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 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.

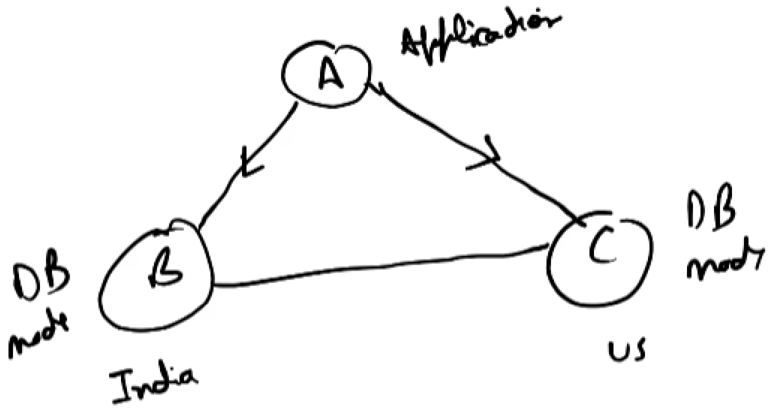
TCP and UDP are used to transport/transfer the data.

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.

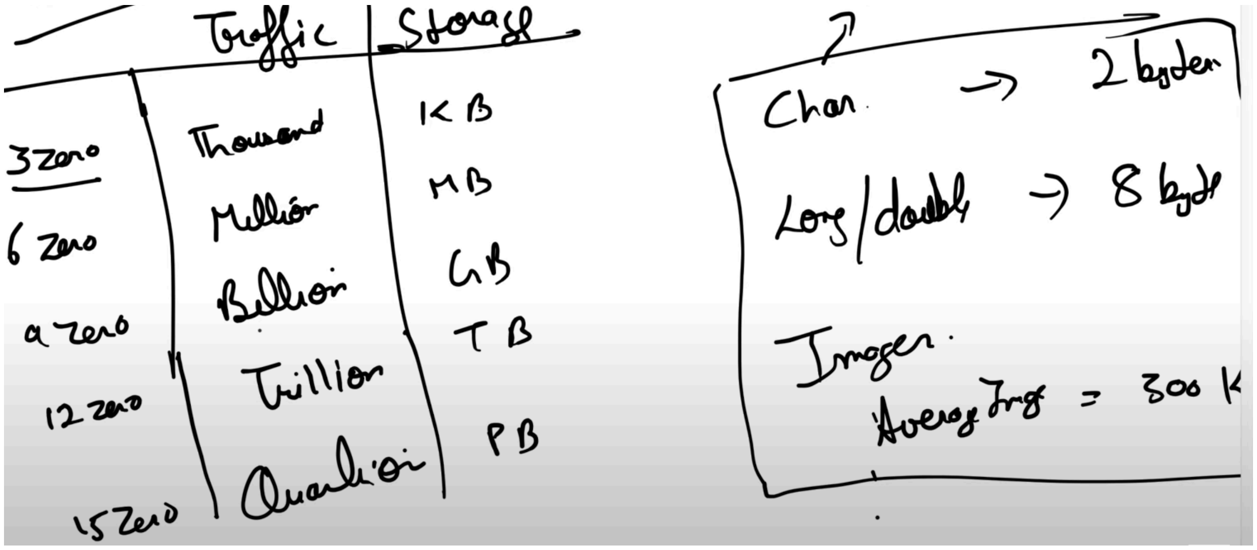
→ Consistent Hashing

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.

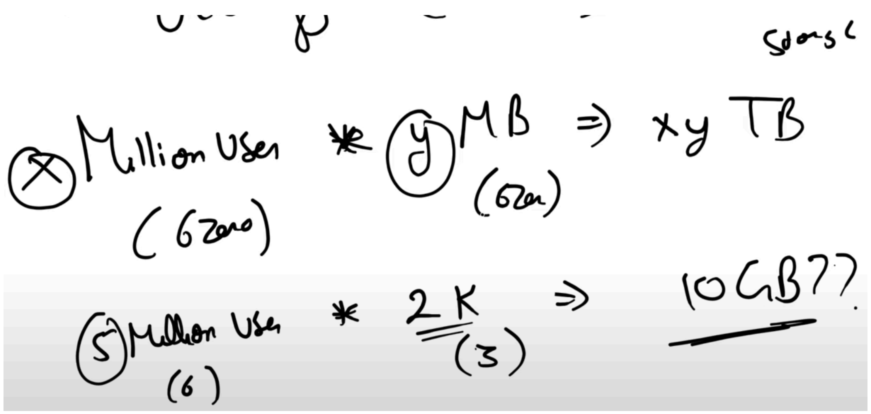
Consistent hashing is a distributed hashing technique that places servers and keys on a circular hash ring using a hash function. It helps distribute load evenly and minimizes the number of keys that need rebalancing when servers are added or removed, making it ideal for scalable systems. By mapping 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:


Storage
x Million Users (6 zero) * y MB (6 zero) ⇒ xy TB
5 Million Users (6) * 2 K (3) ⇒ 10 GB??