**System Design**

**1. Introduction**

I’ve been diving into system design, focusing on how to build a video streaming service. This involves creating diagrams and coding out the structure of the system. Here’s a rundown of what I’ve learned and how I understand it.

**2. Use Case Diagram**

The use case diagram helps me understand how users interact with the system. For this video service:

Actors: The main actor is the User.

Use Cases: Users can watch a video and retrieve specific frames from it.

**3. Class Diagram**

The class diagram breaks down the main components of the system into classes, each with its attributes (data) and methods (behaviors).

3.1 Video Class

Attributes:

`id`: A unique identifier for the video.

`frames`: An ordered list of frames that make up the video.

`metadata`: This could be any extra info about the video, like who uploaded it or its length, stored in a JSON format.

Behaviors:

`getFrame(timestamp)`: This method fetches a specific frame based on the timestamp provided.

3.2 User Class

Attributes:

`id`: A unique identifier for the user.

`name`: The user's name.

`email`: The user's email address.

Behaviors:

`getId()`: This method returns the user's ID.

3.3 WatchedVideo Class

Attributes:

`id`: Unique identifier for the action of watching a video.

`videoId`: The ID of the video being watched.

`userId`: The ID of the user watching the video.

`seekTimestamp`: The timestamp up to which the video has been watched.

Behaviors:

`getSeekTimestamp()`: Returns the timestamp indicating how much of the video has been watched.

3.4 VideoConsumingService Class

Attributes: This class doesn’t have attributes.

Behaviors:

`getSeekTime(userId, videoId)`: Gets the seek time for a specific user and video.

`getFrame(videoId, timestamp)`: Retrieves a frame from a video at a specific timestamp.

4. Sequence Diagram

The sequence diagram shows how different components interact over time. Here’s the flow:

1. User requests to watch a video.

2. VideoConsumingService retrieves the current seek time for the video.

3. User asks for a specific frame.

4. VideoConsumingService sends a request to the VideoServiceto get the frame.

5. VideoService fetches and sends back the frame.

6. VideoConsumingService delivers the frame to the User.

This process has taught me how to design a video streaming system from scratch. By creating and understanding these diagrams and implementing the code, I’ve learned how to structure a system effectively and ensure all components work together smoothly.

**Load Balancing**

Load balancing is the process of distributing traffic across multiple servers to ensure no single server becomes overwhelmed.

**Example:** Google employs load balancing extensively across its global infrastructure to distribute search queries and traffic evenly across its massive server farms.

**Caching**

Store frequently accessed data in-memory (like RAM) to reduce the load on the server or database. Implementing caching can dramatically improve response times.

**Example:** Reddit uses caching to store frequently accessed content like hot posts and comments so that they can be served quickly without querying the database each time.

Overall,I got a good idea about system design concepts

Coming to these two problems just they are under design so I tried to solve it

Intuition and Approach:

The problem revolves around implementing a simple hash set, and the challenge lies in efficiently handling operations like adding, removing, and checking for the existence of keys. The hash set must be designed to achieve near-constant time complexity for these operations. Here's a step-by-step breakdown of the approach used:

Basic Idea of a Hash Set:

A hash set is essentially a data structure that uses a hash function to compute an index (or hash code) for each key. This index determines where the key should be stored in an array (or other data structure).

The goal is to have the key-value pairs spread out evenly across the array to minimize collisions, which are situations where two different keys hash to the same index.

Using an Array with Linked Lists for Collision Resolution:

The array size is set to \( 2^{15} \), which provides a good trade-off between memory usage and the likelihood of collisions.

Each element of the array is a bucket that can hold multiple keys. If multiple keys hash to the same index, they are stored in a list (or linked list) at that index. This method is called separate chaining.

Hash Function Design:

A simple multiplicative hash function is used: `eval\_hash(key) = ((key \* 1031237) & (1<<20) - 1) >> 5`.

The constant `1031237` is a large prime number that helps in achieving a good distribution of hash values.

Bitwise operations (`&` and `>>`) are used to map the result to a valid index within the array's bounds. This makes the hash function fast and effective.

Operations:

Add: The key is hashed to find the appropriate bucket. If the key is not already in the bucket, it is added.

Remove: The key is hashed, and if found in the corresponding bucket, it is removed.

Contains: The key is hashed, and the bucket is checked to see if the key exists.

Complexity Analysis:

Time Complexity:

Average Case: Add, Remove, Contains: O(1) - The hash function computes the index in constant time. Assuming a good hash function and a reasonable load factor (number of keys relative to the number of buckets), each bucket should have a small number of keys, resulting in near-constant time operations.

Worst Case: Add, Remove, Contains: O(n) - In the worst case, all keys hash to the same bucket, leading to a list traversal. However, this is rare with a good hash function and sufficient bucket size.

A screenshot of a computer program

Description automatically generated

2. I solved the problem of implementing a hash set using a simple and effective multiplicative hashing approach. Here's how I approached the solution and why it works efficiently for the given constraints:

Problem Understanding

The goal was to implement a hash set that supports three main operations: `add`, `remove`, and `contains`. The challenge was to manage up to 10,000 operations efficiently, using a hash function to map keys to indices in an array.

Choosing the Hash Function

I decided to use a multiplicative hashing function, which is straightforward yet effective for distributing keys uniformly across the array. The multiplicative hashing function works by multiplying the key by a large, odd number (or a prime number), then reducing the result to a manageable range using bitwise operations.

Hash Function Details

Multiplicative Hashing Formula: I used the formula: `hash = (key \* a) % (2^w) >> (w - m)`.

Key: The integer key we are trying to hash.

a = 1031237: This is a large odd number I chose randomly. Using a large odd number or a prime helps to distribute the keys more uniformly.

w = 20: I chose 20 as the size of the machine word, ensuring it is larger than `m` and suitable for the key size.

m = 15: This is the number of bits for the output hash, which determines the size of our hash set array. I chose `m = 15` because `2^15 = 32,768`, which comfortably handles the maximum number of operations with minimal collisions.

Hash Function Implementation: Using Python’s bitwise operations:

`((key \* 1031237) & ((1 << 20) - 1)) >> 5`

`key \* 1031237`: Multiplies the key by the large odd number.

`& ((1 << 20) - 1)`: This performs a bitwise AND with `2^20 - 1` to get the last 20 bits, simulating a modulo `2^20`.

`>> 5`: Shifts the result right by 5 bits, effectively narrowing down the range to `2^15` possibilities.

Implementing the Hash Set

Initialization: I initialized the hash set with an array of size `2^15`, each containing an empty list. This size is determined by `m = 15`, which allows for a manageable number of buckets, balancing memory use and collision probability.

Add Operation: To add a key:

Compute its hash using the `eval\_hash` function.

Check the corresponding bucket (a list at the index given by the hash).

If the key isn’t already present, add it to the list.

Remove Operation: To remove a key:

Compute the hash of the key.

If the key exists in the corresponding bucket, remove it from the list.

Contains Operation: To check if a key exists:

Compute the hash of the key.

Check if the key is present in the corresponding bucket and return `True` or `False`.

Complexity Analysis

Space Complexity: `O(2^15)` because the size of the array is fixed at `2^15`. This is efficient and suitable for up to 10,000 operations.

Time Complexity: In the average case, each operation (`add`, `remove`, `contains`) runs in `O(1)` time, assuming minimal collisions. However, in the worst-case scenario (with many collisions), the time complexity could degrade to `O(n)` for the number of keys in a single bucket.

By using a well-designed hash function and a reasonable array size, I was able to implement a hash set that efficiently handles up to 10,000 operations. The multiplicative hashing method provides a good balance between time and space complexity, making it suitable for the constraints provided. The implementation also shows how simple bitwise operations can effectively manage memory and handle potential collisions, ensuring fast and reliable performance.

A screenshot of a computer program

Description automatically generated