System Design

People Tech Group Week 3(L1 Batch 2)

Name: Shrestha Sai Laxmi Yadavilli
Email: shrestha.yadavilli22@gmail.com

Table of Contents

1. Introduction

The study of system design involves understanding how to create scalable, efficient, and reliable software architectures that can handle complex requirements. This week's task is focused on grasping the basic concepts and components required for building such systems, which are important skills for software engineers working on large-scale projects.

1.1 Topics Covered

- **What is System Design?** Understanding the process of creating an architecture to handle specific requirements.
- **Steps to Approaching a System Design Problem:** Techniques for breaking down complex problems into manageable components.
- **Basic Concepts in System Design:** Core principles like reliability, availability, and scalability.
- **Functional and Non-Functional Requirements:** Differentiating between essential features and quality attributes.
- **Approaches to Solving System Design:** Top-down and bottom-up strategies for tackling design problems.
- **Components of System Design:** Key elements like load balancers, caching, sharding, and more.
- **Scalability in System Design:** Techniques to ensure a system can handle increased demand.
- **Databases in System Design:** Choosing between SQL and NoSQL databases.
- **High-Level and Low-Level Design:** Differentiating between the high level design and the low level design.
- **Advantages and Disadvantages of System Design:** The pros and cons of implementing a structured design process.
- **System Design Problems:** Case studies like designing a URL shortener and a YouTube-like platform.

1.2 Objective of the Week's Task

The goal for this week was to understand the foundational concepts of system design, learn how to approach design problems, and become familiar with the essential components that make up scalable and efficient software systems.

2. Understanding the Basic System Design Concepts

2.1 What is System Design?

System design is the process of defining the architecture, components, and data flow of a software system to meet certain requirements. It involves creating a plan for how different parts of the system will interact and work together to handle large-scale problems effectively. It's a vital skill for software engineers, especially for creating systems that can handle growth and increased demand.

2.2 Steps to approaching a system design problem

- **Understand the requirements**: Before designing anything, it's important to know what the system needs to do. This means understanding the problem you're solving, how fast it should work, how much it should handle, and any security concerns.
- **Identify the major parts**: Break down the system into its key pieces and figure out how they connect. Understand how each part helps the system function as a whole.
- **Choose the right technology**: Based on what the system needs, pick the right tools and technologies. This includes deciding on hardware, software, databases, programming languages, and tools.
- **Define how parts communicate**: Set up how different parts of the system will talk to each other, using things like APIs (how software components interact), protocols, and data formats.
- **Design the data structure**: Plan how data will be organized, stored, and shared within the system. This includes designing the database structure and how data flows between different parts.
- **Think about growth and speed**: Make sure your design can handle more users or data in the future. Use techniques like load balancing (distributing work evenly), caching (storing data for quick access), and database tuning.
- **Test the design**: Check if your design works by testing it with real-world examples. Adjust as needed if you find problems.

- **Deploy and keep it running**: Once everything is ready, launch the system. Keep it up-to-date, fix issues, and add new features over time.

2.3 Basic Concepts in System Design

Reliability in System Design
Reliability means a system meets user needs consistently. A reliable system should function properly and deliver all intended features without failure. Fault tolerance is key; it allows the system to keep working even if some components fail. A fault does not always lead to a failure, which occurs when the system can no longer provide its services.

Availability in System Design
Availability refers to how often a system is operational and ready to serve users. High availability is crucial for critical systems like hospitals or banks, where delays can be costly. Avoiding single points of failure and quickly detecting issues are essential for maintaining availability.

Scalability in System Design
Scalability is the system's ability to handle increased loads. When designing, plan for a load much greater than expected (e.g., design for 10X and test for 100X). This is vital for scenarios like e-commerce flash sales, where sudden spikes in traffic occur. Understanding potential load is crucial for ensuring scalability.

Functional Requirements
Functional requirements specify what the system must do. They outline the essential features that users expect and must be incorporated into the system. These requirements are often expressed as inputs, operations, and expected outputs. For example, features to be designed and edge cases to consider.

Non-Functional Requirements
Non-functional requirements define the quality attributes the system should meet, such as performance, security, and scalability. These are not about what the system does but how it performs. For instance, the system should process requests with minimal latency or ensure high availability.

Functional Vs Non-Functional Requirements
- Functional requirements defines what the system should do while Non-Functional requirements defines how the system should perform.
- Functional requirements are specified by users, whereas non-functional requirements are specified by technical teams.
- Functional requirements are mandatory for the system, while non-functional requirements enhance quality but are not mandatory.
- Functional requirements are captured in use cases, in contrast to non-functional requirements that are captured as quality attributes.
- Functional requirements are usually easier to define, while non-functional requirements tend to be more complex and harder to articulate.

Approaches to solving System Design
- **Bottom-up Approach:** The bottom-up approach starts with the lowest level components and gradually integrates them to form the complete system. This strategy allows for reusing existing solutions, making it efficient. However, it can struggle to align with the overall problem structure, and achieving high-quality solutions may be challenging.
- **Top-Down Approach:** The top-down approach involves dividing the entire system into subsystems, refining each layer until the lowest level is reached. This method ensures a strong focus on requirements, leading to a responsive design. Conversely, it may overlook opportunities for component reuse and could result in a complex architecture that is difficult to manage.

2.4 Components of System Design

- **Load Balancer**: A load balancer evenly distributes incoming requests among multiple servers, preventing any single server from becoming overwhelmed. This enhances efficiency and ensures the system can handle high traffic.
- **Caching:** Caching leverages the locality of reference principle, meaning recently accessed data is likely to be requested again. It acts like short-term memory, providing fast access to frequently used

data, albeit with limited space. Types of Caches are: Application Server Cache, Distributed Cache, Global Cache, Content Delivery Network (CDN). Different strategies dictate which data is removed when the cache reaches its limit: FIFO (First In, First Out), LIFO (Last In, First Out), LRU (Least Recently Used), Most Recently Used, Least Frequently Used, Random Replacement

- **Sharding:** Sharding, or data partitioning, involves splitting a large database into smaller, manageable pieces for efficiency. It can be achieved through horizontal partitioning (storing different rows across tables) or vertical partitioning (storing feature-specific tables separately). Each method has its challenges, such as ensuring uniform data distribution and maintaining referential integrity. Common strategies include hash-based, list, and round-robin partitioning. While sharding improves performance, it complicates data retrieval, necessitating approaches like denormalization to address inefficiencies.
- **Key-value Stores**: These NoSQL databases store data as pairs of unique keys and values, enabling quick data retrieval. They are particularly effective for applications that require rapid access to frequently used data.
- **Blob Storage & Databases**: Blob storage is designed for unstructured data like images and videos, offering scalability for large files. In contrast, databases manage structured data, allowing for complex queries and data relationships.
- **Rate Limiters**: Rate limiters manage the number of requests a system can process within a specified timeframe. They protect systems from overload and ensure stable performance during peak usage times.
- **Monitoring System**: Monitoring systems gather and analyze metrics related to system performance. They provide insights into operational health and help identify issues before they affect users.
- **Distributed System Messaging Queue**: These queues facilitate asynchronous communication between components in a distributed system. They enhance scalability and reliability by decoupling message producers from consumers.
- **Distributed Search**: Distributed search systems leverage multiple servers to index and search large datasets efficiently. This parallel processing improves response times and scalability for search operations.

2.5 Scalability in System Design

2.5.1 What is Scalability?

Scalability is the ability of a system to grow and handle more work as demand increases. A scalable system can keep performing well even when the number of users or the amount of data rises. This quality is crucial for businesses because it ensures that their services remain fast and reliable as they expand.

2.5.2 Importance of Scalability in System Design

- It allows systems to support more users and larger amounts of data without slowing down.
- Scalable systems can spread out workloads, leading to quicker response times.
- They stay operational even during traffic spikes or failures, making them dependable.
- Scalable systems can adjust resources based on demand, helping to save money.
- They enable businesses to add new features easily, adapting quickly to market changes.

2.5.3 Methods for achieving Scalability

Vertical Scaling:
Vertical Scaling means upgrading a single machine by adding more powerful hardware, like a better CPU or more memory. It enhances a single server's capabilities. For instance, if your server has limited power, you can boost its performance by upgrading components. While this approach is straightforward and easy to manage, it has limits based on hardware capabilities and can become costly if high-end upgrades are needed. Additionally, it creates a single point of failure; if the server fails, everything relying on it stops working.

Horizontal Scaling:
Horizontal Scaling involves adding more machines to share the workload. Both methods help improve system performance and capacity. This strategy can handle more users and requests by distributing tasks across multiple machines. It improves reliability because if one server fails, others can keep running. However, managing multiple servers can be more complex and expensive due to increased infrastructure and maintenance needs.

2.5.4 Components that Help Increase Scalability

- **Load Balancer**: Distributes incoming traffic across multiple servers to prevent overload.
- **Caching**: Stores frequently accessed data temporarily to reduce retrieval times and backend load.
- **Database Replication**: Copies data to multiple locations to improve access speed and reliability.
- **Database Sharding**: Divides a database into smaller parts to distribute data and workload effectively.
- **Microservices Architecture**: Breaks applications into smaller, independent services that can be scaled individually.
- **Data Partitioning**: Splits data into manageable segments, improving performance and scalability.
- **Content Delivery Networks (CDNs)**: Caches content closer to users, reducing loading times.
- **Queueing Systems**: Manages requests by allowing them to be processed asynchronously, helping to balance traffic.

2.6 Databases in System Design

When designing a system, selecting the appropriate database management system (DBMS) is critical. The choice between SQL (Structured Query Language) and NoSQL (Not Only SQL) databases can significantly impact your system's performance and scalability.

SQL databases, known for their structured and tabular data models, require a predefined schema. This makes them ideal for applications needing complex queries, strict data integrity, and robust relationships. They typically adhere to ACID (Atomicity, Consistency, Isolation, Durability) principles, ensuring data integrity. However, they generally scale vertically, which can become expensive and challenging for large-scale applications. MySQL, PostgreSQL, Oracle and Microsoft SQL Server are some examples of SQL databases.

NoSQL databases provide a flexible schema and can handle unstructured or semi-structured data. They prioritize performance and scalability, often using the BASE (Basically Available, Soft State, Eventually Consistent) model, which allows for greater availability at the cost of strict consistency. NoSQL databases excel in horizontal scaling, distributing data across multiple servers to accommodate large volumes of traffic and data. MongoDB, Cassandra, Redis, Neo4j are some examples of NoSQL databases.

2.7 High Level Design

High-Level Design (HLD) provides an overarching view of a system's architecture and components. It outlines the system's structure, including database design and the relationships between modules, focusing on what needs to be done rather than how to do it.

The HLD document includes data flows, flowcharts, and data structures, which aid developers in understanding system interactions and architecture. It outlines both functional and non-functional requirements without delving into detailed technical aspects. HLD serves to model operational requirements and how modules interact, acting as a reference for development. It includes diagrams illustrating the architecture, interfaces, and typical user workflows.

2.8 Low Level Design

Low-Level Design (LLD) focuses on the detailed specifications of system components and their interactions. It translates the high-level design into a precise blueprint, detailing algorithms, data structures, and interfaces. LLD serves as a roadmap for developers during coding, ensuring the system's functionality is implemented correctly and efficiently.

Class diagrams play a crucial role in LLD by illustrating entities within components, making it easier for developers to transform these diagrams into code. While High Level Diagram provides a broad overview of the system's architecture, Low Level Diagram delves into specifics about component structures and their operations, often utilizing UML diagrams and principles like OOP and SOLID for clarity and organization.

2.9 Advantages and Disadvantages of System Design

Advantages:

- System design helps clarify what the system needs to do and any limitations, making it easier to understand the problem you're solving.

- By choosing the right tools and organizing data properly, system design helps the system run faster and more smoothly.
- Good system design ensures that the system can handle more users, data, or features as it expands in the future.
- With clear guidelines on how different parts connect and share data, it's easier to update and fix the system over time.
- System design makes it easier to share ideas with others, like developers and users, so everyone knows how the system works and what to expect.

Disadvantages:

- Designing a system, especially a large or complicated one, can be a long process.
- System design can be expensive due to the need for research, prototypes, and thorough testing.
- You need technical knowledge about hardware, software, databases, and data structures to design a system properly.
- Once a system is designed and built, making changes can be tough, especially if it's already up and running.

3. System Design Problems

3.1 Problem 1: URL Shortner

The need for efficient and concise URL management has become a big matter in this technical age. URL shortening services, such as bit.ly, and TinyURL, play a massive role in transforming lengthy web addresses into shorter, shareable links. As the demand for such services grows, it has become vital to understand the System Design of URL Shortner and mastering the art of designing a scalable and reliable URL-shortening system, to gain a crucial skill for software engineers.

Approach:

1. **Requirements and Components:**
   - **Functional Requirements:** Shorten URLs, redirect to original URLs.
   - **Non-Functional Requirements**: Scalability, low latency, high availability, fault tolerance.
   - **Frontend**: User interface for submitting URLs and displaying short links.
   - **Backend**: Handles URL shortening, redirection, and analytics.
   - **Database**: Stores mappings of short URLs to long URLs and any associated metadata.

2. **Algorithm:**
   - Base62 encoder allows to use the combination of characters and numbers which contains A-Z, a-z, 0–9 total(26 + 26 + 10 = 62).
   - So for 7 characters short URL, we can serve 62^7 ~= 3500 billion URLs which is quite enough in comparison to base10 (base10 only contains numbers 0-9 so you will get only 10M combinations).
   - We can generate a random number for the given long URL and convert it to base62 and use the hash as a short URL id.

3. **Schema:**
   - URLs: Contains fields like id, long_url, short_url, creation_date, and click_count.
   - Here we use a NoSQL database (like MongoDB) for flexibility and horizontal scaling or a SQL database (like PostgreSQL) for structured data.

4. **Redirect Service**
   - Set up an endpoint (e.g., /r/{short_url}) that looks up the short URL in the database and redirects to the corresponding long URL.
   - The challenge here is performance of the url shortner at scale. For this we can perform caching for frequently accessed URLs.

Challenges Faced:

- One of the challenges for this system design could be handling traffic, which can be solved by implementing load balancing and sharding in the database to districute traffic over multiple servers.
- Handling multiple users making changes at the same time can be solved with database transactions.

- Keeping the service available is important. This can be achieved with backup databases that can take over if needed.
- To prevent misuse, we can limit how often users can create short links and check if URLs are safe by implementing URL validation, and monitoring for suspicious activities.
- Lastly, creating a user-friendly interface can improve user experience, which can be accomplished by focusing on a simple and clean design with fast processing times.

3.2 Problem 2: Design Youtube

Youtube is one of the most popular and extensible biggest streaming video streaming services and the architecture contains various components that enhance user experience. When it comes down to Youtube services, it is been that commonly used in the daily world with tremendous users so the system design corresponding is likely a bit complex.
Video rendering system designs are very expensive when it comes to the cost of designing systems. It is a primary reason why you find paid courses do have their own player. Designing YouTube involves considering various components and functionalities that make up the platform.

Approach:

1. **Requirements and Components:**
   - **Functional Requirements:** User-friendly interface for video interactions, support for various video formats, seamless sharing options, commenting system with threading and moderation, real-time updates for comments, advanced search capabilities based on titles, descriptions, and metadata, filtering options by date, duration, and popularity, customized news feeds based on user activity and preferences, intelligent recommendation algorithms
   - **Non-Functional Requirements**: redundant systems for high availability, regular backups and disaster recovery strategies, load balancing for high traffic handling, monitoring for quick issue resolution, horizontal scaling techniques, caching mechanisms to reduce backend load

2. **High Level Design:**
   - The design should be easy to use and visually attractive, making it simple for users to navigate across computers, phones, and smart TVs.
   - Users can create accounts to upload videos, comment, and subscribe. Secure login methods like email/password and social media sign-ins should be available.
   - Users should upload videos in various formats. A strong storage system is needed to handle a lot of video data, including converting videos into different quality options for smooth streaming.
   - After uploading, videos need to be optimized. This includes converting videos into various resolutions and formats, generating thumbnails, and adding features like closed captions.
   - A Content Delivery Network (CDN) will help distribute videos quickly around the world, using adaptive streaming to adjust video quality based on the user's internet speed.
   - A system that suggests videos to users based on their interests and viewing history should be implemented using machine learning.
   - Users can comment, like, share videos, manage subscriptions, and create playlists.
   - A system should be in place to identify and remove inappropriate content, using both automated tools and human review.
   - Creators should receive data on how their videos perform, including viewership and engagement stats, to help them improve their content.
   - Options for creators to earn money should be offered, such as ads and memberships, with a system in place to share revenue fairly.
   - The system needs to support many users and video uploads simultaneously. Techniques like caching and load balancing will help maintain performance.
   - User data and video information should be stored securely, and tools should be available to analyze user behavior and system performance.
   - Developers should have access to APIs to create new features or connect with other services like social media and advertising networks.

3. **Low-level Design:**
   - Load Balancing helps distribute incoming traffic across multiple servers, ensuring no single server gets overloaded. This leads to better resource use.

- YouTube handles a vast amount of data, including user content and video files. It needs the right storage solutions, like distributed file systems or object storage, to manage this data effectively.
- A CDN caches video content in various locations around the world, reducing buffering and improving loading times for users.
- YouTube needs to support different video formats and resolutions, which requires converting uploaded videos into various formats suitable for streaming on different devices.
- Efficient database management is vital for storing and retrieving data on users, videos, and comments. Techniques like database sharding, indexing, and caching help manage high traffic.
- A recommendation engine uses algorithms to analyze user behavior and suggest relevant videos, boosting user engagement.
- A robust system tracks performance and user activity, helping identify issues and optimize the system.
- Secure login methods are essential to protect user data and ensure only authorized users can upload or modify content.
- This ensures a safe environment by detecting and removing inappropriate content through machine learning, human reviews, and community reporting.
- Users expect notifications for events like new uploads or comments. A reliable notification system ensures these updates are delivered promptly.

Challenges Faced:

- It's challenging to maintain performance as user traffic grows, so horizontal scaling, load balancing, and using cloud infrastructure are essential.
- Ensuring low latency and high availability for video streaming requires using CDNs and adaptive streaming protocols.
- Managing inappropriate or harmful content effectively involves combining machine learning algorithms with human review processes.
- Delivering instant notifications and updates is addressed by implementing scalable messaging systems and WebSockets.
- Protecting user information and complying with regulations is managed through robust encryption and secure authentication mechanisms.

4. References

1. [1] "System Design for Beginners Course," YouTube, [Online]. Available: https://www.youtube.com/watch?v=m8Icp_Cid5o.

2. [2] "Basics of System Design Playlist," YouTube, [Online]. Available: https://www.youtube.com/playlist?list=PLt4nG7RVVk1g_LutiJ8_LvE914rIE5z4u.

3. [3] "System Design Tutorial," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/system-design-tutorial/ .

4. [4] "Top 10 System Design Interview Questions and Answers," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/top-10-system-design-interview-questions-and-answers/.