes its database to something faster or cheaper	For better performance or new features
Application Migration	Moving data while changing software (like old CRM → new CRM)	Export/Import, APIs, CSV files	Shifting employee records from one HR app to another	Upgrade tools or switch to better software
Cloud Migration	Moving data from local computer/server to cloud (or one cloud to another)	AWS DMS, Azure Migrate, Google Transfer	Company shifts website database to AWS RDS (cloud)	Make app more reliable, faster, and easier to scale
Business Process Migration	Moving data during a big business change (like company merger)	ETL tools, SQL, Kafka	Two companies merge and combine their customer data	To keep all systems working after a business change
Schema Migration	Changing the structure of your database (add/remove columns, split tables)	Alembic, Liquibase, Django Migrations	Add a new column `phone_number` to `users` table or split `users` into multiple tables	Support new features or better data organization
Data Format Migration	Changing the format of the data file (CSV → JSON or CSV → Parquet)	Python Pandas, Spark, File converters	Changing logs from `.csv` to `.parquet` for faster analytics	Use faster formats, save space, or work with new tools
Platform Migration	Moving your app/data to a different OS (like Windows → Linux) or environment	Docker, Ansible, Kubernetes	Moving app from Windows server to Linux-based server	Reduce cost, standardize or modernize setup

Microservices Migration	Splitting one big app's database into smaller service databases	Kafka, Debezium, SQL, Python scripts	Breaking a big `users` table into `login_service` and `profile_service` for microservices	Scale faster, allow teams to work independently

Strategies

Strategy	Simple Meaning	How It Works	When to Use	Pros	Cons
Big Bang Migration	Move everything at once	Stop old system → Move all data → Start new one	Small systems or when downtime is okay	Simple, fast	Needs complete downtime, risky if it fails
Incremental Migration	Move data in small steps	Move some data → Keep both systems running → Repeat	Large apps with real-time users	No big downtime, safer	Complex logic, longer process
Trickle Migration	Move few users or features first	Pick 5–10% users → Test → Slowly move more	Testing new system gradually	Low risk, easy to monitor	Takes more time
Parallel Run	Run both systems side-by-side	New + old run together → Compare results → Switch	Banking, healthcare, sensitive systems	Safest, double-check possible	Uses double resources, expensive
Phased Migration	Move one module at a time	Move `users` first → then `orders` → then `payments`	When splitting monolith into microservices	Step-by-step testing, controlled	Needs planning and coordination

Dual Write Migration	Write to both old and new systems together	Every new write goes to both DBs → Sync old data too	During system upgrade without data loss	No missed data, safe	Write failures may cause mismatch
Backfill + Live Sync	First move old data, then sync new changes	Migrate past data in bulk → Sync new data in real-time	When historical + live data both matter	Perfect for big data + real-time sync	Needs CDC tools like Kafka or Debezium

Now , We will be talking about database migration …

Data Migration

You move all your Excel files, images, videos, and reports to Google Drive.
That's data migration.

Database Migration

You move your `users` table from MySQL to PostgreSQL, including schema and records.
That's database migration.

https://youtu.be/SYaSlxEEHvI

Why Do We Migrate Databases?

Reason	Description
Change DB Engine	Move from MySQL to PostgreSQL, or SQL to NoSQL
Cloud Adoption	Move from on-premise DB to cloud (like AWS RDS, GCP)
Version Upgrade	Move from Oracle 10g to 12c, or Postgres 10 → 15
Schema Redesign	Normalize, split tables, or switch to microservices
Performance	Use a faster, more scalable, or cost-effective DB
Business Changes	Mergers, replatforming, legal/regulatory compliance

Types of Database Migrations

Type	What’s Changing
Homogeneous Migration	Same DB type (e.g., MySQL 5.7 → MySQL 8.0)
Heterogeneous Migration	Different DB types (e.g., Oracle → PostgreSQL)
Cloud Migration	Moving DB to cloud (e.g., on-premise SQL → AWS RDS)
Schema Migration	Changing structure (adding columns, new relations)
Sharding Migration	Splitting large DB into smaller parts
Replication-based Migration	Using CDC or tools like Kafka to sync data in real-time

(CDC = Change Data Capture
It is a **technique used to track and capture changes** (like inserts, updates, deletes) made to the data in a **database**, and then send those changes **somewhere else**, like another database or a real-time system.)

Ab process smjhte hai :

Step 1: Planning

Sabse pehle :

- Kya migrate karna hai?
  → Data? Schema? Procedures? Purani ya naye version mein?
- Kahan se kahan migrate karna hai?
  → e.g., MySQL se PostgreSQL
- Kya downtime allow hai? Ya live traffic handle karna hai?
- Kaunsi strategy use hogi?
  → Big bang, dual write, backfill?

Output: Ek full migration plan ban jaata hai.

Step 2: Assessment (Check karna compatibility)

- **Data types match karte hain ya nahi**?
  (e.g., `int` vs `bigint`, `text` vs `varchar`)
- **Stored procedures**, **triggers**, aur **functions** nayi DB mein kaam karenge ya nahi?
- Primary keys, indexes, constraints, relationships ko study karna.

Output: Samajh aa jaata hai kya convert karna padega, kya chhodna padega.

Step 3: Schema Migration (Structure banana)

Ab tum naye DB mein:

- Same tables ka structure create karogi.
- Columns, data types, keys, constraints define karogi.
- Stored procedures aur views ko rewrite karogi (agar zarurat hai).

Tools: `Flyway`, `Liquibase`, `DBeaver`, `pgAdmin`, `Oracle SQL Dev`.

Output: Nayi database ready hai data accept karne ke liye.

Step 4: Data Migration (Data ka transfer)

Ab actual **data copy** karna start hota hai.

2 tareeke:

1. **Offline/Batch** — sab data ek baar mein transfer (Big Bang)
2. **Real-time sync** — new DB ke saath live update bhi hoti rahe

Tools: `pgloader`, `AWS DMS`, Python + Pandas, SQL scripts, CSV import/export

Example:

```
INSERT INTO new_db.users SELECT * FROM old_db.users;
```

Output: Data successfully nayi DB mein aa gaya.

Step 5: Data Validation (Check karna sab sahi aaya ya nahi)

- Row counts match ho rahe hain?
- Data loss toh nahi hua?
- Jo queries old DB mein result deti thi, woh hi new mein bhi de rahi hai?
- Primary keys, indexes, aur constraints kaam kar rahe hain?

Tip: Use `checksums`, `row comparison scripts`, `unit testing`.

Output: Confirm ho jaata hai ki data migrate hua sahi se.

Step 6: Live Sync Setup (Agar users active hain)

Agar tumhara app live hai, toh migration ke baad:

- Tumko **new updates bhi dono jagah likhne** padte hain (dual write)
- Ya fir tumko **real-time syncing** karni padti hai (Kafka, Debezium)

Tools:

- Kafka + Debezium (CDC)
- AWS DMS CDC mode
- Custom code in Python/Node.js

Output: Old aur new DB sync mein chal rahe hote hain.

Step 7: Switch Over (Naye DB par shift karna)

Ab jab sab kuch ready ho gaya:

- Tum app ka DB connection string update karti ho
- App new DB se data read/write karta hai

Tum monitor karti ho:

- Logs
- Errors
- Performance

Output: Naya DB live ho jaata hai production ke liye

Step 8: Post-Migration Cleanup

Last step mein:

- Old DB ko archive ya delete karna (optional)
- New DB ko optimize karna (indexes, vacuum)
- Monitoring setup karna
- Documentation complete karna

Output: Migration process safely complete ho jaata hai!

```
Planning
    ↓
Assessment
    ↓
Schema Migration
    ↓
Data Migration
    ↓
Validation
    ↓
Live Sync (if needed)
    ↓
Switch Over
    ↓
Cleanup
```

Real-Life Example:

Suppose:

- Tum ek ecommerce app ke liye `MySQL → PostgreSQL` migrate kar rahi ho.
- Downtime allowed nahi hai.(mtlb System ko kabhi bhi band nahi kiya jaa sakta not even for a short time)
- Tum schema banana, pgloader se data copy, Debezium se sync, aur last mein app config change karke switch over kar dogi.

Boom! Database migration ho gaya safely.

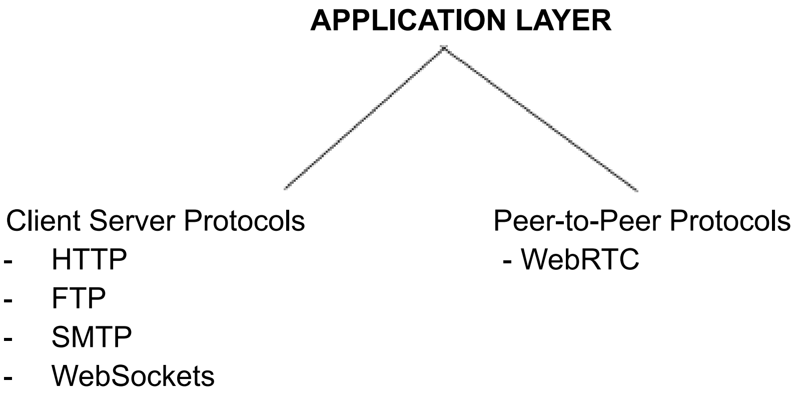
(tools k baare m sabhi, we see them later kabhi)

EOT

Network Protocols

Key Purposes of Network Protocols:

Purpose	Description
Data Transmission	Ensures data is sent and received correctly.
Error Detection & Handling	Helps identify and correct errors during data transfer.
Data Compression	Reduces the size of data for faster transmission.
Authentication & Security	Verifies identity and protects data.
Flow Control	Manages how fast data is sent to avoid overwhelming the receiver.

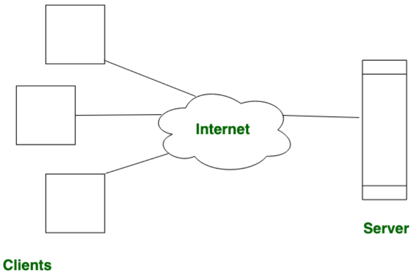


APPLICATION LAYER

Client Server Protocols
- HTTP
- FTP
- SMTP
- WebSockets

Peer-to-Peer Protocols
- WebRTC

In the Client-Server Network, Clients and servers are differentiated, and Specific servers and clients are present. In Client-Server Network, a Centralized server is used to store the data because its management is centralized. In Client-Server Network, the Server responds to the services which is requested by the Client.



Client-Server Network Model

(sabhi protocols bhi ese hi hai bss websocket different hsi qki vo bidirectional hai , client can talk to server , server can talk to client but ye peer to peer nahi hai qki 2 clients can't talk aapas mei smjhe , whatsapp telegram wagera koi messaging app design krne k kaam aata h)

HTTP (Hypertext Transfer Protocol)

HTTP provides a standard between a web browser and a web server to establish communication. It is a set of rules for transferring data from one computer to another. Data such as text, images, and other multimedia files are shared on the World Wide Web. Whenever a web user opens their web browser, the user indirectly uses HTTP. It is an application protocol that is used for distributed, collaborative, hypermedia information systems. HTTP is **stateless**, which means it does **not remember** anything between two requests.

Every time you refresh a page or click a new link, it's like starting fresh.

Types of HTTP Requests

Method	Purpose	What It Means	Real-Life Example
GET	Retrieve data from server	"Give me this data"	Viewing a webpage, loading profile info
POST	Submit new data to the server	"Here's some new data"	Filling a form, uploading a photo
PUT	Create or completely update a resource	"Replace everything with this new version"	Editing full profile details
PATCH	Partially update a resource	"Update only this part of it"	Changing just your phone number in your profile
DELETE	Delete a specific resource from the server	"Remove this item"	Deleting a post or account
HEAD	Get only headers, not the content	"Tell me about this page, but don't send the full page"	Checking if a page exists without loading it

OPTIONS	Get supported methods & communication info	"What can I do with this resource?"	Browser checks what methods (GET, POST, etc.) allowed
TRACE	Debugging request by echoing it back	"Send me back exactly what I sent you"	Rarely used due to security concerns
CONNECT	Create a tunnel (mostly for HTTPS)	"Make a secure connection through a proxy"	Used when connecting to secure websites (HTTPS)

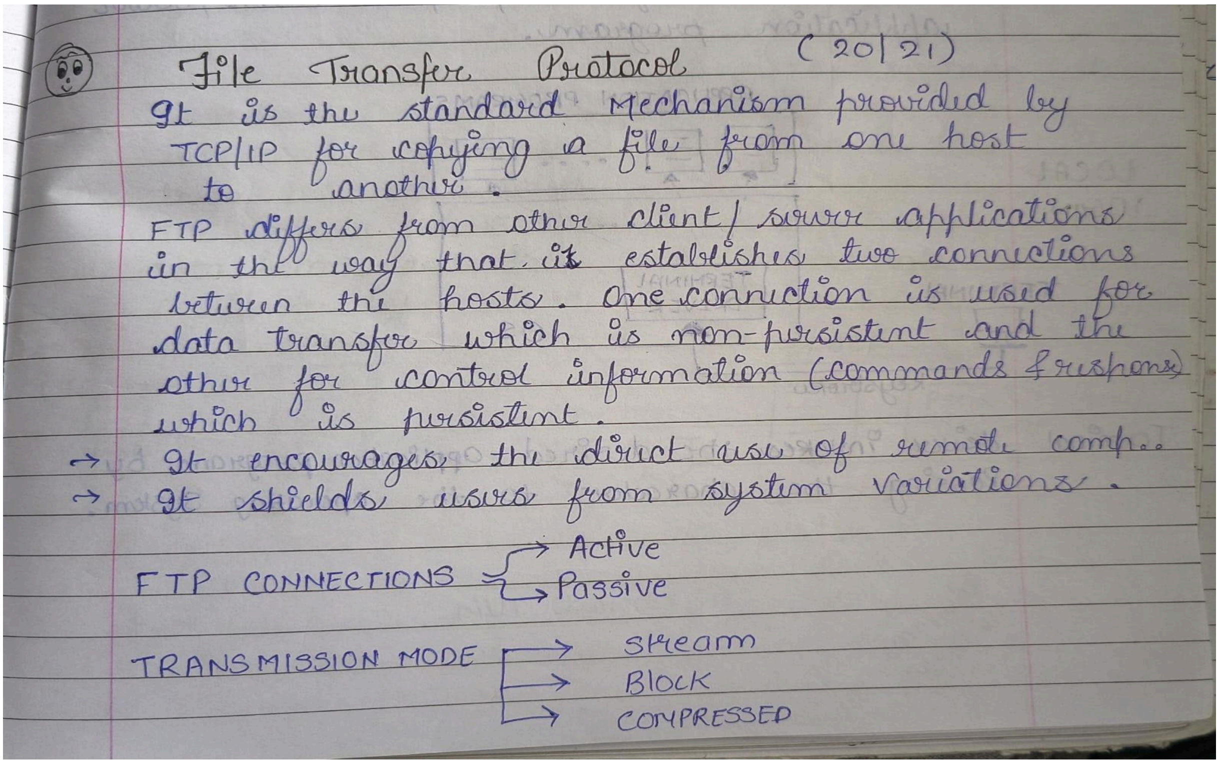
HTTP Response Codes (Status Codes)

Code	Category	Meaning	What It Means	Real-Life Example
100	Informational	Continue	Server received your request, keep sending more	Uploading large files in parts
200	Success	OK	Request was successful, everything went fine ✅	Webpage loaded correctly
204	Success	No Content	Successful but nothing to show	Successfully deleted something
301	Redirection	Moved Permanently	Resource permanently moved to new URL	Old blog redirects to new one
400	Client Error	Bad Request	The request is broken or incorrect	Wrong data format sent

401	Client Error	Unauthorized	You need to log in first	Trying to access Gmail without logging in
403	Client Error	Forbidden	You’re not allowed, even if logged in	Trying to open admin dashboard without permission
404	Client Error	Not Found	The page or resource doesn’t exist	Typing wrong URL
405	Client Error	Method Not Allowed	You used the wrong HTTP method	Sending POST where only GET is allowed
408	Client Error	Request Timeout	You took too long, server stopped waiting	Slow internet, server cancels your request
429	Client Error	Too Many Requests	You sent too many requests in a short time	Refreshing a page too often
500	Server Error	Internal Server Error	Server crashed or had an issue	Server-side bug
503	Server Error	Service Unavailable	Server is down or overloaded	Too much traffic on a website
504	Server Error	Gateway Timeout	Server didn’t respond in time	Backend service is taking too long

Security k liye HTTPS use krte hai baaki using SSL/TLS certificate , SSL /TLS we will read it in detail later .

FTP (File transfer protocol)



File Transfer Protocol (20|21)

It is the standard Mechanism provided by TCP/IP for copying a file from one host to another.

FTP differs from other client/server applications in the way that it establishes two connections between the hosts. One connection is used for data transfer which is non-persistent and the other for control information (commands & responses) which is persistent.

→ It encourages the direct use of remote comp..

→ It shields users from system variations.

FTP CONNECTIONS → Active / → Passive

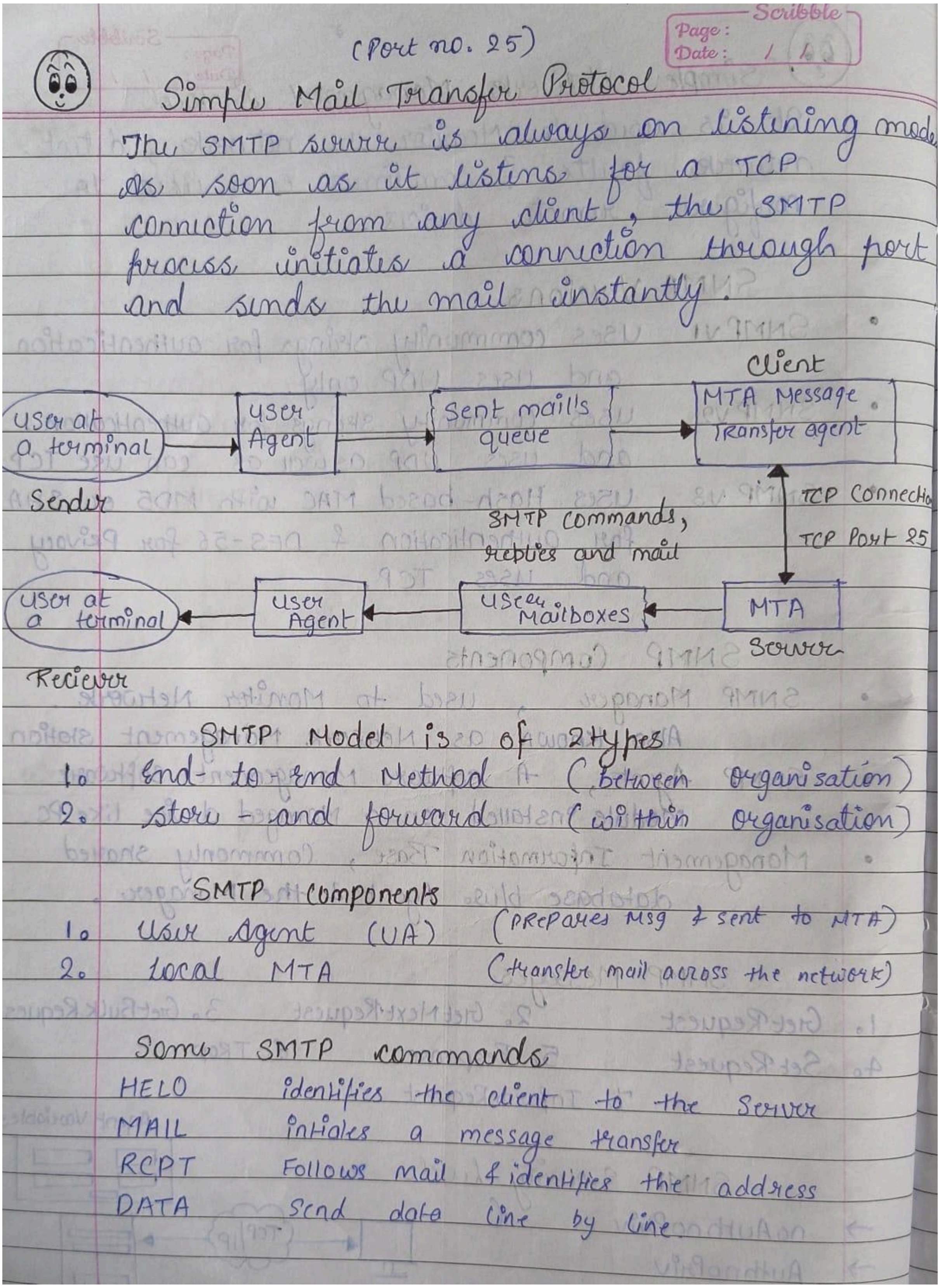
TRANSMISSION MODE → Stream / → Block / → COMPRESSED

Active mei server client ko call krta hai mtlb Server **client ke port pe connection** banata hai. Aur passive mei client server ko call krta hai mtlb Server ek port batata hai, aur **client khud connect karta hai**.

FTP Commands

Command	Meaning
`cd`	Changes the **current folder** on the remote (server) side
`close`	Closes the **connection** to the FTP server, but doesn't quit FTP
`quit`	Exits or **quits** the FTP session completely
`pwd`	Shows the **current folder path** on the remote server
`dir` or `ls`	Lists all **files and folders** in the current remote directory
`help`	Shows a list of **all FTP client commands** you can use
`remotehelp`	Shows all the **commands supported by the server** (server-side help)
`type`	Sets the **file transfer type** (e.g., binary or ASCII)
`struct`	Sets the **file structure type** (e.g., file, record, or page structure)

SMTP (Simple Mail Transfer Protocol)



(Port no. 25)

Simple Mail Transfer Protocol

The SMTP server is always on listening mode as soon as it listens for a TCP connection from any client, the SMTP process initiates a connection through port and sends the mail instantly.

Client
user at a terminal
User Agent
Sent mail's queue
MTA Message Transfer agent
Sender
TCP Connection
TCP Port 25
SMTP commands, replies and mail
user at a terminal
User Agent
User Mailboxes
MTA
Server
Reciever

SMTP Model is of 2types

1. End-to-end Method (between organisation)
2. store-and forward (within organisation)

SMTP components

1. User Agent (UA) (PREPARES MSg & sent to MTA)
2. Local MTA (transfer mail across the network)

Some SMTP commands

HELO	identifies the client to the Server
MAIL	initiates a message transfer
RCPT	Follows mail & identifies the address
DATA	Send data line by line

Ye to send krne k liye hota hai bss mail , recieve k liye to alg hote hai IMAP yaa POP3
To ye unhi k sath use hota hai , mta - msg transfer agent h

internet Message Access Protocol .

*

Difference between POP3 and IMAP

POP3	IMAP
It is a simple protocol that only allows downloading messages from your Inbox to your local comp..	It is much more advance and allows the user to see all the folders on the mail Server.
The mail can only be accessed from a single device at a time.	Messages can be accessed across Multiple Devices
To read the mail it has to be downloaded on the local system.	The mail content can be read partially before downloading.
The user can't create, delete or rename email on the mail server	The user can create, delete or remove email on the mail server.
It is Unidirectional	It is Bi-directional
The user can't organize mails in the mailbox of the mail server.	The user can organize the emails directly on the mail server.
It doesn't allow a user to sync emails	It allows a user to sync their emails
It is fast	It is slower

WebSocket

WebSocket is bidirectional, a full-duplex protocol that is used in the same scenario of client-server communication, unlike HTTP which starts from **ws://** or **wss://**. It is a stateful protocol, which means the connection between client and server will stay alive until it gets terminated by either party (client or server). After closing the connection by either of the client or server, the connection is terminated from both ends. Let's take an example of client-server communication, there is the client which is a web browser, and a server, whenever we initiate the connection between client and server, the client-server makes the handshaking and decides to create a new connection and this connection will keep alive until terminated by any of them. When the connection is established and alive the communication takes place using the same connection channel until it is terminated.

Feature	HTTP	WebSocket
Connection	One-way (client → server only)	Two-way (client ↔ server)
Connection Type	Short-lived (request-response)	Long-lived (open till closed)
Use Cases	Web browsing, forms, APIs	Chat apps, gaming, stock prices
Real-time?	No (needs refresh/polling)	Yes (instant updates)

Peer-to-Peer Network

This model does not differentiate the clients and the servers, In this each and every node is itself client and server. In Peer-to-Peer Network, Each and every node can do both request and respond for the services.

- Peer-to-peer networks are often created by collections of 12 or fewer machines. All of these computers use unique security to keep their data, but they also share data with every other node.
- In peer-to-peer networks, the nodes both consume and produce resources. Therefore, as the number of nodes grows, so does the peer-to-peer network's capability for resource sharing. This is distinct from client-server networks where an increase in nodes causes the server to become overloaded.
- It is challenging to give nodes in peer-to-peer networks proper security because they function as both clients and servers. A denial of service attack may result from this.

WebRTC

WebRTC stands for Web Real-Time Communication. It is an open source and free project that used to provide real-time communication to mobile applications and web browsers with the help of API's(Application Programming Interface).

#Tum laptop pe Google Meet kholti ho aur dusri side se koi call karta hai.

- Video + Audio tum dono ke devices ke beech **directly** stream hota hai.
- Ye sab **WebRTC** ke through ho raha hai.
- Browser hi camera, mic, network sab handle karta hai — bina extra software ke.

WebRTC ke 3 Main Components –

MediaStream – Captures audio/video from your camera or mic using `getUserMedia()`.
"Mujhe kya bhejna hai?"

RTCPeerConnection – Connects your browser directly to another for voice/video calls.
"Kisse connect hona hai?"

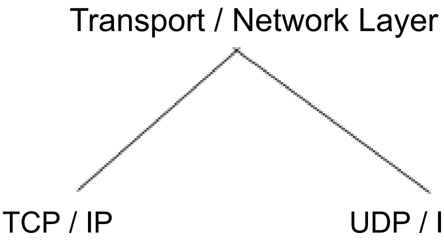
RTCDataChannel – Sends extra data (like chat or files) between connected users.
"Aur kuch data bhejna hai?"

WebRTC vs WebSocket

Feature	WebRTC	WebSocket
Purpose	Real-time **media** + data sharing	Real-time **text/data** messaging
Audio/Video?	Yes	No
Peer-to-peer?	Yes (direct browser-to-browser)	No (via server only)
Use Cases	Video calls, audio calls, P2P share	Live chat, games, notifications

Let's end application layer ,

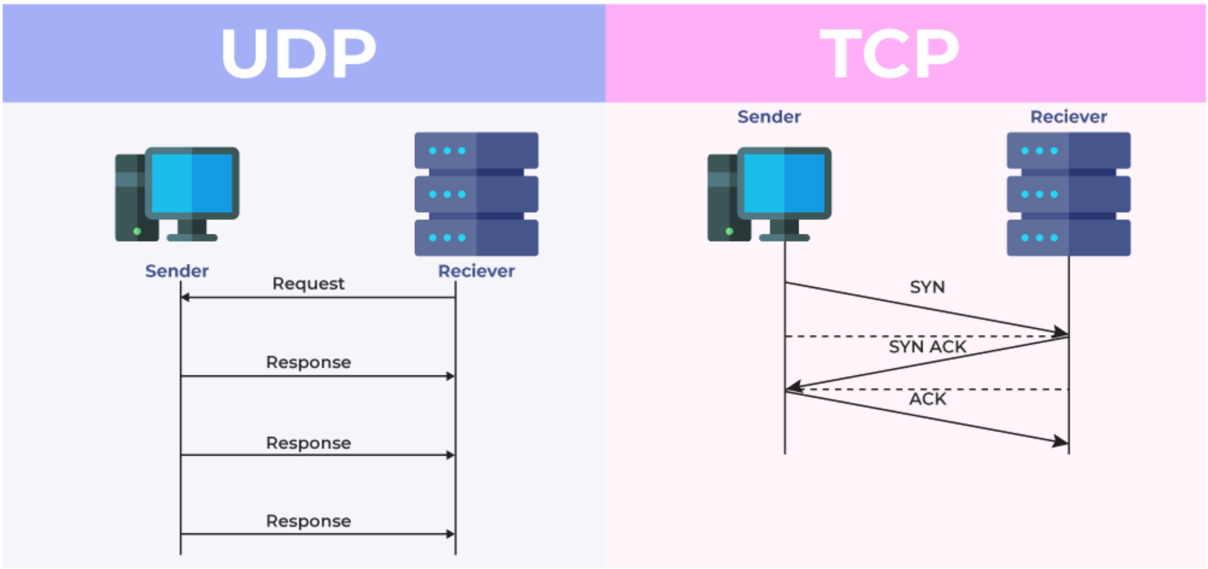
Feature	Client-Server Model	Peer-to-Peer Model
Structure	Centralized (one main server)	Decentralized (all devices equal)
Role	Server serves, client consumes	All devices serve and consume
Speed (Scalability)	Can slow down if many clients connect	Usually scales better with more peers
Example	Google, Facebook, YouTube	BitTorrent, ShareIt, Blockchain
Reliability	Depends on server	Depends on multiple peers
Cost	Needs strong server infrastructure	Cheaper, no central server required



Transport / Network Layer

TCP / IP

UDP / IP



TCP/IP – Reliable Delivery

TCP (Transmission Control Protocol)
Data ko **break karke, order mein bhejta hai**, aur ensure karta hai ki sab data sahi se pahucha ya nahi.

- Connection-based (jaise phone call)
- Data ordered hota hai
- Reliable but slower
- Example: Web browsing, Email, File download

Main tab tak wait karunga jab tak data safely aur sahi order mein nahi mil jaata."

UDP/IP – Fast but Unreliable

UDP (User Datagram Protocol)
Data ko quickly bhejta hai, **check nahi karta** ki pahucha ya nahi.

- No connection needed (jaise voice note bhejna)
- Faster but no guarantee
- Useful where speed is more important than reliability
- Example: Live video streaming, Online gaming, Video calls

📌 *"Main fast bhejta hoon, pahucha ya nahi uski chinta nahi."*

Basis	TCP (Transmission Control Protocol)	UDP (User Datagram Protocol)
Type of Service	**Connection-oriented** – Pehle connection banta hai, fir data bheja jaata hai.	**Connectionless** – Seedha data bhejna start kar deta hai, bina connection banaye.
Reliability	Reliable – Pura data safely pahuchta hai, aur server confirm karta hai.	Not Reliable – Data mil bhi sakta hai, miss bhi ho sakta hai.
Error Checking	Strong error check + data control + feedback (acknowledgment).	Basic error check only (checksum), no feedback.
Acknowledgment	Har message ke bad confirmation milta hai – "Haan, mil gaya!"	No confirmation – bas bhej diya, ab dekha jaayega.
Sequence	Packets **ordered** milte hain – jese bheja, waise hi milta hai.	Packets **unordered** milte hain – mil bhi sakta hai ulte sequence mein.
Speed	Thoda slow – secure aur check karke bhejta hai.	Very fast – sidha bhejta hai, checking me time nahi lagata.
Retransmission	Lost packets wapas bheje ja sakte hain.	Lost packets dobara nahi bheje jaate.
Header Size	20–60 bytes (badi info hoti hai)	8 bytes only (chhoti info)
Weight	Heavy-weight (zyada data handle karta hai)	Light-weight (simple aur lightweight)
Handshake (Connection)	Uses 3-way handshake (SYN, SYN-ACK, ACK) – pehle “Hi”, fir “Okay”, fir “Start”	No handshake – sidha “Bhej diya”
Broadcast Support	No broadcasting	Broadcasting/multicasting possible
Used By Protocols	HTTP, HTTPS, FTP, SMTP, Telnet	DNS, DHCP, TFTP, VoIP, SNMP, RIP
Data Stream Type	**Byte stream** – data ek continuous stream mein aata hai	**Message stream** – data blocks mein bhejta hai
Overhead (Load)	More overhead than UDP due to error checking	Very low overhead – faster but less safe
Applications	Web browsing, email, file transfer, banking – jahan **accuracy zaroori hai**	Video streaming, online games, voice calls – jahan **speed zaroori hai**

EOT

CAP THEOREM

Term	Meaning	Example
C – Consistency	All nodes see the same data at the same time.	If you write data to one node and read from another, you get the latest value.
A – Availability	The system always responds to requests (doesn’t go down).	Even if a node fails, users get a response.
P – Partition Tolerance	The system works even if communication between nodes is broken.	If the network splits, the system keeps running.

The Rule of CAP Theorem

> You can only choose 2 out of 3: Consistency, Availability, Partition Tolerance.

CA mein "P kyon nahi ho sakta?"
C (Consistency) ka matlab hai: har user ko latest sahi data hi milna chahiye.

A (Availability) ka matlab hai: har user ko turant response milna chahiye — chahe kuch bhi ho.

Ab agar Partition ho gaya (network toot gaya), to:

- System confirm nahi kar sakta ki data har jagah same hai ya nahi.
- To agar tum C follow karti ho, to tumhe wait karna padega jab tak network wapas theek na ho — kyunki bina doosre node se pooche tum sure nahi ho sakti ki data consistent hai.
- But Availability allow nahi karti wait karna, wo bolti hai: “Turant jawab do!”

Isliye:

CA dono tab tak possible hai jab Partition na ho.
Partition aaya, to ya to tum wait karo (C), ya turant jawab do (A) — dono nahi ho sakta.

CP mein "A kyon nahi ho sakta?"

- C chahiye → Matlab system har baar sahi aur updated data de.
- P bhi chahiye → Matlab network fail ho gaya ho, phir bhi system band na ho.
- Ab agar network fail hai (Partition ho gaya),
  - Aur tumhe Consistency bhi chahiye, to tumhe verify karna padega ki doosre server pe data same hai ya nahi.
  - Lekin network hi fail hai!
    Tum doosre node se pooch hi nahi sakti, to verify nahi ho raha.
- Jab tak verify nahi hota, tum response nahi dogi (taaki galat data na mile).

Isliye:

Availability chali gayi — kyunki system turant reply nahi de sakta.

PA mein "C kyon nahi ho sakti?"

- P chahiye → Matlab network toot jaye to bhi system chalna chahiye.
- A chahiye → Matlab system har request ka turant response de.
- Ab agar network toot gaya hai, aur tum turant response dogi, to:
  - Tumhare paas baaki server se poochhne ka option hi nahi hai.
  - To tum confirm nahi kar sakti ki data updated hai ya nahi.
  - Tum jo bhi reply dogi, wo shayad purana ho sakta hai.

👉 Isliye:

Consistency nahi milti, kyunki tum confirm nahi kar pa rahi ho ki sab jagah same data hai.

CAP theorem banaya gaya tha taaki distributed systems ke design mein logical limit samajh aaye — aur hum impossible expectations na rakhein.

EOT

Data Consistency Levels , Transaction Isolation Levels , Two General's Problem

Consistency Type	Guarantee	Behavior	Latency	Common Usage
Strong Consistency	Every read after a write returns the **most recent** value.	Like interacting with a **single machine** — total correctness.	High	Banking, Transactions
Linearizability	Reads and writes appear as if they occur in **real-time order**.	Strongest form — totally ordered system-wide.	Very High	Leader elections, system config
Sequential Consistency	Operations appear in the **same order** to all, but not necessarily real-time.	Order is preserved, but delays possible between write and visibility.	Medium	Multiplayer games (action sync)
Causal Consistency	Operations that are **causally related** are seen in the same order.	Cause-effect relationships preserved (e.g., post → comment).	Medium	Social media, chat apps
Eventual Consistency	Data will **eventually** become consistent across all replicas.	You might get stale data temporarily, but eventually it's updated.	Low	Likes, Feeds, Followers list
Read-Your-Writes (RYW)	After writing something, your **own reads** will reflect the write.	Only for the same user/session; others might not see it immediately.	Low	Profile updates, drafts

Monotonic Reads	Once you see a value, you'll **never see an older value** again.	Guarantees data won’t move backward.	Medium	Timeline scrolls
Session Consistency	Within a session, reads are always consistent with your writes.	Simple version of causal + RYW within user session.	Medium	Shopping carts, dashboards

How Consistency is maintained?

1. **Replication** – Data is copied to multiple nodes.
2. **Quorum** – Read from/write to a majority of nodes to maintain agreement.
3. **Consensus Protocols** – Like Paxos or Raft used to ensure all agree on data state.
4. **Locks** – Used to prevent conflicting writes (in Strong consistency setups).
5. **Vector Clocks** – Used in Eventual consistency to track causality between updates.

Challenges in Consistency

1. **Network Latency** – Hard to sync all nodes instantly.
2. **Node Failure** – Some nodes might go down during write.
3. **Partitioning** – If network breaks, confirming data gets tough.
4. **Write Conflicts** – Same data updated at 2 places, how to merge?

Real-World Databases and Their Consistency

Database	Default Consistency
MongoDB	CP – Strong (can be tuned)
Cassandra	AP – Eventual (can be tuned)
DynamoDB	Eventual (optional Strong reads)
HBase	CP – Strong consistency
Redis	Varies – can be strong or eventual depending on setup

To reduce latency we can somewhat cache some of the information
You fetch it once and cache it in other server and connect it indirectly and response it back always instead directly getting data , it can reduce latency but not remove it , there is always a chance of cache miss.

So if you have multiple copies of data then it can solve your lots of problems.
It will cure your single point of failure also network issues.

What is the Two Generals' Problem?

The Two Generals' Problem is a thought experiment in computer science and distributed systems that shows how two systems cannot reliably agree over an unreliable communication channel — no matter how many messages are sent.

The Story (Simple Version):

- Imagine **two generals**, General A and General B, are planning to attack a city.
- They are stationed on **two opposite hills**, and they must **attack at the same time** to win.
- The only way they can communicate is by **sending messengers through the valley** — which is **controlled by the enemy**.

Now, suppose:

1. General A sends a message: “Let’s attack at 5 AM.”
2. But General A can’t be sure General B received it safely.
3. So General B sends back: “I got your message. Yes, 5 AM works.”
4. But now General B isn’t sure if **his** message reached A.
5. So A sends back another message: “Got your confirmation.”

...and this goes on **forever**, because **neither general can be 100% sure the last message was delivered**.
Thus, they can **never confidently agree** ki attack krna ki ni krna kab krna vo krega ki nahi ufff .

Why is this Important in System Design?

This problem illustrates the **challenge of achieving consensus** in distributed systems (like microservices, databases, or networked machines) where:

- Message loss is possible
- Network is unreliable
- Acknowledgments can be lost too

It proves that:

> **Perfect coordination between two parties over an unreliable network is impossible** — unless there's some external guarantee of delivery.

That's why we build systems with **timeouts, retries, logs, consensus protocols, and fault-tolerance**.

Technique	What it Does
Timeouts & Retries	If no response in time, resend message or abort
Acknowledge + Logging	Systems log messages and resends if crash happens
Quorum-based Consensus	Requires agreement from majority (used in Raft, Paxos)
2-Phase Commit (2PC)	Used in databases to coordinate transactions across nodes
Eventually Consistent Systems	Assume all nodes will be in sync eventually (used in NoSQL systems)

Baad mei we will see all in detail !

After consistency levels, now let’s dive into Transaction Isolation Levels — another core concept in databases and system design, especially for maintaining data correctness when many users access data at the same time.

What Are Transaction Isolation Levels?

When multiple users or systems are **reading/writing the same data at the same time**, we need **rules** to protect the data.
These rules are called **transaction isolation levels**.

They define: "How much one transaction is **isolated** from another."

More isolation = more **safety** but **slower**.
Less isolation = more **speed**, but **risk of incorrect data**.

The 4 Standard Isolation Levels (from SQL standard)

Level	Allows	Prevents	Speed	Safety
Read Uncommitted	✅ Dirty Reads, ✅ Non-repeatable Reads, ✅ Phantoms	❌ Nothing is really prevented	✅ Fastest	❌ Least Safe
Read Committed	❌ Dirty Reads, ✅ Non-repeatable Reads, ✅ Phantoms	✅ Dirty reads prevented	✅ Fast	⚠️ Medium
Repeatable Read	❌ Dirty Reads, ❌ Non-repeatable Reads, ✅ Phantoms	✅ Reads are repeatable	❌ Slower	✅ Safer
Serializable	❌ Dirty, ❌ Non-repeatable, ❌ Phantoms	✅ Total Isolation – like running one at a time	❌❌ Slowest	✅✅ Very Safe

Dirty Read	Reading data that **someone else wrote but hasn't saved yet**	You see ₹1000 deposit, but user cancels it before saving
Non-repeatable Read	You read a row, and **it changes** when you read again	You check your order = "Pending", check again = "Delivered"
Phantom Read	You run a query and **new rows appear** the next time	"SELECT * FROM orders WHERE qty > 5" → new rows show up later

Ye dirty read we can say truecaller p call aane k pehle pta chl jaata hai call kr rha saamne waala insaan

Non repeatable umm , whatsapp msg seen , even if you mark it as unread , it's not gonna go back to not seen for receiver

And phantom read tumne apne msgs check kiye no one msged , 2 sec. Baad ek msg aaya to count 0 to 1 hogaya

EOT

Message Queues , DB as a Msg queue, Publisher - Subscriber Model

What is a Message Queue?

A Message Queue is a system that lets two services talk to each other asynchronously using messages.

It works just like a "Line (queue)" where:

- One side sends messages = Producer (Sender)
- Other side reads messages = Consumer (Receiver)

Messages wait in the queue until the receiver is ready to process them — like people standing in line!

Why do we use Message Queues?

Reason	Explanation
Decoupling	Sender & receiver don't have to be online at the same time
Reliability	If one service crashes, messages won't be lost — they wait in the queue
Scalability	Can handle high traffic by processing messages one-by-one or in batches
Asynchronous	Sender doesn't wait for receiver's response — faster and more flexible

Popular Message Queue Tools

Tool	Description
RabbitMQ	Lightweight, supports complex routing
Kafka	High-throughput, real-time big data processing
Amazon SQS	Cloud-based simple queue service
ActiveMQ	Reliable Java-based messaging
Redis Streams	Fast, in-memory queuing

Now we will discuss some problems and possible solutions okayiee…

1. Duplicate Messages (Same message processed multiple times)

Problem:

- Consumer receives the same message more than once.
- This may lead to double payment, double order, etc.

Why it happens:

- Message queue uses “at least once delivery” to ensure no data loss.
- But if consumer fails to acknowledge, message is re-sent.

Solution:

Technique	Description
Idempotency	Make sure the operation has no effect if repeated (e.g., "order #123 already exists")
Deduplication IDs	Add a unique ID in each message; if it's already processed, skip it
Transactional DB Writes	Use DB transactions to avoid inserting duplicates

2. Message Loss

Problem:

- Some messages just disappear and are never processed.

Why it happens:

- Queue crashed
- Consumer crashed after picking the message but before completing it
- Message TTL (time to live) expired

Solution:

Technique	Description
Persistent Queues	Use disk-based queues (e.g., Kafka, RabbitMQ with durability enabled)

Acknowledgements	Message is only removed after consumer confirms it
Retries + Dead-Letter Queue (DLQ)	Unprocessed messages go to a separate queue for retry/debugging

(Disk-Based Queue wo hoti hai jo messages ko memory (RAM) me nahi, hard disk pe store karti hai.)

3. Message Processing Delay / Slow Consumers

Problem:

- Messages pile up because consumers are too slow to keep up.
- Queue becomes huge = latency and memory issues

Why it happens:

- Heavy logic inside consumer
- Too many messages, too few consumers

Solution:

Technique	Description
Horizontal Scaling	Add more consumer instances to handle load
Batch Processing	Process messages in groups, not one-by-one
Backpressure	Tell producer to slow down or pause if queue is full (like flow control)

4. Ordering Issues (Messages processed out of order)

Problem:

- Message 2 gets processed before message 1.
- Bad for systems where order matters (e.g., transactions, delivery updates)

Why it happens:

- Multiple consumers process messages in parallel

- Some consumers are faster than others

Solution:

Technique	Description
Partition-based ordering	In tools like Kafka, you can assign messages of the same key to same partition
Single-threaded consumers	For critical ordered data, process in one thread only
Sequence IDs	Attach sequence numbers in messages and re-order them before use (if needed)

(Har partition ke andar messages ordered (sequence mein) rehte hain.
Par different partitions mein order maintain nahi hota.)
(One thread mein hona ka matlab hai ki sirf ek kaam ek time pe ho raha hai — na parallel, na multi-tasking.)

5. Dead Consumers (Consumer crashes or hangs)

Problem:

- Consumer fails mid-way and message stays unprocessed
- System becomes stuck or data becomes stale

Why it happens:

- Bugs, crashes, memory leaks in consumer code

Solution:

Technique	Description
Message Acknowledgement Timeout	If consumer doesn’t ack within time, message is requeued
Heartbeat Monitoring	Detect if consumer is alive or needs to be restarted
Retry Mechanism	Reassign failed messages after delay or send to DLQ

6. Queue Overflow (Too many messages to handle)

Problem:

- Queue becomes full due to sudden spike in incoming messages
- New messages are dropped or delayed

Why it happens:

- Under-provisioned system
- No rate-limiting on producer side

Solution:

Technique	Description
Auto-scaling consumers	Add more consumers dynamically based on queue size
Rate-limiting Producers	Don’t allow more messages than queue can handle
Message Expiry (TTL)	Discard old messages that are no longer useful

7. Security Issues

Problem:

- Malicious users can send fake messages or flood the queue

Solution:

Technique	Description
Authentication & Authorization	Only allow trusted producers/consumers to access the queue
Message Signing	Encrypt and verify messages to ensure integrity
Rate-limiting	Prevent flooding or DoS (denial of service) attacks

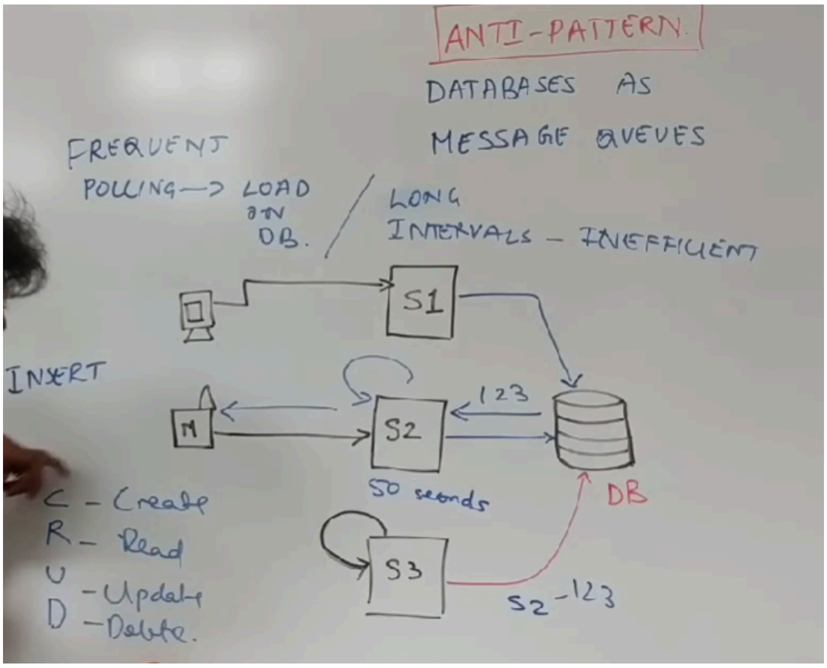
(DoS (Denial of Service) ek type ka cyber attack hota hai jisme attacker aapke server ya system ko itna busy kar deta hai ki normal users use nahi kar paate.)

Database as a Message Queue

Normally, hum **RabbitMQ**, **Kafka**, etc. use karte hain messages queue karne ke liye.

How Database as Queue Works:

Step	Action
1	Producer inserts a new row in **“Messages” table** — contains task/data
2	Consumer checks the table every few seconds (polling)
3	Consumer picks unprocessed rows (e.g., `status='pending'`)
4	Processes the message, updates status to `'done'` or `'failed'`
5	Done



Databases are often used to store various types of information, but one case where it becomes an a problem is when being used as a message broker.

The database is rarely designed to deal with messaging features, and hence is a poor substitute of a specialized message queue. When designing a system, this pattern is considered an anti pattern.

Maan lo ek system `M` koi kaam deta hai, jaise "ye task karo", aur wo task database ke table mein likh diya jaata hai. Ab jo kaam karne wale workers hain (S1, S2, S3), wo baar-baar database se poochh rahe hain: "kya koi naya kaam aaya?" — isse database par bahut load padta hai. Fir S2 ek kaam (jaise task 123) uthata hai aur bolta hai “main kar raha hoon”, lekin usse us kaam ko complete karne mein 50 seconds lag jaate hain. Tab tak ya to koi aur worker (S3) bhi wahi kaam utha sakta hai ya baaki system wait karte hain. Ye sab isliye ho raha hai kyunki kaam khud se push nahi ho raha — sab manually check kar rahe hain. Aur isi wajah se, jab hum database ko message queue jaise use karte hain, to wo system slow, confusing, aur galat ho jaata hai. Isi liye isse **“anti-pattern”** bola jaata hai — aisi cheez jo lagti sahi hai par reality mein system kharaab karti hai.

Here are possible drawbacks:
1) Polling intervals have to be set correctly. Too long makes the system is inefficient. Too short makes the database undergo heavy read load.
2) Read and write operation heavy DB. Usually, they are good at one of the two.
3) Manual delete procedures to be written to remove read messages.
4) Scaling is difficult conceptually and physically.

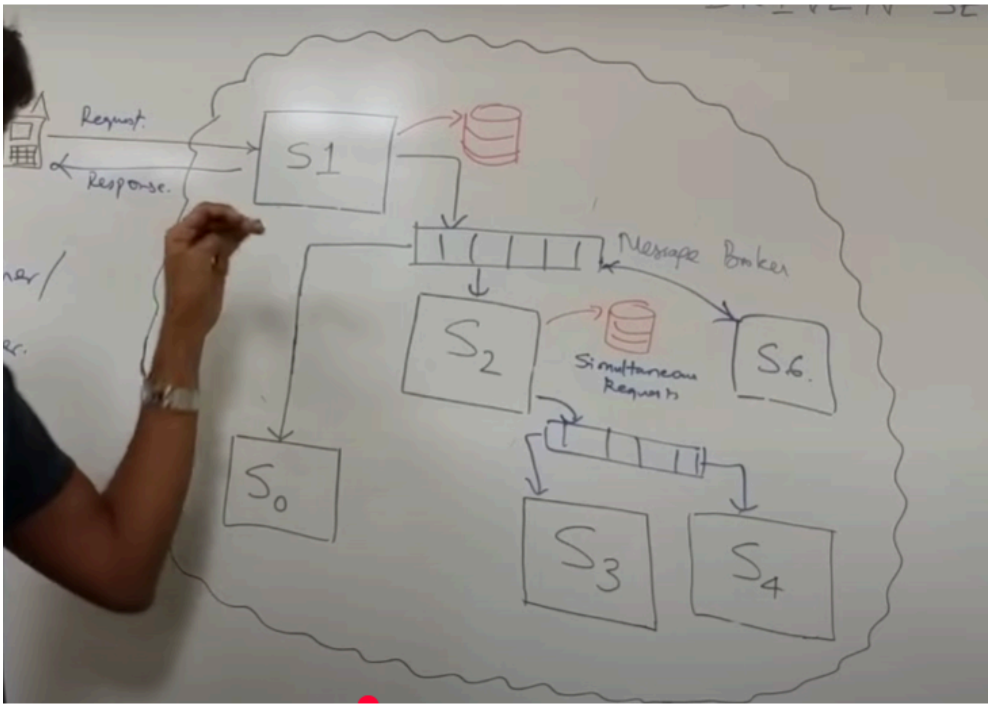
Disadvantages of a Message Queue:
1) Adds more moving parts to the system.
2) Cost of setting up the MQ along with training is large.
3) Maybe be overkill for a small service.

In general, for a small application, databases are fine as they bring no additional moving part to the system. For complex message sending requirements, it is useful to have an abstraction such as a message queue handle message delivery for us.

Public Subscriber Model

Consider a scenario of synchronous message passing. You have two components in your system that communicate with each other. Let's call the sender and receiver. The receiver asks for a service from the sender and the sender serves the request and waits for an acknowledgment from the receiver.

- There is another receiver that requests a service from the sender. The sender is blocked since it hasn't yet received any acknowledgment from the first receiver.
- The sender isn't able to serve the second receiver which can create problems. To solve this drawback, the Pub-Sub model was introduced.



The Pub/Sub (Publisher/Subscriber) model is a messaging pattern used in software architecture to facilitate asynchronous communication between different components or systems.

Publisher

The Publisher is responsible for creating and sending messages to the Pub/Sub system.

Publishers categorize messages into topics or channels based on their content. They do not need to know the identity of the subscribers.

Subscriber

The Subscriber is a recipient of messages in the Pub/Sub system. Subscribers express interest in receiving messages from specific topics.

They do not need to know the identity of the publishers. Subscribers receive messages from topics to which they are subscribed.

Topic

A Topic is a named channel or category to which messages are published. Publishers send messages to specific topics, and subscribers can subscribe to one or more topics to receive messages of interest.

Topics help categorize messages and enable targeted message delivery to interested subscribers.

Message Broker

The Message Broker is an intermediary component that manages the routing of messages between publishers and subscribers.

It receives messages from publishers and forwards them to subscribers based on their subscriptions.

The Message Broker ensures that messages are delivered to the correct subscribers and can provide additional features such as message persistence, scalability, and reliability.

Message

A Message is the unit of data exchanged between publishers and subscribers in the Pub/Sub system.

Messages can contain any type of data, such as text, JSON, or binary data. Publishers create messages and send them to the Pub/Sub system, and subscribers receive and process these messages.

Subscription

A Subscription represents a connection between a subscriber and a topic. Subscriptions define which messages a subscriber will receive based on the topics to which it is subscribed.

Subscriptions can have different configurations, such as message delivery guarantees (e.g., at-most-once, at-least-once) and acknowledgment mechanisms.

Pub/Sub Offers:

Problem	Pub/Sub Solution
Tight coupling	Services become **independent** (decoupled).
Scalability issues	You can add more subscribers **without changing the publisher**.
Fragile systems	If one service fails, others still continue.
Complex message routing	Central broker handles delivery.
Waiting for responses	Fully **asynchronous**, improves performance.

Drawbacks / Limitations of Pub/Sub

Drawback	Explanation
No control over order	Event order is **not guaranteed** unless carefully managed.
No acknowledgment = lost message	If subscriber fails to process, message might get lost (unless broker supports retries, like Kafka).
Difficult debugging	Hard to trace where a message went or who processed it.
Duplication risk	Same event might be processed **multiple times** → needs idempotency.
Delayed processing	No guarantee of instant reaction; delays can occur.
Requires good design discipline	Badly implemented pub/sub can become **event spaghetti** — hard to maintain.

When to Use Pub/Sub Model

1. **Multiple services need to react** to the same event
   (e.g., Order placed → Billing, Shipping, Email, Analytics)
2. **Loose coupling is needed**
   (Publisher doesn't know or care who the subscribers are)
3. **System needs to scale easily**
   (Add new services without changing existing code)
4. **Asynchronous processing is okay**
   (You don't need an immediate response)
5. **You want high flexibility and modular design**
6. **You have frequent updates or events to broadcast**
   (e.g., stock price updates, notifications, logs)

When NOT to Use Pub/Sub Model

1. **Only one or two simple actions are needed**
   (No need to notify multiple services)
2. **You need an immediate response**
   (e.g., Login → Show dashboard right away)
3. **Your app is small or has simple workflows**
   (E.g., basic to-do app, single backend service)
4. **You want easier debugging and monitoring**
   (Pub/Sub adds complexity)
5. **You don't want to manage extra infrastructure**
   (Message broker like Kafka or RabbitMQ adds setup and cost)
6. **Your team is not familiar with async/event-driven design**

There are two types of Pub/Sub services: **Standard Pub/Sub** and **Pub/Sub Lite**.

The **Standard Pub/Sub** service is what most people use. It's very reliable — it makes sure that messages reach all the right subscribers safely. It also connects easily with other tools and services, and you don't need to worry about scaling — it automatically handles heavy loads. Plus, it stores your messages in multiple locations, so even if something goes wrong in one place, your data is still safe.

On the other hand, **Pub/Sub Lite** is a cheaper version. It's good if you're trying to save money, but it comes with some downsides. It's not as reliable, and you have to manage things like storage and speed by yourself. Also, it only stores your data in fewer places, so there's a bit more risk. Still, it's a good choice for projects where perfect delivery isn't critical.

Pub/Sub vs. Other Messaging Technologies

1. Pub/Sub vs Message Queue (RabbitMQ)

- **Message Queue:**
  Ek message → sirf ek consumer ko milta hai
  "Ek parcel ek hi delivery boy ko diya."
- **Pub/Sub:**
  Ek message → sab interested services ko milta hai
  "Ek news sabko loudspeaker pe suna di."

2. Pub/Sub vs Kafka (Streaming Platform)

- **Kafka:**
  Continuous data stream + messages store bhi hote hain
  "Live CCTV recording jahan aap pura past bhi dekh sakte ho."

- **Pub/Sub:**
  Real-time event alert, storage short-term
  “Ghanti baji, sab alert ho gaye — bas kaam khatam.”

3. Pub/Sub vs WebSockets

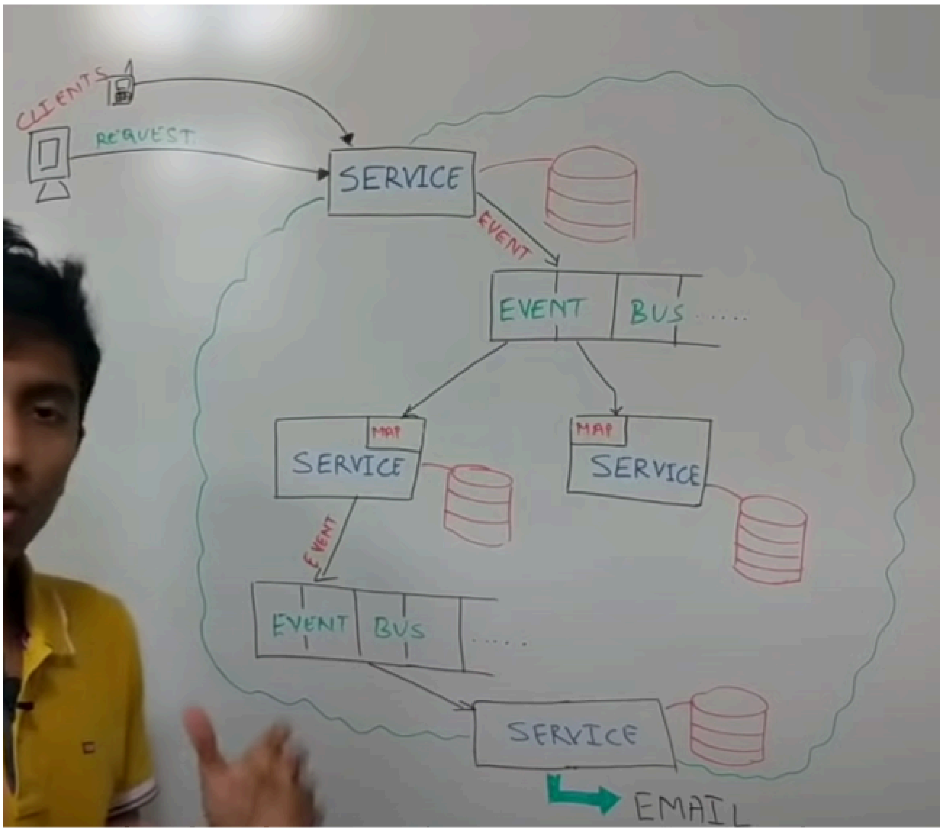
- **WebSockets**:
  2-way live talk hoti hai (client ↔ server)
  “Chat app jahan dono message bhej sakte hain.”
- **Pub/Sub:**
  1-way broadcast hoti hai (publisher → many subscribers)
  Teacher ne announce kiya, sab students ne suna.”

4. Pub/Sub vs HTTP APIs

- **HTTP API:**
  Request bhejte ho → wait karte ho reply ka
  “Call kiya, jab tak ‘hello’ na aaye, ruke rahe.”
- **Pub/Sub:**
  Message bhejte ho → turant nikal jaate ho
  “WhatsApp pe message drop kiya, saamne wala jab chahe padhe.”

EOT

Event Driven Architecture



Basically something has happened that concerns me and if it concerns anyone else you can consume this event and see if this even is relevant to them .

Git, it used this event driven architecture .

An **event-driven system** is a software architecture where:

- A service doesn't directly call another.
- It just **publishes an event** (e.g., "Order Placed").
- Whoever is **interested in that event** listens and reacts.

Isme **Pub/Sub model** ka use hota hai — **event ko broadcast** kiya jaata hai, aur jo service sun rahi hai vo kaam kar leti hai.

Client ne ek request bheji, jaise ki ek user ne online order place kiya. Yeh request sabse pehle ek **main service** ke paas jaati hai jo order ka data database mein save kar leti hai. Order save hone ke baad, yeh service ek **event generate karti hai** — jaise "OrderPlaced" — aur ise bhejti hai **Event Bus** ko. Event Bus ek **messenger** ki tarah kaam karta hai, jo is event ko sab un services tak pahucha deta hai jo is event ko sun rahi hoti hain. Jaise hi yeh event aata hai, **Billing Service** invoice banati hai, **Inventory Service** stock update karti hai, aur sab apne-apne database mein kaam save karte hain. Inmein se kuch services phir **naye events generate karti hain**, jaise "InvoiceGenerated" ya "StockUpdated", jo **next Event Bus** tak jaate hain. Finally, ek service hoti hai jo in events ko sunke **confirmation email bhej deti hai** user ko. Poora system ek chain reaction ki tarah kaam karta hai — bina kisi direct call ke, sab apne kaam independently karte hain, bas events ke through interconnected hote hain.

Event Driven Systems pass and persist events. They have evolved from the publisher-subscriber model, and the design has some advantages. Events are immutable and can be replayed to allow the systems to take snapshots of their behavior. This allows services to 'self heal'.

Events are stored in a **message queue** (like Kafka or Pub/Sub).

In event-driven systems, sometimes a message (event) might fail to reach the service that should handle it — due to network errors, server crash, etc. To fix this, systems use **idempotency** and **retry logic**.

What does that mean?

- **Idempotency**: Even if the same event is processed multiple times, the result remains the same.
  Example: If a user is charged once, repeating the event **won't charge again**.

- **Retry logic**: The system keeps **sending the message again and again** until the other side says, "I got it!" (acknowledgment).

  This makes sure that no message is lost.

Components of EDA

Component	Simple Meaning
Event Source	Something that **creates an event** (user click, sensor update, service action)
Event	A **message that says something happened**, like "OrderPlaced"
Event Bus / Event Broker	The **middleman** that passes events from publishers to subscribers
Publisher	The component that **sends** (publishes) the event

Subscriber	The component that **listens** to the event and takes action
Event Handler	The **code logic** that runs when a subscriber receives an event
Dispatcher	Routes events to the correct handler (optional, advanced systems)
Aggregator	Combines **many small events into one big event** (e.g., 5 sensor readings → 1 summary)
Listener	Constantly **watches the event bus** for certain events to react to them

Use Event-Driven Architecture when:

- You need **real-time reactions** (e.g. chat, online orders)
- Your system has **many independent services**
- You expect the system to **scale or grow**
- You want to **integrate different services** (e.g. external APIs, databases)
- You're building **IoT apps, gaming, e-commerce, or finance systems**

Benefits of EDA

- **Loose Coupling**: Services don't need to know each other
- **Real-time processing**: React to events instantly
- **Flexible & Scalable**: Add/remove services easily
- **Modular**: Easy to develop and test small parts separately
- **Retry Logic**: Failed events can be retried till successful
- **Focus on business logic**, not message delivery

Challenges of EDA

- **Hard to understand the flow** — can't track what happens after an event easily
- **Debugging is tricky** — many services may react to a single event
- **Event order issues** — events may arrive late or out of order
- **Latency risk** — slight delays if event traffic is high
- **System becomes complex** as more events & services are added

Event-Driven vs Message-Driven Architecture (EDA vs MDA)

Feature	Event-Driven (EDA)	Message-Driven (MDA)
Focus	*What happened* (events)	*What to do* (commands/messages)
Trigger	Triggered by system state changes	Triggered by request for action
Examples	"OrderPlaced", "PaymentReceived"	"SendEmail", "ProcessOrder"
Style	Asynchronous, reactive	Can be synchronous or asynchronous
Use Case	Reactions to real-world or system events	Service-to-service communication
Decoupling	High (very loose coupling)	Medium (still decoupled, but closer connection)

EOT