**HLD Topics.**

**Network Protocols:**

**- Definition:** Network protocols define the rules and regulations for communication between systems over a network.

**- OSI Model:** Refers to the Open Systems Interconnection Model, a layered architecture for network communication (not covered in detail).

**- Key Layers:**

**- Application Layer:** Handles user-facing communication (e.g., web browsing, email).

**- Transport Layer:** Manages reliable data transfer between applications (e.g., TCP/UDP).

**Application Layer Protocols:**

**- Client-Server Model:**

**- Client:** Initiates requests (e.g., web browser).

**- Server:** Responds to requests (e.g., web server).

**- Examples:** HTTP, FTP, SMTP, IMAP, Web Sockets.

**- Peer-to-Peer Model:**

**- Peers:** all machines can send and receive requests from each other

**- Example:** File sharing, instant messaging, WebRTC.

**HTTP (Hypertext Transfer Protocol):**

**- Key Points:**

- Most widely used protocol for web communication.

- Connection-oriented.

- Used for accessing web pages, web applications.

**- Example:** WhatsApp (uses web sockets over HTTP for messaging).

**FTP (File Transfer Protocol):**

**- Key Points:**

- Transferring files between computers.

- Two connections: Control connection and Data connection.

**SMTP (Simple Mail Transfer Protocol):**

**- Key Points:**

- Sending emails.

- Works with IMAP for receiving and reading emails.

**IMAP (Internet Message Access Protocol):**

**- Key Points:**

- Receiving and reading emails.

- Allows access to emails from multiple devices.

**Transport Layer Protocols:**

**- TCP (Transmission Control Protocol):**

**- Key Points:**

- Connection-oriented (establishes a virtual connection).

- Divides data into packets and sequences them for reliable transmission.

- Provides error checking and retransmission.

**- Use Case:** WhatsApp, applications requiring reliable data transfer.

**- UDP (User Datagram Protocol):**

**- Key Points:**

- Connectionless.

- Sends data in packets without establishing a connection.

- No guarantee of delivery or order.

- Faster and more efficient than TCP.

**- Use Case:** Live streaming, video calling, applications where some data loss is acceptable.

**Key Takeaways:**

- Understanding network protocols is crucial for designing distributed systems.

- Client-server and peer-to-peer models are fundamental architectures for communication.

- TCP provides reliable data transfer, while UDP prioritizes speed.

- The choice of protocol depends on the specific requirements of the application.

**CAP Theorem:**

**- CAP:** Stands for Consistency, Availability, Partition Tolerance.

**- Desired Properties of Distributed Systems:** These three properties are desirable in a distributed system.

**- CAP Trade-off:** You cannot have all three properties simultaneously. You must choose two out of the three.

**- Example:**

- Consider a distributed database with nodes in India and the US.

- A user's data is replicated across both locations.

- A distributed system should ideally be consistent (same data everywhere), available (responding to requests), and tolerant to partitions (network disruptions).

**Understanding Each Property:**

**- Consistency:** Ensures that all nodes have the same, up-to-date data at any given time.

**- Availability:** Guarantees that every request is successfully processed by at least one node.

**- Partition Tolerance:** Allows the system to continue functioning even if communication between nodes is disrupted.

**Why CAP Properties Cannot Co-Exist:**

**- Case 1: CA (Consistency and Availability) -** Not possible with Partition Tolerance

**- Scenario:** A partition occurs, separating nodes A and B.

**- Conflict:** Node A writes updated data, but node B cannot access it due to the partition.

**- Result:** Inconsistency between nodes.

**- Case 2: CP (Consistency and Partition Tolerance) -** Not possible with Availability

**- Scenario:** A partition occurs, separating nodes A and B.

**- Strategy:** To maintain consistency, only one node (A) is allowed to process writes.

**- Result:** Node B becomes unavailable for writes during the partition.

**- Case 3: AP (Availability and Partition Tolerance) -** Not possible with Consistency

**- Scenario:** A partition occurs, separating nodes A and B.

**- Strategy:** To maintain availability, both nodes can process writes.

**- Result:** Potential for inconsistent data between nodes.

**CAP Trade-off in Real-world Systems:**

- Importance of Partition Tolerance: In today's distributed systems, network disruptions are common.

- Choosing between CP and AP:

- CP: Choose this option for systems where consistency is critical, even if it means some temporary downtime.

- AP: Choose this option for systems where availability is paramount, even if it means some data inconsistency.

**Key Takeaways:**

- Understanding CAP is crucial for effective distributed system design.

- Early consideration of CAP constraints can prevent costly changes later.

- The choice between CP and AP depends on the specific needs of your system.

- Partition tolerance is a key factor in modern distributed systems.

**Disadvantages of Monolithic Architecture:**

- Tight Coupling

- Changing one line can impact other components

- Need to test/deploy entire application for one change

- Difficult to scale

- If one component needs scaling, entire application needs to be scaled

- Expensive deployments and rollbacks

- Entire application needs redeployment for a small change

- Large codebase

- Everything in a single application

- Codebase grows large over time

- Difficult to make changes, understand impact

**Why Microservices?**

- To overcome monolithic disadvantages

- Split large application into small services

**Advantages of Microservices:**

- Better separation of concerns

- Loose coupling between services

- Independent deployment of services

- Easy to scale out specific services

- Faster release cycles

**Disadvantages of Microservices:**

- Proper service boundaries/decomposition is challenging

- Inter-service communication is complex

- Monitoring calls across services

- Handling failures

- Distributed transaction management is difficult

- Across multiple databases

**Microservices Design Phases:**

- Decomposition patterns

- Database patterns

- Communication patterns

- Integration patterns

- Deployment patterns

- Cross-cutting concerns like monitoring, logging

**Decomposition Patterns:**

**- By business capability:**

- Split based on business functions like order mgmt, inventory, etc

**- By subdomain - Domain Driven Design (DDD):**

- Split large domains into multiple services

- E.g. splitting payment domain

**Microservices Design patterns:**

- Strangler Pattern: Used for refactoring a monolithic service into microservices.

- Saga Pattern: Solves the problem of distributed transactions across multiple databases.

- CQRS (Command Query Responsibility Segregation): Separates command (write) and query (read) operations for improved performance and scalability.

**Strangler Pattern:**

**- Purpose:** Gradually refactoring a monolithic application into microservices.

**- How it Works:**

- A "controller" is introduced to handle requests.

- Initially, the controller forwards all traffic to the monolithic application.

- Gradually, specific functionalities are extracted into microservices, and the controller routes traffic to them.

- As more functionalities are migrated, the controller forwards less traffic to the monolithic application, eventually strangling it.

**- Advantages:**

- Minimizes disruption to existing services.

- Allows for a gradual transition to microservices.

**- Example:**

- Imagine a monolithic e-commerce website being refactored into microservices.

- The controller initially directs all traffic to the monolithic website.

- Gradually, functionalities like order placement, inventory management, and payment processing are moved to individual microservices.

- The controller gradually routes more traffic to these microservices, eventually reducing the reliance on the monolithic application.

**Data Management in Microservices:**

- Two Approaches:

- Database for Each Individual Service: Each microservice has its own dedicated database, promoting autonomy and isolation.

- Shared Database: All microservices share a single database, simplifying data access but potentially leading to complexities.

- Why Database per Service is Preferred:

- Scalability: Allows for independent scaling of individual services without impacting others.

- Isolation: Changes in one service's database don't affect others.

- Technology Flexibility: Services can choose different databases based on their specific needs.

- Advantages of Shared Database

- Join Query

- Transactional Property (ACID)

- Drawbacks of Shared Database:

- Performance Bottlenecks: Increased contention and performance issues as more services access the same database.

- Complexity: Managing dependencies and ensuring consistency across multiple services becomes difficult.

- Limited Scalability: Scaling the entire database is necessary, even if only one service needs more resources.

**Saga Pattern:**

**- Purpose:** Managing distributed transactions across multiple databases, ensuring data consistency even if some operations fail.

**- How it Works:**

- A sequence of local transactions is executed within each participating microservice.

- Each transaction updates the database and publishes an event.

- Subsequent transactions listen to these events and continue the process.

- In case of failure, compensation events are published to undo completed operations and maintain consistency.

**- Types of Sagas:**

**- Choreography:** Each service manages its own transactions and listens to events from other services.

**- Orchestration:** A centralized orchestrator manages the transaction flow and handles compensation logic.

**- Example:**

- An order processing saga involving services for order creation, inventory management, and payment processing.

- If the payment service fails, compensation events are triggered to cancel the order and update inventory.

**- Advantages:**

- Guarantees data consistency in distributed systems.

- Provides a mechanism for handling failures and rollbacks.

- Allows for flexibility in service interactions.

**- Disadvantages:**

- Increased complexity compared to local transactions.

- Requires careful design and implementation to ensure correctness.

**- Interview Question Example:**

- Explain how you would handle a transaction involving transferring money between two users in a microservice architecture.

**CQRS Pattern:**

**- Purpose:** Separating read (query) operations from write (command) operations for better performance and scalability.

**- How it Works:**

- The system maintains separate models for read and write operations.

- Write operations are performed through commands, updating the write model.

- Read operations access the read model, which can be optimized for fast retrieval.

**- Advantages:**

- Performance Improvement: Optimized read models can handle queries more efficiently.

- Scalability: Read and write models can be scaled independently based on their specific needs.

- Flexibility: Allows for different data structures and query languages for read and write operations.

**- Example:**

- A blog application where write operations are performed on a relational database, while read operations access a denormalized view optimized for fast search.

**- Challenges:**

- Maintaining consistency between the read and write models.

- Ensuring the read model is up-to-date with changes in the write model.

**Steps for Scaling:**

**1. Single Server:**

- Basic setup with a single server for the application, database, and client.

- Suitable for the initial stage with zero users.

**2. Application and Database Separation:**

- Introduces a separate layer for the application server, handling business logic.

- Database server handles data storage and retrieval.

- Enables independent scaling of both application and database.

**3. Load Balancing and Multiple Application Servers:**

- Introduces a load balancer to distribute incoming requests across multiple application servers.

- Load balancer provides security and privacy.

- Ensures efficient handling of increased traffic by distributing workload.

**4. Database Replication:**

- Implements a master-slave configuration for the database.

- Master database handles write operations, while slave databases handle read operations.

- Improves performance and provides redundancy in case of database failure.

**5. Caching:**

- Utilizes a caching layer to store frequently accessed data in memory.

- Application server checks the cache first before accessing the database.

- Reduces database load and improves response time.

- Uses time-to-live (TTL) to manage cached data expiry.

**6. Content Delivery Network (CDN):**

- Uses a distributed network of servers to cache static content closer to users.

- Reduces latency and improves website performance for users worldwide.

- Handles requests for static content like images, videos, and JavaScript files.

- In case of cache miss, CDN first ask neighbour CDN for the data then from origin DB.

**7. Multiple Data Centers:**

- Distributes the application and database across multiple data centers.

- Reduces load on individual data centers and improves reliability.

- Enables geographically distributed user access with minimal latency.

- Load balancer distributes requests to different data centers based on user location.

**8. Messaging Queues:**

- Uses messaging queues to handle asynchronous tasks like sending notifications or emails.

- Decouples tasks from the main application flow.

- Improves performance and reliability by handling high-volume tasks efficiently.

- Utilizes messaging platforms like RabbitMQ or Kafka.

**9. Database Scaling:**

**- Vertical Scaling:** Increase the capacity of existing database servers (CPU, RAM). This has limitations and eventually reaches a ceiling.

**- Horizontal Scaling / Data Sharding :** Split the database into multiple servers or shards, distributing data across them. This offers better scalability.

- Splits data across multiple databases or tables based on a specific key.

- Can be implemented through vertical sharding (splitting columns) or horizontal sharding (splitting rows).

**Consistent Hashing:**

<https://blog.algomaster.io/p/consistent-hashing-explained>  
  
**Back of Envelope Estimation:**

|  |  |  |
| --- | --- | --- |
| **Zeros** | **Traffic** | **Storage** |
| 3 Zeros | Thousands | KB |
| 6 Zeros | Million | MB |
| 9 Zeros | Billion | GB |
| 12 Zeros | Trillion | TB |
| 15 Zeros | Quadrillion | PB |

<https://systemdesign.one/back-of-the-envelope/>

**Idempotency:**

<https://medium.com/@debasisdasnospdii/idempotency-in-scalable-distributed-architectures-example-e23a12c70048>

<https://blog.algomaster.io/p/idempotency-in-distributed-systems>

**Active-passive and active-active Database architecture:**

<https://medium.com/@gaurabhinav54/active-active-vs-active-passive-data-centers-architecture-2cef51d47785>

<https://www.mongodb.com/developer/products/mongodb/active-active-application-architectures/>

**Kafka vs. RabbitMQ:**

<https://medium.com/@PubNub/kafka-vs-rabbitmq-choosing-the-right-messaging-broker-78ca02530e63>  
  
**How Microservices Communicate Using a Service Mesh**

**1. Introduction**

Microservices architecture breaks applications into small, independent services that communicate over a network. As the number of services grows, managing communication (e.g., service discovery, load balancing, security, and observability) becomes complex.

A **Service Mesh** is a dedicated infrastructure layer that handles service-to-service communication, providing reliability, security, and observability without requiring changes to application code.

**2. Key Components of a Service Mesh**

A service mesh consists of:

**a. Data Plane**

* **Sidecar Proxies (e.g., Envoy, Linkerd-proxy)**
  + Deployed alongside each microservice as a separate container/pod.
  + Intercepts all incoming/outgoing traffic.
  + Handles load balancing, retries, timeouts, TLS encryption, and observability.

**b. Control Plane**

* Manages and configures the proxies (e.g., Istio, Linkerd, Consul).
* Provides:
  + Service discovery
  + Traffic management (A/B testing, canary deployments)
  + Security (mTLS, RBAC)
  + Monitoring & tracing

**3. How Microservices Communicate via Service Mesh**

**Step 1: Service Discovery**

* When **Service A** wants to call **Service B**, the request first goes to **Service A’s sidecar proxy**.
* The proxy queries the **service mesh control plane** to find the available instances of **Service B**.

**Step 2: Secure Communication (mTLS)**

* The service mesh automatically encrypts traffic using **mutual TLS (mTLS)**.
* Both services authenticate each other before exchanging data.

**Step 3: Traffic Routing & Load Balancing**

* The sidecar proxy applies routing rules (e.g., canary releases, blue-green deployments).
* Distributes traffic across healthy instances of **Service B**.

**Step 4: Resilience Features**

* The proxy handles:
  + **Retries** (if a request fails)
  + **Circuit breaking** (stops sending requests to a failing service)
  + **Timeouts** (prevents long waits)

**Step 5: Observability (Logs, Metrics, Traces)**

* The service mesh collects telemetry data (latency, errors, request rates).
* Integrates with tools like **Prometheus (metrics), Jaeger (tracing), and Grafana (visualization)**.

**4. Benefits of Using a Service Mesh**

✅ **Decouples networking logic from business logic**  
✅ **Improves security with automatic mTLS**  
✅ **Simplifies traffic management & observability**  
✅ **Enables zero-trust networking (fine-grained access control)**

**5. Popular Service Mesh Implementations**

* **Istio** (Most feature-rich, uses Envoy)
* **Linkerd** (Lightweight, simpler to deploy)
* **Consul** (By HashiCorp, integrates with their ecosystem)
* **AWS App Mesh** (For AWS-native microservices)

**6. Conclusion**

A service mesh abstracts away the complexity of microservice communication, providing security, reliability, and observability out of the box. By offloading networking concerns to sidecar proxies, developers can focus on business logic while the mesh ensures smooth, secure, and efficient communication.  
  
**Optimistic and Pessimistic Locking(Concurrency Control):**  
<https://medium.com/@abhirup.acharya009/managing-concurrent-access-optimistic-locking-vs-pessimistic-locking-0f6a64294db7>

I struggled with system design—until I learned these concepts (it might help you too):  
  
1. Idempotent API:  
↳ [**https://lnkd.in/erMkqwq4**](https://lnkd.in/erMkqwq4)  
  
2. Saga Design Pattern:  
↳ [**https://lnkd.in/eFXC4-aJ**](https://lnkd.in/eFXC4-aJ)  
  
3. Protocol Buffers vs JSON:  
↳ [**https://lnkd.in/egcFxe2t**](https://lnkd.in/egcFxe2t)  
  
4. Consistent Hashing:  
↳ [**https://lnkd.in/eUP9DbCg**](https://lnkd.in/eUP9DbCg)  
  
5. Capacity Planning:  
↳ [**https://lnkd.in/ev358pj3**](https://lnkd.in/ev358pj3)  
  
6. Service Discovery:  
↳ [**https://lnkd.in/eCYYwQfU**](https://lnkd.in/eCYYwQfU)  
  
7. Monolith vs Microservices:  
↳ [**https://lnkd.in/e3EBtg\_v**](https://lnkd.in/e3EBtg_v)  
  
8. Microservices Lessons From Netflix:  
↳[**https://lnkd.in/eZSM3CRB**](https://lnkd.in/eZSM3CRB)  
  
9. What Happens When You Type a URL Into Your Browser?:  
↳ [**https://lnkd.in/eusuDn5z**](https://lnkd.in/eusuDn5z)  
  
10. Caching Patterns:  
↳ [**https://lnkd.in/gJ8kWMxZ**](https://lnkd.in/gJ8kWMxZ)  
  
11. Modular Monolith Architecture:  
↳ [**https://lnkd.in/edUqkCKR**](https://lnkd.in/edUqkCKR)  
  
12. How Databases Keep Passwords Securely:  
↳ [**https://lnkd.in/edd8-N5k**](https://lnkd.in/edd8-N5k)  
  
13. Redis Use Cases:  
↳ [**https://lnkd.in/ekJMjMG3**](https://lnkd.in/ekJMjMG3)  
  
14. How to Scale an App to 10 Million Users on AWS:  
↳ [**https://lnkd.in/eU736g9Q**](https://lnkd.in/eU736g9Q)