Basic Foundations of System Design: From Principles to a Use Case: Live Streaming Application

-Siddhant Patil

1. Introduction to System Design:

a) Overview:

This document offers a comprehensive introduction to system design, tailored for software developers and engineers who want to begin their System design journey with no prior knowledge.

The document is divided into two parts:

- Part 1: Basic engineering design patterns used in building large-scale distributed systems.
- Part 2: Applying these principles to design a live streaming video app.

b) Objectives and Learning Goals:

- Identify Basic Design Patterns: You will be able to recognize and understand basic engineering design patterns.
- Design Large-Scale Systems: Learn how to apply these patterns to design scalable and reliable distributed systems.
- Understand Real-World Applications: See how these principles are used in real-world applications like Google Maps and social media platforms.

c) Key Terms

- **System Design:** The process of defining the architecture, modules, interfaces, and data for a system to satisfy specified requirements from the Product Requirements Document (PRD).

  Example: Think of system design as the blueprint for a house. You wouldn't want to start building without one, unless you enjoy watching things crumble!

- **Large-Scale**: Systems designed to handle a vast amount of data, users, or transactions efficiently.

  Example: Google Maps handles data from the entire world's geography, used by millions daily, with constant updates and high-performance expectations.

- **Distributed Systems**: Systems where components located on different networked computers communicate and coordinate their actions by passing messages.

  Example: The servers for a service like Netflix are spread across the globe. If you're in India, you don't want to wait for a server in the US to respond.

By understanding these concepts and how they apply to real-world examples, you'll be equipped to tackle the challenges of designing robust, scalable systems.

2. Understanding Large-Scale Distributed Systems:

Large-scale distributed systems are complex systems where components located on different networked computers work together to achieve a common goal. These systems are designed to handle large volumes of data and numerous simultaneous user requests efficiently.

Here are some key characteristics:

- Scalability: Ability to handle increasing amounts of work by adding more resources (e.g. more servers).
- Fault Tolerance: System continues to operate even if some components fail.
- Consistency: Ensures that all users see the same data at the same time.
- Latency: Minimizing the delay in both data processing and communication between components. Low latency is crucial for a good user experience.

Imagine a distributed system as a well-coordinated "orchestra". Each musician (server) plays their part perfectly, even if one of them misses a beat or two.

Real world example: Google Maps

- Data Handling: Stores detailed geographic data for the entire world.
- User Load: Millions of users simultaneously accessing and updating maps.
- Performance Expectations: Users expect quick and accurate responses. No one likes a detour due to slow map updates.
- Reliability: Uses globally distributed servers to ensure the service is always available, even if one server fails.
- Distributed Servers: Servers located worldwide to ensure fast response times and high availability.

Engineering Considerations (The 5-finger fist!):

a) Performance:

Users expect fast load times and quick responses. Slow performance can lead to user frustration and loss of engagement. Efficient systems make optimal use of resources, reducing costs and improving user experience.

Example: Google Maps returning directions in milliseconds, even during peak usage times.

b) Reliability:

Systems need to be available 24/7, especially for critical services. Systems must handle errors gracefully without affecting the overall functionality.

Example: Social media platforms remaining online and functional even during major events like elections.

c) Load Balancing:

Distributing the workload across multiple servers to ensure no single server is overwhelmed.

Example: During online shopping sales, load balancing helps handle the surge in traffic without crashing the site.

d) Caching:
Storing frequently accessed data in a temporary storage area to improve access speed.
Example: When you revisit a webpage, it loads faster because the content is cached.

e) Redundancy:
Having multiple copies of data or systems so that if one fails, others can take over.
Example: Cloud storage services like Google Drive ensure your data is backed up and available even if one server goes down.

By understanding these engineering fundamental concepts and seeing how they apply in real-world examples, we are better prepared to design and work with large-scale distributed systems.

3. Introduction to Design Patterns:

Design patterns are tried-and-tested solutions to common problems in software design. They represent best practices used by experienced developers to solve recurring problems efficiently. Think of them as reusable templates or blueprints that you can customize to fit your specific needs.

Most Commonly used Patterns:

a) Publisher-Subscriber Model- The Publisher-Subscriber (Pub-Sub) model is a popular design pattern used to manage communication between different components of a system without them needing to know about each other.
Use Case: Think of a news website (Publisher) that sends notifications for breaking news. Users (Subscribers) receive these notifications on their devices through a notification service (Broker). If Tom Cruise posts a new movie trailer, millions of fans get notified instantly without Brad having to message each one personally.

b) Singleton Pattern- Ensures a class has only one instance and provides a global point of access to it. Prevents the creation of multiple instances that could lead to resource conflicts and inconsistency.
Example: Think of a captain of a ship. There's only one captain who makes all the decisions.
However, it is difficult for testing and gives issues in multithreaded environments.

c) Factory Pattern- Creates objects without specifying the exact class of object that will be created. Promotes code flexibility and reusability by decoupling object creation from the code that uses the objects.
Example: Imagine a cake factory where you order cakes by specifying the type, and the factory decides how to make it

d) Observer Pattern- Allows an object to notify other objects about changes in its state. Enhances communication between objects in a loosely coupled manner, promoting a flexible and scalable architecture. However, it can lead to performance issues if not managed correctly, especially with many observers. *Example: Think of a weather station that updates all connected displays whenever there's a change in weather conditions*.

e) Decorator Pattern- Adds additional functionality to objects dynamically. Provides greater flexibility in extending functionalities compared to inheritance.
Example: Consider a coffee shop where you can add various toppings to your coffee, like whipped cream or caramel.

f) Load Balancer Pattern- Distributes incoming network traffic across multiple servers to ensure no single server becomes overwhelmed. Enhances system reliability and performance by preventing server overloads and ensuring high availability.
Example: Imagine a busy restaurant where a host directs incoming guests to different tables to ensure no single waiter is overwhelmed.

g) Cache Pattern- Stores frequently accessed data in a temporary storage area to reduce the time it takes to access this data. Reduces latency and improves response times by minimizing the need to fetch data from the original source repeatedly.
Example: Think of a frequently used spice rack in the kitchen, which saves you from searching through the entire pantry each time you cook.

A Most used Architectural Style:

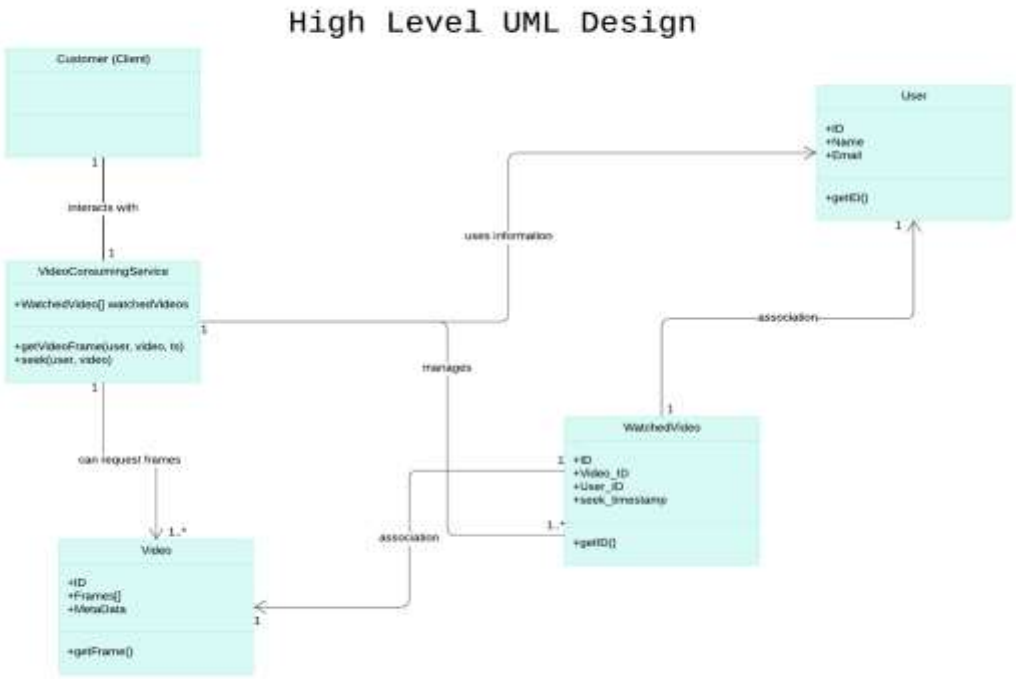
h) Microservices Architecture- An architectural style that structures an application as a collection of small, loosely coupled services, each running in its own process and communicating through lightweight mechanisms, often HTTP/REST with JSON. This pattern is widely adopted due to its ability to handle large-scale, complex systems with numerous, independently deployable services.

By understanding and applying these design patterns, you can build more robust, scalable, and maintainable systems. Each pattern addresses specific challenges and provides a structured approach to solving common problems in software design. So, the next time you face a design challenge, remember- there's probably a pattern for that!

4. Use Case: Designing a Robust Live Streaming App.

Overview:

This use case details the process of designing a robust live streaming application. It covers identifying user requirements, translating these requirements into technical solutions, implementing engineering considerations, and processing and delivering video content efficiently. The goal is to ensure a seamless and engaging user experience while maintaining system reliability and scalability.



(Note: The above UML is created by me (based on my perception))

a) Identifying User Requirements: To design a live streaming app that meets user expectations, we first identify the key requirements-

- Live Streaming: Users want to watch live video content without buffering or delays.
- Comments: Users should be able to post comments in real-time, fostering interaction with the content and other viewers.
- Reactions: Features like likes or emojis allow users to engage with the stream instantly.
  Example: A live sports event streaming on platforms. Users expect to watch the match in real-time, comment on the plays, and react with likes or emojis.

Once we have identified user requirements, we translate them into technical solutions-

- Streaming: Implement a robust video streaming protocol that ensures minimal latency and high-quality playback. Protocols like WebRTC (P2P) and MPEG DASH are suitable for different scenarios.
- Comments: Develop a real-time messaging system using HTTP to handle comments without causing delays. This system must be scalable to handle thousands of simultaneous comments.

- Reactions: Use lightweight data packets to send and display reactions instantly, ensuring minimal impact on performance.

b) Engineering Considerations: To ensure the app is reliable and can scale with user demand, we incorporate several engineering considerations:
   - Fault Tolerance and Redundancy- Distribute the load across multiple servers in different locations to ensure availability. Duplicate data across servers so that if one fails, another can take over seamlessly. Implement continuous monitoring of server health to reroute traffic in case of server failure.
   - Scalability and Extensibility- Add more servers to handle increased user load and data (like Horizontal Scaling). Use a modular approach to add new features without affecting existing functionality.
   - Testing and Validation of Design- Simulate high traffic to ensure the system can handle peak loads (Load Testing). Verify that all components work together smoothly. Lastly, collect feedback from real users to identify and fix any issues before going live.

c) Implementation Details: Choosing the right protocols and data storage solutions is crucial for performance and scalability:
   Network Protocols:
   - HTTP: Reliable for sending comments and reactions, as it's stateless and handles retries well.
   - WebRTC: Ideal for real-time video communication due to its low latency.
   - MPEG DASH: Best for adaptive streaming, adjusting video quality based on network conditions.

   Data Storage Solutions
   - SQL: Suitable for structured data like user profiles and comments.
   - NoSQL: Good for flexible data models like reaction data.
   - NewSQL: An option that combines the benefits of SQL and NoSQL.
   - File Systems: Ideal for storing large video files.

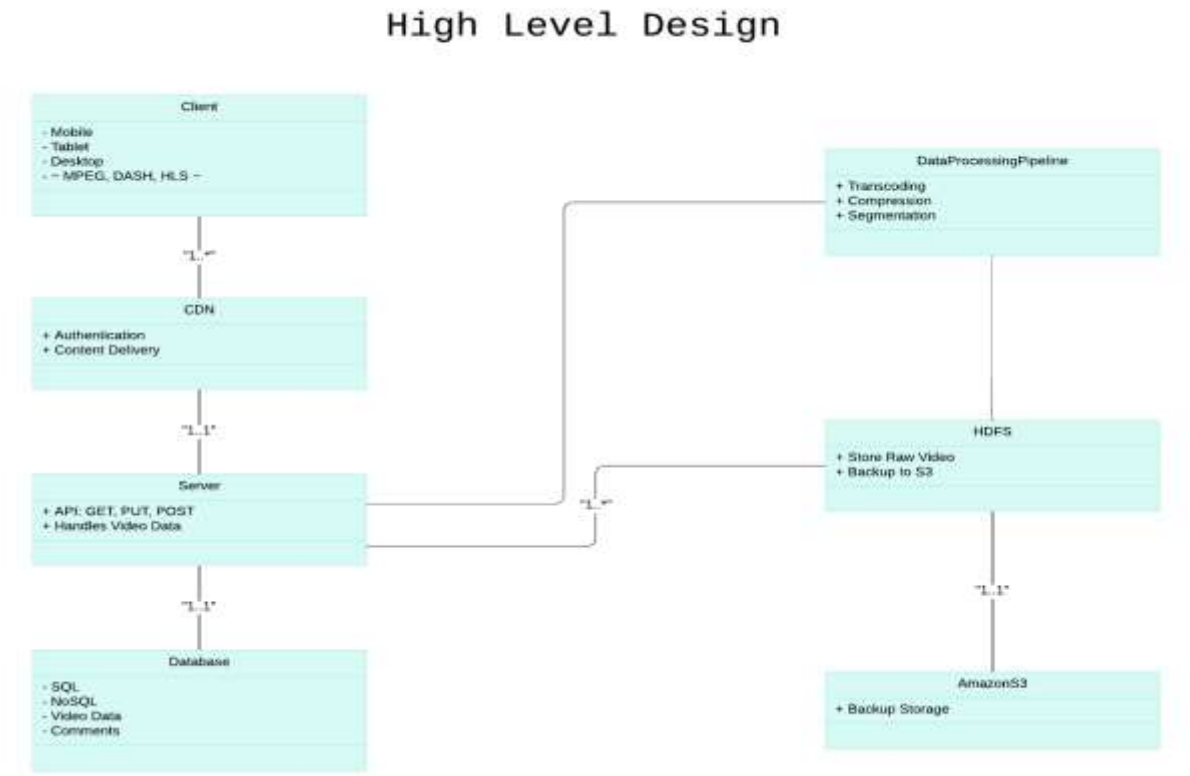
   Additional Consideration: Content Delivery Networks (CDNs)
   Store copies of content closer to users to reduce latency via Caching. Balance the load across multiple servers to avoid overloading any single server.
   Eg: Akamai, Cloudflare, or Amazon CloudFront.

d) Video Transformation and MapReduce Pattern: Efficiently processing raw video data is essential for delivering content in various formats and resolutions.
   - Segmentation: Break the video into manageable chunks for easier processing and transform these chunks into different formats (e.g., H.264) and resolutions (e.g., 1080p, 720p).
   - Using MapReduce for Efficient Transformation: Split the video into smaller segments for parallel processing and then combine processed segments into the desired formats and resolutions.

Examples of Resolution and Format Conversion-
High Definition (1080p): For users with high-speed internet and large screens.
Standard Definition (480p): For users with slower internet connections.
Mobile-Friendly Formats: Optimized for different mobile devices.

5. Conclusion

Designing a robust live streaming app involves understanding user requirements, translating these into technical solutions, implementing engineering best practices, and efficiently processing and delivering video content. By focusing on these aspects, you can create a seamless and engaging experience for users, ensuring reliability and scalability for future growth.



(Note: The above UML is created by me (based on my perception))