System Design

High Level Design

→ Network Protocols and System Design: Client-Server vs. Peer-to-Peer, WebSocket vs. WebRTC

Client Server uses HTTP, FTP, SMTP and Web Sockets. Peer to Peer uses WebRTC. Client Server is a centralized architecture where clients talk (request) to server, and servers gives response. It's a one way communication.

WebSocket is bi-directional (two way communication) i.e. server can also initiate talk with the client. WebSocket is used when we want to design messaging app like WhatsApp and Telegram etc.

HTTP is a client to server protocol. It is connection oriented and we access web pages in this.

FTP is not used generally because the data is not encrypted in this.

SMTP is used to send the email and it's generally used with IMAP which reads/access the email from server.

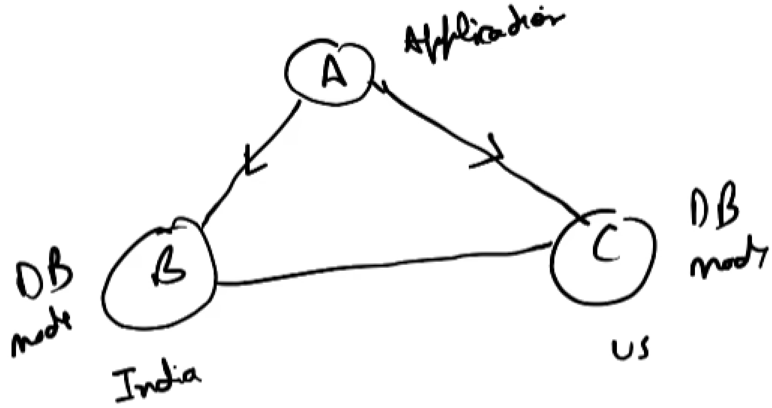
TCP and UDP are used to transport/transfer the data.

In TCP/IP, we create a virtual connection to send (transfer) the data packets (they have sequences like 1, 2, 3 etc.) to the server. So, it can maintain ordering of these packets and it sends the acknowledgement when packet is received. If it misses the acknowledgement for packet 3 then it resends.

In UDP/IP, the data packets are sent parallelly. No ordering is maintained in this. UDP is fast because there is no acknowledgement, no ordering and no connection is maintained. UDP is used in live streams, or video calling etc.

Peer to Peer is when server and clients can all talk to each other. WebRTC is Peer to Peer. They are fast as they use UDP to transfer data.

→ Understanding the CAP Theorem in Distributed Systems



Application A can query from server B or C.

In any distributed data system, it's impossible to simultaneously guarantee all three of the following:

- **Consistency (C):** Every read receives the most recent write or an error.
  If A writes a=5 then B will replicate data to DB node C. So, when read happens using C then it should be same (a=5).
- **Availability (A):** Every request should receive a (non-error) response, without guarantee that it contains the most recent write.
- **Partition Tolerance (P):** The system continues to operate despite arbitrary partitioning due to network failures.
  If for example, there happens a communication breakage during replication. Application can still query (request) the server and get a response back. Basically system is up.
- In real work we don't wanna trade-off (**P**). In the presence of a network partition, a distributed system must choose between consistency and availability.

→ Microservices

What are Microservices?

Microservices is an architectural style where an application is broken into smaller, independent services that work together.

Why it matters today:

As apps grow more complex, this style helps teams move faster, scale better, and deploy independently.

Monolithic vs Microservices:

Monolithic Architecture: One big codebase; all features are tightly connected.

Microservices Architecture: Each feature/module is a separate service that can be updated or deployed on its own.

❌ Disadvantages of Monolithic Architecture

1. **Scalability Issues:**

   If one part of the app gets heavy traffic, the whole system might need to scale unnecessarily, increasing load times and costs.

2. **Hard to Maintain & Deploy:**

   Even a small change means redeploying the whole app — risky and time-consuming.

3. **Tight Coupling:**

   Components depend too much on each other; a change in one area can break others.

✅ Advantages of Microservices

1. **Service Division:**

   Breaks a big app into smaller parts (like user service, product service, etc.) — each with its own purpose.

2. **Scalability & Flexibility:**

   You can scale only the services you need — for example, just the login service during heavy traffic.

3. **Independent Deployment:**

   One team can deploy updates to a service without affecting the whole app.

⚠️ Challenges of Microservices

1. **Service Decomposition:**

Deciding *how* to split services can be tricky. If done poorly, services might talk to each other too much.

2. **Transactions Across Services:**

   Keeping data consistent across multiple services (like when placing an order) can get complex.

3. **Monitoring Multiple Services:**

   More services = more logs, more things to track, harder to debug.

🔁 Integration Patterns

- **Communication Between Services:**

  Could be REST APIs, message queues, or event-driven architecture (like using Kafka).

- **Database Management:**

  Typically, each service has its own database to ensure loose coupling — but this can make querying data across services harder.

🌟 Best Practices

1. **Understand Business Capabilities:**

   Design services based on what the business needs, not just technical boundaries.

2. **Keep Services Small:**

   Small enough to manage easily, but big enough to be useful on their own.

3. **Monitoring & Troubleshooting:**

   Use centralized logging, distributed tracing, and monitoring tools (like Prometheus, Grafana, or ELK stack) to keep track of everything.

🚀 Phases in a Microservices Journey

When a team moves from monolith to microservices, it typically goes through several phases:

1. **Understanding the Monolith**

- Analyze the existing monolithic application.
- Identify tightly coupled modules and dependencies.
- Understand how features are grouped together.

2. **Service Identification & Decomposition**
   - Find logical boundaries for splitting.
   - Choose how to break it up — either by business functions or subdomains (more on that below).
3. **Building Independent Services**
   - Develop services that can run, deploy, and scale independently.
   - Each service should have its own logic and database.
4. **Implementing Communication Mechanisms**
   - Decide how services talk to each other — REST, gRPC, messaging (Kafka, RabbitMQ, etc.)
5. **Deploying and Monitoring**
   - Use CI/CD pipelines for deployment.
   - Set up tools for logging, monitoring, and tracing service calls.
6. **Evolving and Scaling**
   - Gradually migrate more parts of the monolith.
   - Improve, refactor, and scale individual services as needed.

🧩 Decomposition Patterns

These are strategies or “ways” to break a large system into microservices:

1. Decomposition by Business Capability :

Imagine you are building an **E-commerce Platform** like **Amazon**.

You would divide your microservices based on **business capabilities** (what the business needs functionally):

- **User Service** – handles user registration, login, profile management.

- **Product Catalog Service** – manages products, categories, and product search.
- **Order Service** – handles shopping carts, checkout, order placement.
- **Payment Service** – manages payment processing, refunds.
- **Shipping Service** – manages delivery, tracking shipments.
- **Notification Service** – sends emails, SMS, push notifications.

👉 Here, each service matches a **core business function**.

This way, different teams can work independently on users, payments, shipping, etc.

---

2. Decomposition by Subdomains (Domain-Driven Design - DDD):

You apply **subdomain decomposition** based on **bounded contexts** within the domain:

- **Core Subdomain** (most important to business):
  - **Ordering Subdomain** → Manages order lifecycle (cart → checkout → order placed).
- **Supporting Subdomain** (supports the core but isn't core itself):
  - **Inventory Subdomain** → Tracks stock levels and warehouse data.
  - **Payments Subdomain** → Deals with payment authorization and processing.
  - **Shipping Subdomain** → Organizes delivery and logistics.
- **Generic Subdomain** (common tasks, can even use third-party tools):
  - **Authentication Subdomain** → Manages login, registration (could use something like OAuth/Identity Providers).

👉 Here, decomposition happens based on **domain knowledge**, focusing on how different parts of the business work internally, not just services.

Bounded contexts are isolated to prevent confusion between, say, payment logic and order logic.

→ Strangler Pattern, CQRS, SAGA Pattern

Strangler Pattern is used when we are refactoring our code from monolithic to microservices. We start creating services and we test those created services by sending traffic to them using controller like 10% to microservice and check if they are performing well. And at last we sends the 100% traffic to microservice and strangling the monolithic part.

Data Management in Microservice -

1. **Database for each individual microservice**
   We can scale only a single service and increase only the database of that service. Modification is easy in this because the data does not impact the other service database.

   Issues:
   Easy part of Shared database is hard for this.

2. **Shared Database**
   Query Join and maintaining transactional property (ACID) is easy in this.

   Issues:
   In shared database, if we want to scale one service we have to increase the capacity of whole database. If we want to modify our database, so we have to check if adding or deleting anything in database impact any other service.

SAGA is used for Transactional property (ACID property) because it is difficult to maintain the ACID property in the individual database, when the query (request) should go from one service to another service. So SAGA makes it easier for us to maintain the Transactional property in all of the database (Order service update it's local DB and then request will go to the Payment Service and update its database and so on...).
One service
publish an event and other service listen that event and check if there was a failure then it will send failure event back. So it will rollback the database.

CQRS (Command Query Request Segregation) help in query JOIN like tables in different database. It creates a view which have both the tables in this and then can JOIN them together.

→ Overview of Scaling Applications

- **Goal:**

  How do you grow an application from 0 users to 1 million users without it crashing or slowing down?

- **Focus:**

  Start small (single server) → Introduce smart scaling → Use load balancers and database tricks to handle growth.

🛠️ Initial Setup

- **Starting Point:**

  Just one server — handles everything: app code, database, business logic, etc.

- **Reality:**

  This is fine for college projects or very early-stage startups where traffic is low.

🏛️ Application Architecture

- **Separation of Responsibilities:**
  - **Business Logic** should not mix with server management tasks (like routing requests).
  - Separate layers: presentation layer, business logic layer, data layer.

- **Why:**

  Cleaner code, easier maintenance, and better scalability later.

⚖️ Load Balancing

- **What is a Load Balancer?**

  A tool that splits incoming traffic across multiple servers.

- **Why:**

  Prevents one server from getting overwhelmed.

- **Key Concepts:**
  - **Server Redundancy:** Always have extra servers ready.
  - **Failover:** If one server crashes, the load balancer shifts traffic to healthy servers.

🗄️ Database Replication

- **Master-Slave Architecture:**
  - **Master DB:** Handles write operations.
  - **Slave DBs:** Handle read operations (copies of the master).
- **Why Use It:**
  - Better performance (read-heavy apps benefit).
  - Increased fault tolerance (if master fails, slaves can take over).

⚡ Caching Strategies

- **What is Caching?**

  Saving copies of data temporarily to reduce database load.

- **Examples:**
  - Cache popular product details, user profiles, etc.
- **Cache Expiration:**
  - Set rules for when cache data should expire or refresh (to avoid stale data).

🌍 Content Delivery Network (CDN)

- **What is a CDN?**

  A network of servers worldwide that store and deliver static content (images, videos, etc.).

- **Why:**

- Speeds up access by serving content from the server closest to the user.
- Reduces main server load.

- **Latency Fix:**

  If users are in India and your server is in the US, a CDN can deliver content from an India-based server.

📩 Messaging Systems

- **Why Messaging?**

  Some tasks don't need to happen immediately (e.g., sending email notifications).

- **Solution:**

  Use messaging systems like **RabbitMQ** or **Kafka** for **asynchronous** processing.

- **Benefit:**

  Main system remains fast and responsive while background jobs get done later.

🛢️ Database Scaling

- **Vertical Scaling (Scale Up):**
  - Add more CPU, RAM to a single database server.
  - **Limitations:** Eventually, a single machine's power runs out.
- **Horizontal Scaling (Scale Out):**
  - Add more database servers and split the data across them (sharding).
  - **Better long-term:** No single point of failure and almost unlimited growth.

🏁 Implementation of Scalability

- **Real-World Scaling:**

  Not just about handling more users, but also:

  - Keeping performance fast.

- Maintaining security.
- Managing operational costs.

- **Challenges:**
  - More infrastructure = more complexity.
  - Monitoring and disaster recovery become critical.

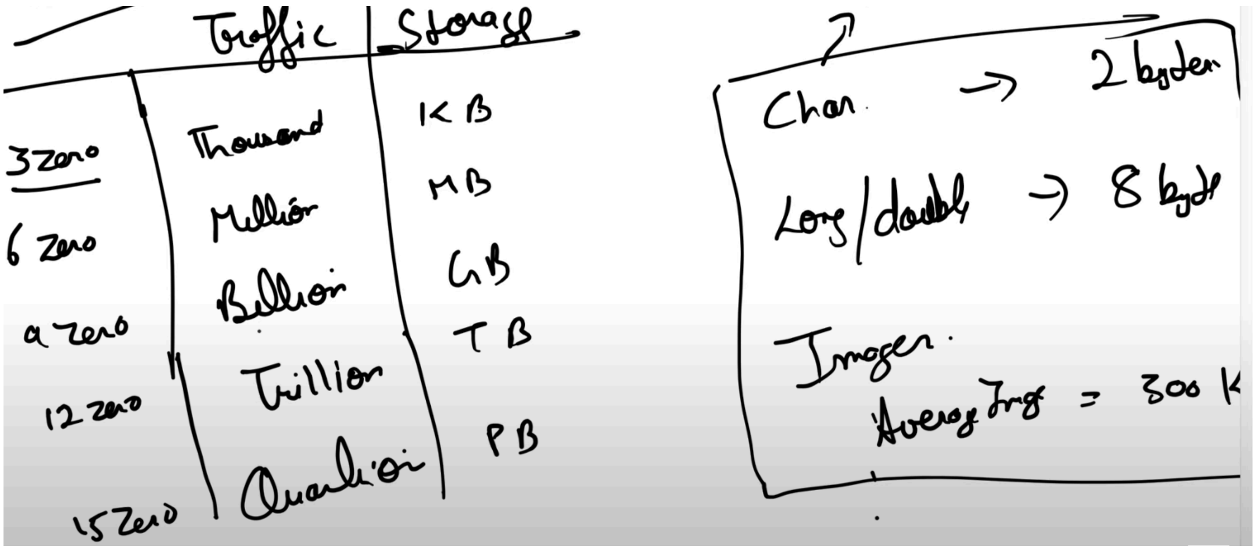
→ Consistent Hashing

Hashing is the process of giving a key to a hash function and it gives a value back. Mod Hashing is used when the number of nodes are fixed or static. Like we provided a key (key = "Avish") and hash function give value 20. Then we can do this 20 % 3, where 3 is fixed number of nodes. So we use consistent hashing for dynamic nodes (DB nodes/ server nodes).
If we have 3 nodes and then 4th node is added then 20 % 4 will give give another node (for eg. node 2, but before it gives node 1 when there were 3 nodes). When we want to access the data that was stored in node 1 now we search in node 2, but we don't find it there. So now we have to rebalance node 1 and node 2.

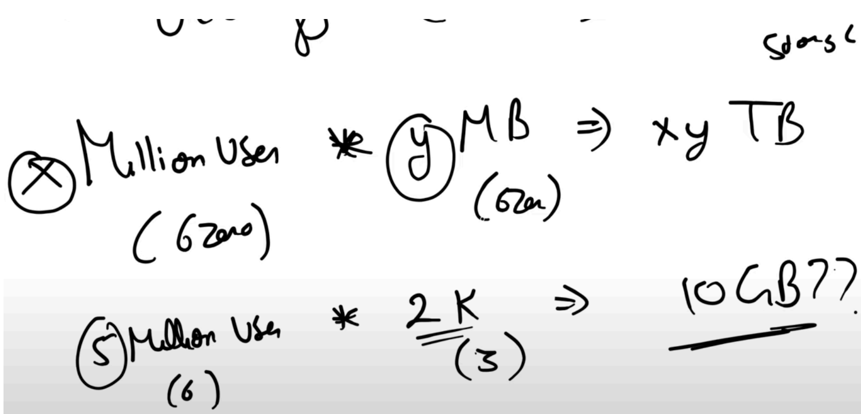
Consistent hashing is a distributed hashing technique that places servers and keys on a circular hash ring using a hash function. It helps distribute load evenly and minimizes the number of keys that need rebalancing when servers are added or removed, making it ideal for scalable systems. By mapping requests to the next available server in a clockwise direction, it ensures efficient data distribution and supports use cases like load balancing and distributed caching. Its main advantages include scalability and reduced rebalancing, though challenges like uneven data distribution and performance issues can arise if not managed properly.
It have a
limitation when the server/nodes are in continuous way like at 3, 4, 5 position. Now server1 at 3 will fulfill all the keys. We can solve this by making virtual objects which means replicate server at different points like server1 at 3, 7, 9 and server2 at 4, 8, 10 etc.

→ Back-of-the-envelope Estimation

Back-of-the-envelope estimation is a quick, high-level approach used in system design to guide decisions and avoid resource overestimation. We have to decide things like CDN, Load Balancer, Storage, Cache (RAM) and how many servers are needed for System Design. It's especially useful in early planning, relying on rough, simplified assumptions like round numbers. The process involves estimating traffic based on user activity, calculating storage needs for posts and media over time, and determining RAM and server requirements for caching and handling load. While estimates are imprecise (rough figure), they help frame realistic designs. Additionally, understanding CAP theorem trade-offs—often favoring availability and partition tolerance in large-scale systems—is crucial for balanced architecture.



Calculation using above cheat sheet:



→ SQL vs NoSQL

SQL Databases:

Uses structured tables with a fixed/predetermined schema and store complete data in a centralized way. They scale vertically (adding more resources to a single server) and follow ACID properties for strong consistency and integrity, making them ideal for systems with relational data and strict consistency needs, like financial applications.

NoSQL Databases:

Include key-value stores, document, column-wise, and graph databases. They are distributed by nature, support horizontal scaling (adding more servers), and follow BASE properties for high availability with eventual consistency. They suit systems that handle large, dynamic, or semi-structured data where performance and scalability matter more than immediate consistency.

Key-value DB like DynamoDB can do query/search based on the key only but Document DB like MongoDB can do query based on both key and value. They both store data in key-value pair. Key value DB can contain values like String, Integer, JSON etc. and Document DB contain values as JSON or XML.

Column wise DB store data in key value but the value of keys can be dynamic i.e. one key have 3 columns data while other key have only 2 columns data. Graph DB store data in nodes and edges.

When to Use SQL:

Flexible queries, relational dependencies, and strict data integrity are required (e.g., financial applications)

When to Use NoSQL:

High availability and performance needed; can handle large, dynamic datasets; some inconsistency is acceptable.

→ WhatsApp System Design

The WhatsApp system design focuses on building a scalable, real-time chat application with core features like one-to-one messaging, group chats (up to 200 members), media sharing, read/delivered receipts, and last seen visibility. Key components include the Gateway Service for message routing, Session Service for managing user sessions, and Group Service for handling group members. Messages flow from sender to recipient via these services, with receipts confirming delivery and read status. Technical challenges include implementing WebSockets for real-time updates, managing user presence, ensuring message ordering and idempotency, and optimizing memory use. Load balancing, service discovery, and consistent hashing help manage high traffic and group dynamics. Additional measures like retry mechanisms, message queues, and prioritizing critical messages ensure reliability under heavy load. Understanding this architecture is essential for designing effective chat systems.

→ Rate Limiter

Rate limiting is a way to control how many requests a user or system can make to a server in a certain period, helping to prevent overload or abuse. There are several common algorithms to do this:

- **Token Bucket**: Think of a bucket that slowly fills with tokens. Each request needs a token. If the bucket is empty, the request is blocked. This allows for bursts of traffic if enough tokens are saved up.
- **Leaky Bucket**: Similar to Token Bucket, but here, water (requests) drips out at a constant rate. It smooths out traffic by processing requests steadily, even if they come in bursts.
- **Fixed Window Counter**: Counts how many requests are made in a set time window (like every minute). If the number goes over the limit, extra requests are blocked. It’s simple but can allow bursts at the edge of windows.
- **Sliding Window Logs**: Stores the exact time of each request and checks how many were made in the recent time period (like the past 60 seconds). It’s more accurate but needs more memory to track all timestamps.
- **Sliding Window Counter**: Combines the fixed window with smoothing by dividing the window into smaller parts, offering better accuracy with less memory use than logs.

A high-level design of a distributed rate limiter involves placing a centralized or decentralized rate-limiting service between users and backend servers. Each incoming request is first checked against the rate limiter, which stores request counts or tokens in a fast, shared storage like Redis. The system uses a consistent algorithm (e.g., Token Bucket or Sliding Window Counter) to decide whether to allow or reject the request. For distributed environments, rate limiter nodes synchronize counters using shared storage or replicate state across nodes to maintain accuracy and avoid abuse, ensuring consistent rate enforcement across multiple servers or regions.

→ Idempotent POST API

Concurrency and idempotency are crucial for building reliable systems. Idempotency ensures repeated requests—whether due to retries or user error—don't cause unintended side effects, like duplicate entries. Concurrency deals with multiple users accessing the same resource, such as booking the same seat, which can lead to conflicts.

By default GET, PUT, DELETE are idempotent in nature while we have to make POST request idempotent. Client should generate Idempotent Key (i.e. UUID). New IK should be generated for each different request.

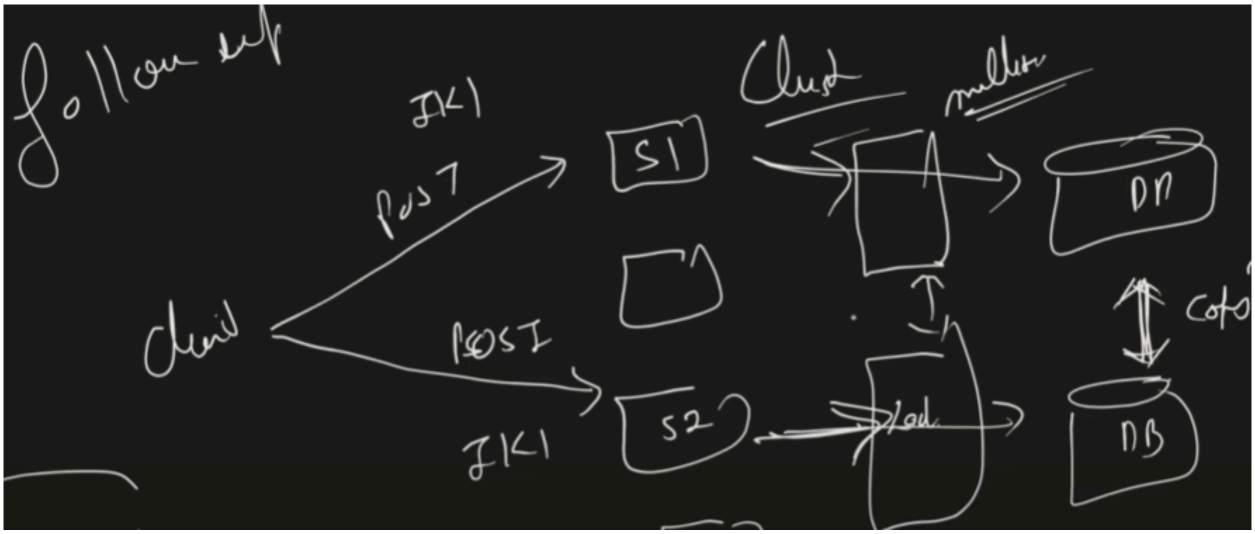
Duplicate request can come as sequential or parallel. The POST request (like add item in cart) flow/steps:

1. Generate IK.
2. Set the IK in the request header and then calls server.
3. Validation: Is IK present in Header? If no, HTTP 400 validation error.
4. If yes, Server will read IK from DB.
5. If IK present in DB? If no, put an entry in DB as with Key name and Status i.e. created or consumed. After consumed it can't be reused.
   If yes, check the IK status. If status is consumed then send HTTP 200 and if created it means previous/original request is still working. So send conflict code HTTP 409.
6. If operation is success i.e. resource is created then it will change the status of IK to consumed/claimed and return HTTP 201.

To handle parallel duplicate request, systems use unique identifiers for operations, mutex locks or synchronization for critical sections, and return existing data for duplicates instead of creating new entries.

Q. What if Client send the duplicate POST request to Server1 and Server3 with same IK and these server have local DB? How to maintain idempotency in this case?

The answer is using Cache. Cache sync is very faster as compared to DB sync.



→ Design High Availability & Resilience System

High availability architecture ensures systems remain operational with minimal downtime, often targeting 99.999% uptime and eliminating single points of failure.

Single Node Architecture:

Single node has a single point of failure. Failure in the database (DB) results in total application down.

Multi-Node Architecture:

Active-Passive Model:

Requires at least two data centers (e.g., one in Mumbai, one in Pune). Only one data center is active (primary) which can handle read/write operations, while others act as replicas and read only DB. Requests can be routed to the disaster recovery (DR) data center if primary fails. Latency issues may arise when requests go through the DR data center.

Active-Active Model:

More efficient use of resources, both data centers can handle requests. Enables simultaneous read/write in both data centers. Complexity in synchronization and conflict resolution (e.g., simultaneous changes in both DBs).

Advantages and Disadvantages:

Active-Passive:

Advantages: Simpler design with lower complexity.

Disadvantages: Resource utilization may be poor; latency on DR requests; potential downtime during DB failover.

Active-Active:

Advantages: Better resource utilization; handles more traffic; both DBs can write/read.

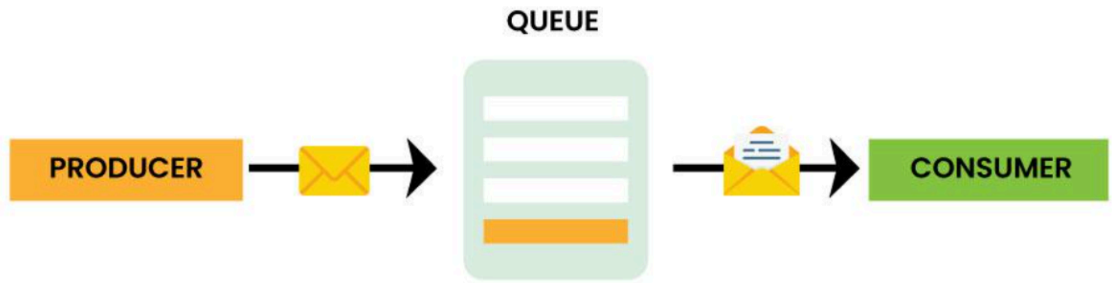
Disadvantages: Higher complexity in maintaining synchronization; potential conflicts in changes.

→ Distributed Messaging Queue

A **message queue** is a communication mechanism that enables asynchronous communication between different components or services in a system. It acts as a temporary storage for messages, allowing producers to send messages without waiting for consumers to process them immediately. This decouples the components, enhancing system resilience and scalability.

Diagram Overview:

- **Producers:** Components that generate and send messages to the queue.
- **Message Queue:** The intermediary that holds messages until they are processed.
- **Consumers:** Components that retrieve and process messages from the queue.
- **Acknowledgments:** Consumers send acknowledgments back to the queue to confirm message processing.



Advantages:

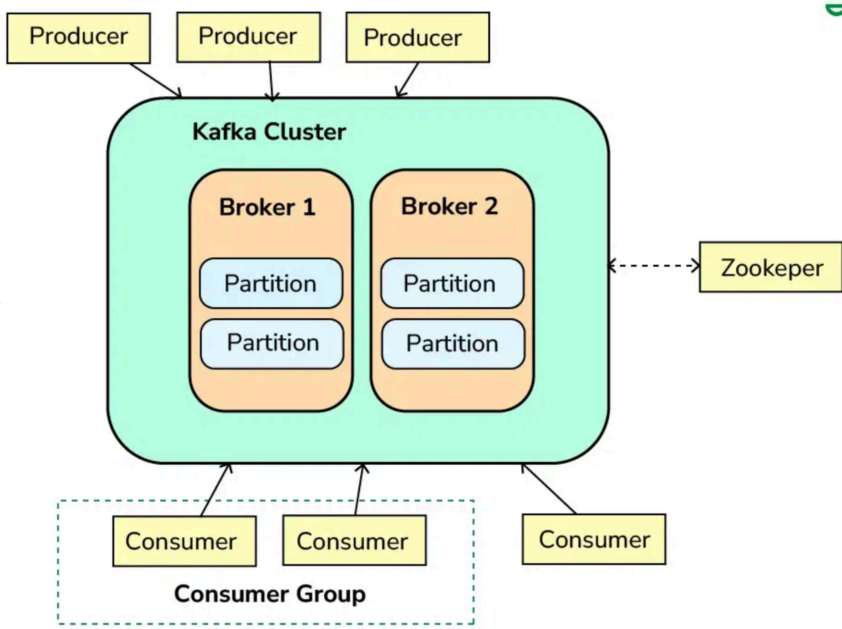
- **Asynchronous Processing:** Producers can continue their tasks without waiting for consumers to process messages.
- **Load Buffering:** Helps in managing varying loads by queuing messages during peak times.
- **Fault Isolation:** Isolates failures by ensuring messages are not lost and can be retried.
- **Scalability:** Facilitates horizontal scaling by adding more consumers to handle increased load.
- **Decoupling:** Allows independent scaling and maintenance of components.

Point-to-Point vs. Pub-Sub

- **Point-to-Point**: Each message is processed by one consumer.
- **Publish-Subscribe**: Messages can be broadcast to multiple consumers simultaneously.

Kafka

Kafka architecture includes producers, consumers, consumer groups, brokers (server), topics, partitions, offset, clusters and zookeeper, which manage message flow, storage, and consumption effectively, boosting throughput and scaling.

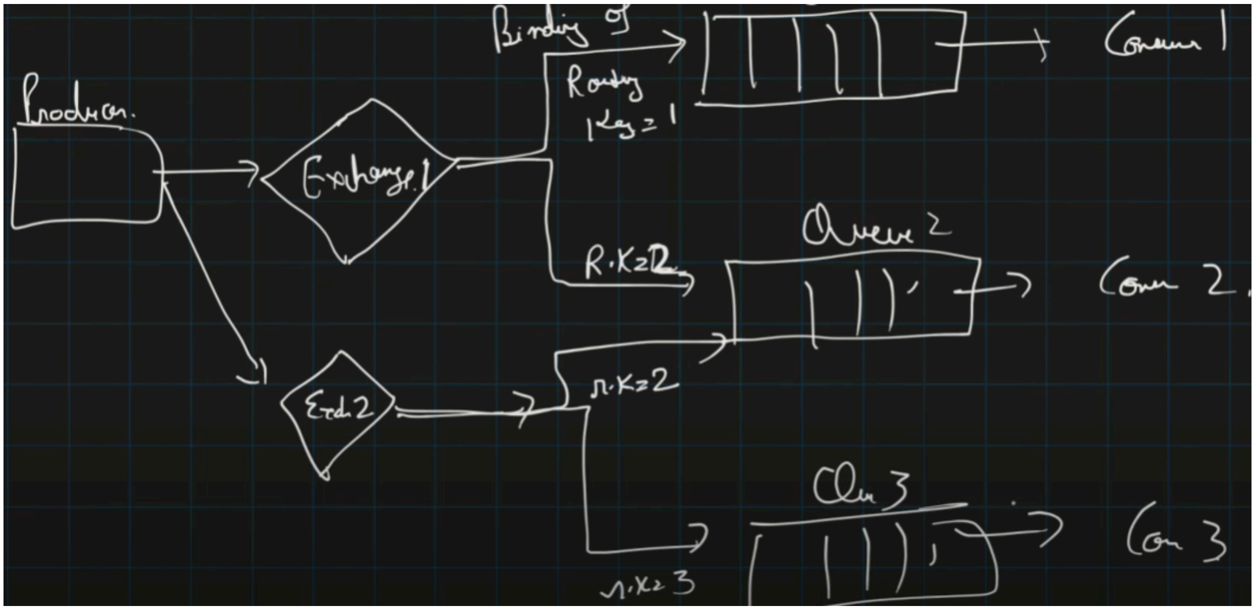


- **Producer**: An application that sends messages to Kafka topics.
- **Consumer**: An application that reads messages from Kafka topics.
- **Consumer Group**: A group of consumers that share the load of reading messages from topics.
- **Broker**: A Kafka server that stores data and serves clients.
- **Topic**: A category to which messages are sent by producers. Topic contains partitions.
- **Partition**: A division of a topic that allows Kafka to horizontally scale and parallelize processing. Partition contains offset. Within a particular consumer group different partition are read by different consumer. For example, partition1 can be read by consumer1 but can't be read by consumer2. But the partition1 can be read by the consumer in different consumer group.
- **Offset**: A unique identifier for each message within a partition, used by consumers to track their position.
- **Cluster**: A group of Kafka brokers that work together to handle data storage and processing.

- **ZooKeeper**: A centralized service used for managing/interact and coordinating Kafka brokers.

RabbitMQ

In this producer talks with an exchange. Exchange is bind with Routing Key.



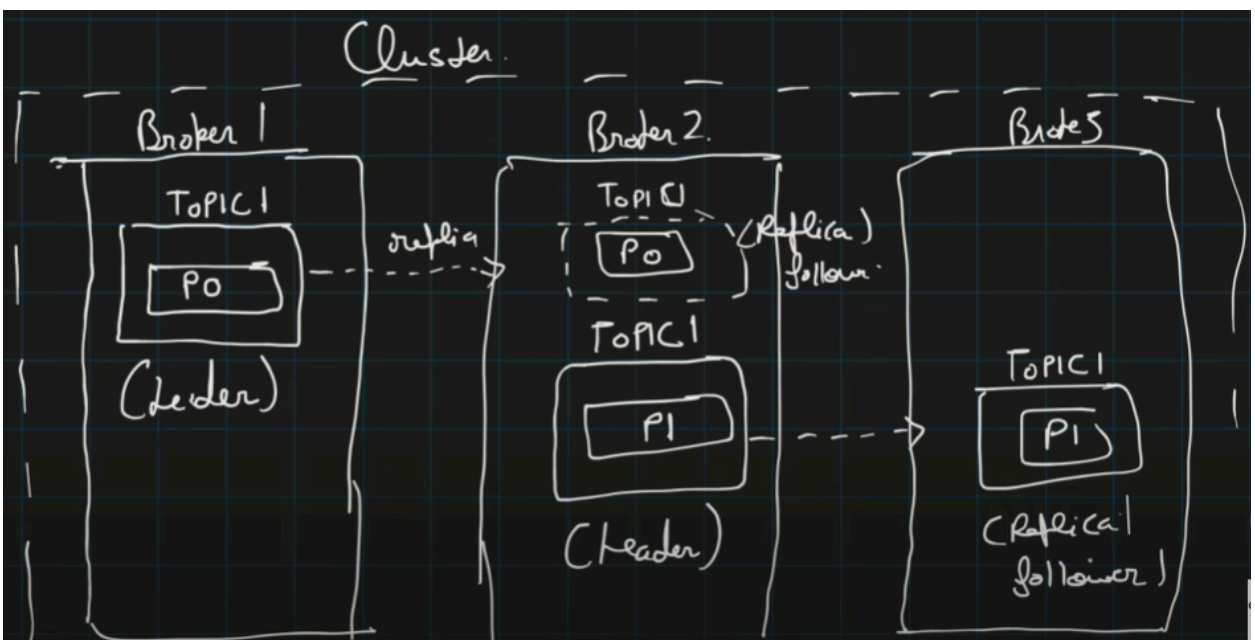
Types of Exchange:

1. Fan out: If a msg comes in exchange1 then the message is delivered to all the queue associated with exchange1.
2. Direct: In this exchange, message is associated with a key like key = 1 and then it compare the msg key and routing key. So it send the msg to queue1.
3. Topic exchange: We can use wildcard in the routing key like *123 and when the msg key is like xyz123 then it goes to that queue which have *123 routing key.

Important Points:

- A message have certain things/details like (* are mandatory) Key (id etc.), Value* (actual content), Partition and Topic*. If key is present it computes hash of it and then push data in the partition. If both partition and key is empty then it will follow round robin for pushing msg in partition offset.

- Consumers are grouped to ensure message load is balanced across multiple consumers. When one consumer fails, another in the same group can continue processing from the last committed offset (index till where the read is successful in partition offset like till index 2).
- RabbitMQ uses a push-based model where messages are actively sent to consumers, while Kafka employs a pull-based method where consumers request messages.
- Q. What happens when queue size limit is reached?
  We can add brokers to solve this.
- Q. What happens when consumer not able to process messages?
  If a message fails to process, it can be retried a specified number of times before being moved to a
  Dead Letter Queue for further investigation or reprocessing and increase the committed offset. While in RabbitMQ the msg is requeue which means it push the msg again in the queue.
- Q. What happens to messages when queue goes down?
  We have replica of the Leader i.e. Topic1 Partition0 and read/write happen through leader. If leader goes down then replica (follower) takes over and become leader.



→ Proxy

A proxy server acts as an intermediary between clients and servers, relaying requests and responses without direct communication between them. There are mainly two types: Forward Proxy and Reverse Proxy, differing primarily in the direction of communication.

Advantages of Forward Proxy

```
Client → Forward Proxy → Internet (Target Server)
```

Provides anonymity (hide clients IP), groups client same requests, restricts access to content (like in corporate accessing entertainment content), increases security, and allows caching of common data.

Disadvantages of Forward Proxy

Operates at the application layer requiring configuration for each application, potentially compromising performance.

Advantages of Reverse Proxy

```
Client → Reverse Proxy → Backend Servers
                          → Server 1
                          → Server 2
                          → Server 3
```

Provides security by hiding server IPs, distributes load among servers, caches data for quick retrieval, and reduces latency. CDN is a reverse proxy.

Proxy vs. VPN

While both provide anonymity, a VPN also encrypts data and creates secure tunnels, which a proxy does not.

Proxy vs. Load Balancer

Reverse proxies can act as load balancers, distributing traffic, while a load balancer alone does not provide anonymity or caching.

Proxy vs. Firewall

While both manage network traffic, a proxy controls and forwards requests between clients and servers, often adding caching and anonymity, whereas a firewall blocks or permits traffic based on security rules to protect the network from threats.

→ Load Balancing & Different Algorithms

Load Balancer distribute client requests among multiple servers to prevent overloading a single server.

Types of Load Balancers

Layer 4 (L4) Load Balancer

Operates at the transport layer; uses IP address and TCP/UDP port for routing.

Layer 7 (L7) Load Balancer

Operates at the application layer; can read HTTP headers, session data, and perform caching and used these for routing to server.

Load Balancing Algorithms

Static Algorithms:

Round Robin:

Requests are distributed sequentially among servers.

Advantages: Simple implementation, equal load distribution.

Disadvantages: Does not consider server capacity. So low capacity server might go down.

Weighted Round Robin:

Assigns weights to servers based on capacity. Server with higher weight can take more request.

Advantages: Balances load based on capacity.

Disadvantages: Still vulnerable to high processing requests that takes much time.

IP Hash:

Uses the client's IP address and calculate hash to assign requests.

Advantages: Ensures the same client connects to the same server.

Disadvantages: Can lead to unequal distribution if requests come through a proxy.

Dynamic Algorithms:

Least Connection:

Sends requests to the server with the fewest active connections.

Advantages: Reduces the chance of overburdening a server.

Disadvantages: Does not differentiate between high and low traffic.

Weighted Least Connection:

Combines weights with active connections for distribution. Calculate the Ratio of No. of Active Connections to its Weight. Server with minimum ratio get the request.
Eg. Server1 have W = 10 and AC = 2, Server2 have W = 1 and AC = 1. When a new request comes it calculate ratio 2/10 and 1/1. Server1 ratio is less so request goes to Server1.

Least Response Time:

Directs requests based on the server's response time (TTFB). Time to First Byte is time interval b/w sending a request and receiving the response from the server.

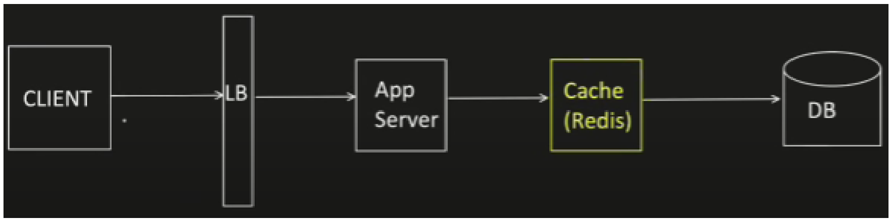
Picks the server which has less (Active connection * TTFB) and if clash, follow round robin.

→ Distributed Cache & Caching Strategies

Caching is a technique to store frequently used data in a fast access memory (RAM) rather than accessing data every time from slow access memory. It makes our system fast, reduce the latency and achieve fault tolerance.

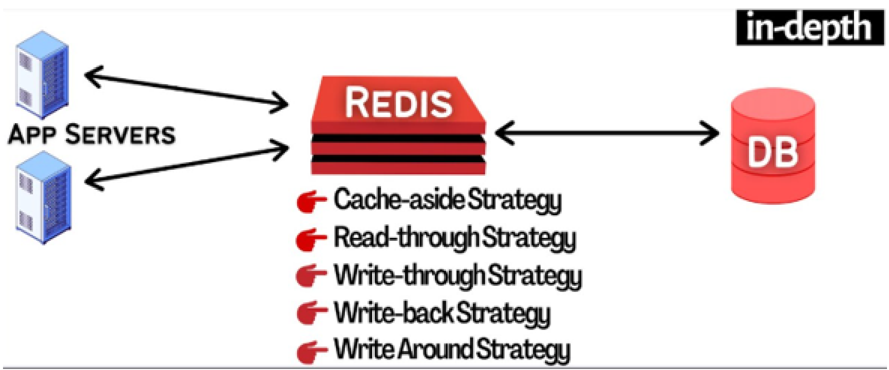
There are different types of caching present at different layer of the system like:
1. Client side Caching (Browser Caching)

2. CDN
3. Load Balancer
4. Server Side Application Caching (like Redis)



Distributed Caching consists of multiple cache servers forming a cache pool. App Server is connected with cache client and that client is connected to cache pool. It uses **consistent hashing** for cache client connections from cache pool.

Caching Strategies:



1. Cache Aside
   Application checks cache first (cache hit or cache miss).
   Data is fetched from DB if it's a cache miss, then store the data in cache and data is returned to the client. App Server handles fetching data from DB.
   Suitable for read-heavy applications but can lead to inconsistency for write because write add data in DB and when client wants to get the data it store that data in cache. And when client update that data, then it doesn't update the data in cache because the data is already present in cache for that.

2. Read Through
   Similar to Cache Aside but Cache handles fetching data from DB.
   No need for application to manage cache; simplicity in logic. Same cons as above.

3. Write Around
   Directly writes data into DB without updating cache.
   It there is an update in data and that data is present in cache, it invalidate the data in cache. So cache is marked dirty and it will re-fetch the data and store new data in cache.
   It resolves inconsistency problem b/w cache and DB. If DB is down, write operation will fail.
   Alone it’s not useful (use Read Through or Cache Aside).

4. Write Through
   First writes to cache, then writes data in DB in a synchronous manner. Ensures consistency between cache and DB.
   Alone it’s not useful, it’ll increase latency (that’s why it’s always used with Read Through or Cache Aside).
   2 Phase Commit is needed because if writing to cache is success but writing to DB fails then it have to rollback. If DB down, write will fail.

5. Write Back (or Behind) Cache
   First writes to cache, then push the message to Queue. Queue then push/writes data in DB in a asynchronous manner.
   Gives better performance when used with Read Through Cache.
   Even when DB fails, write operation will still works.
   Increases performance and fault tolerance, but can risk losing data if cache TTL (Time to Live) expires.
   If data is removed from cache and DB write still not happens due to DB down, then there is a issue (it is handled by keeping the TAT (Turnaround Time is used to refer to the
   time taken to fulfill a cache miss) of cache little higher like 2 days).