Script for the lecture: HLD Popular Interview Subtopics

Agenda

1. Design a Unique ID generator
2. Design a Rate Limiter

Design a Unique ID generator

In this problem, we are required to generate IDs (numeric values - 8 bytes long) that satisfy the following criteria:

1. The ID should be unique.
2. The value of the ID must be incremental.

FAQ: What does incremental mean?

Let us suppose two IDs (numeric values), i1 and i2, are generated at time t1 and t2, respectively. So incremental means here that if t1 < t2, it should imply that i1 < i2.

Ques: What options do we have to construct these IDs?

1. Auto increment in SQL
2. UUID
3. Timestamp
4. Master Slave
5. Timestamp + Server ID
6. Multi-Master

Ques1: Why can't we directly use IDs as 1,2,3,... i.e., the auto-increment feature in SQL to generate the IDs?

Sequential IDs work well on local systems but not in distributed systems. Two different systems might assign the same ID to two different requests in distributed systems. So the uniqueness property will be violated here.

Ques2: Why can't we use UUID?

We can not use UUID here because

1. UUID is random
2. UUID is not numeric.
3. The property of being incremental is not followed, although the values are unique.

Ques3: Why can't we use timestamp values to generate the IDs?

The reason is, again, the failure to assign unique ID values in **distributed systems**. It may be possible that two requests land on two different systems at the same timestamp. So if the timestamp parameter is utilized, both will be given the same ID based on epoch values (the time elapsed between the current timestamp and a pre-decided given timestamp)

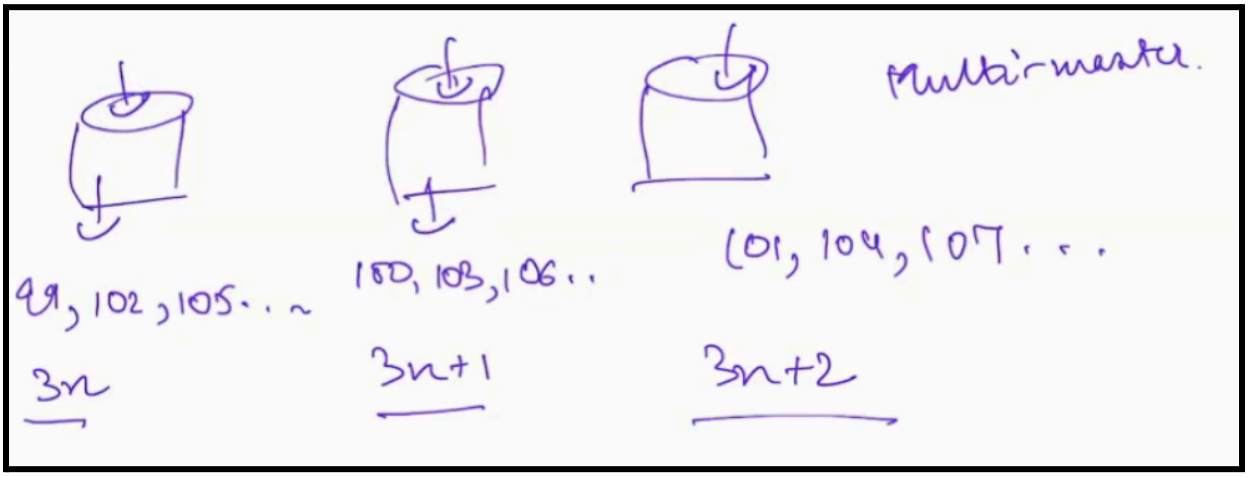
Next Approach: **Timestamp + ServerID**

The question here would be how many bits are to be assigned to the Timestamp and serverID. This situation would be tricky.

Next Approach: **Multi-Master**

Let us assume there are three different machines in a distributed computing environment. Let us number the machines as **M1**, **M2**, and **M3**. Now suppose the machine M1 is set to assign the IDs, which are a **multiple of 3n**, machine M2 is set to assign IDs that are a **multiple of 3n+1**, and machine M3 is set to assign IDs that are a **multiple of 3n+2**.

This situation can be represented diagrammatically as follows:

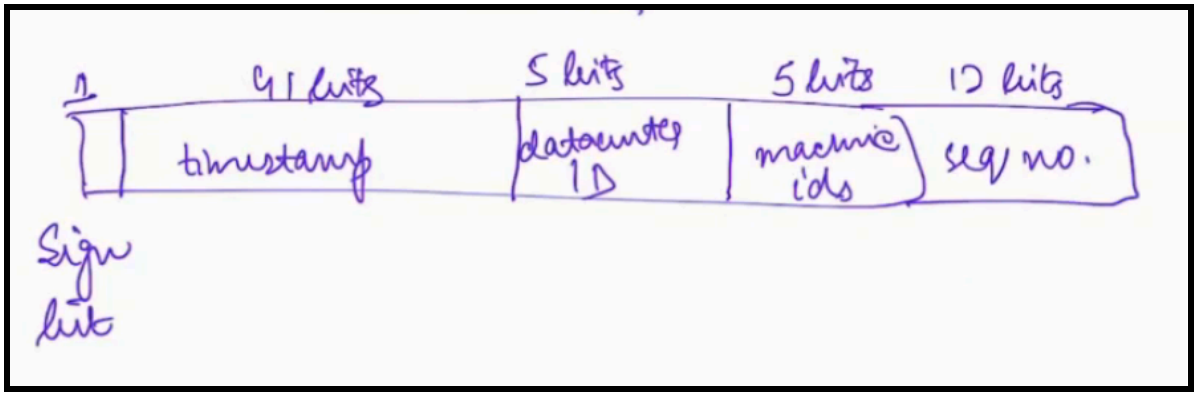


If we assign IDs using this technique, the uniqueness problem will be solved. But it might violate the incremental property of IDs in a distributed system.

For example, let us suppose that request keeps coming consecutively on M1. In this case, it would assign the IDs as 99,102,105,... . Now, after some time, the request comes on some other machine (e.g., M2), then the ID assigned would have a numeric value lower than what was assigned previously.

Twitter Snowflakes Algorithm:

In this algorithm, we have to design a 64-bit solution. The structure of the 64 bits looks as follows.



64-bit solution generated by Twitter Snowflakes Algorithm

Number of Bits (from left to right)	Purpose for which they are reserved
1 bit	Sign
41 bits	Timestamp
5 bits	Data center ID
5 bits	Machine ID
12 bits	Sequence Number

Let us talk about each of the bits one by one in detail:

1. Sign Bit:
The sign bit is never used. Its value is always zero. This bit is just taken as a backup that will be used at some time in the future.

2. Timestamp bits:
This time is the **epoch time**. Previously the benchmark time for calculating the time elapsed was from **1st Jan 1970**. But Twitter changed this benchmark time to **4th November 2010**.

3. Data center bits:
5 bits are reserved for this, which implies that there can be 32 (2^5) data centers.

4. Machine ID bits:

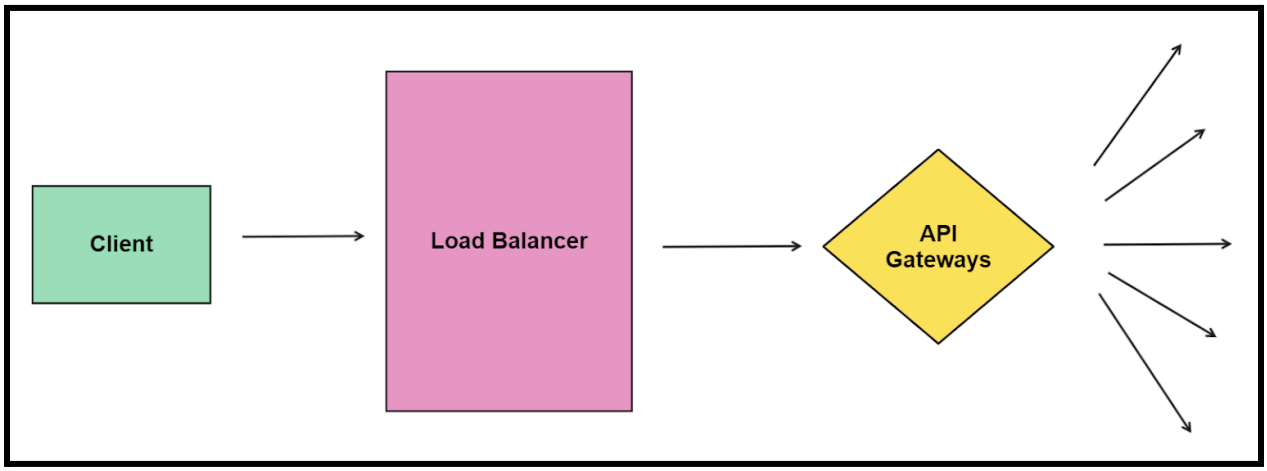
5 bits are reserved for this, which implies that there can be 32 (2^5) machines per data center.

5. Sequence no bits:
These bits are reserved for generating sequence numbers for IDs that are generated at the same timestamp. The sequence number is reset to zero every millisecond. Since we have reserved 12 bits for this, we can have 4096 (2^12) sequence numbers which are certainly more than the IDs that are generated every millisecond by a machine.

Further reading (optional): https://en.wikipedia.org/wiki/Network_Time_Protocol

Design a Rate Limiter

Rate limiter controls the rate of the traffic sent from the client to the server. It helps prevent potential DDoS attacks.

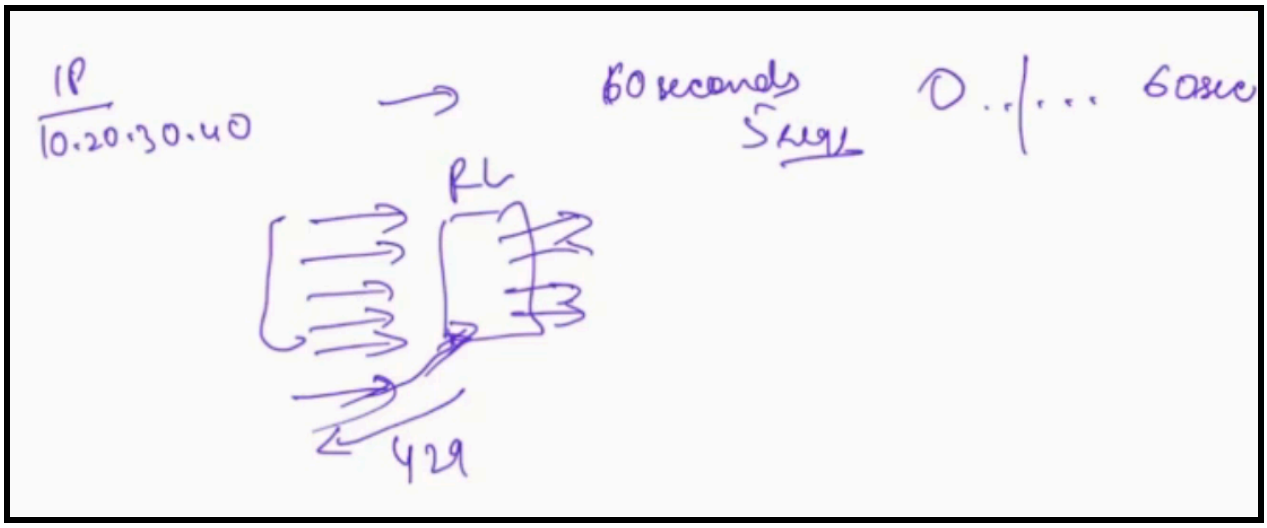


Throttling: It is the process of controlling the usage of the APIs by customers during a given period.

Types of Rate Limiter:

1. Client Side Rate Limiter
2. Server Side Rate Limiter

Let us consider an example. Suppose a user with IP address **10.20.30.40** sends requests to a server where a rate limiter is installed, and a rate of a maximum of 5 requests has been set every 60 seconds. In this case, if the user sends more than five requests in 60 seconds, its 6th request will be rejected, and an error code of 429 will be sent to him. The user can send this request in the next 60 seconds time frame. The situation is diagrammatically represented as follows:



Algorithms for rate limiting:

1. Token Bucket Algorithm
2. Leaking Bucket Algorithm
3. Fixed Window Counter
4. Sliding Window Counter

Data Structure that will be used: **HashMap**
The key would be IP addresses

Another possible solution is to maintain a **deque** for each IP address. We will enter the value of the timestamps at which the requests come in the deque. Suppose the rate limiting window is set for 5 seconds. So any request that is older than 5 seconds is not a concern for us.

The snapshot of deque at t = 15 looks as follows:

Deque:

10	11	12	12	13	14	15	

Now suppose a new request comes at t = 16. So *16 - 5 + 1 = 12*. Any request older than 12 should be removed from the deque.

Deque:

12	12	13	14	15	16		

Now the queue size will tell how many requests it has received in the last 5 seconds. If the deque size is less than the number of requests threshold, we will pass this request; otherwise, we will drop this request. Since the requests are coming in increasing order of the timestamp, the values inside the queue will be in sorted order.

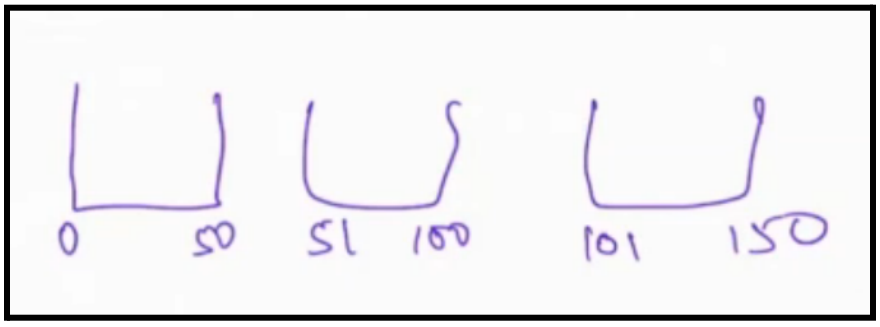
Challenges faced in this solution:

1. Let us suppose that the number of requests is very large (approx 10k). So processing time will increase significantly as we need to delete all these 10k requests one at a time from the deque. The data structure that we maintain in this is **map<string, queue>** where the string denotes the IP address of the client and for each client we have a separate queue which stores its requests in sorted order of time.
2. The solution is **memory intensive**.

Fixed Time Window Solution - Better Solution

Suppose the threshold is set to 100 requests per 50 seconds.

We will maintain buckets of 50 seconds like this.



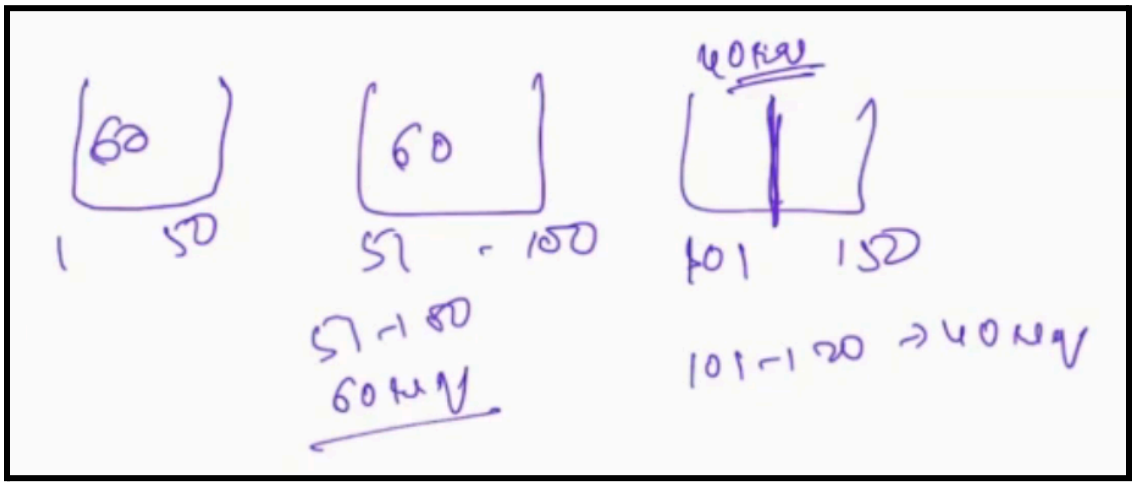
Now consider the following scenario. Suppose we receive 100 requests in the interval 40-50 seconds and again 100 requests in the interval 51-60 seconds. So this would lead to 200 requests in 40-60 seconds, which is **twice the threshold** value we have set. This is a common challenge that is faced when using this solution.

Sliding Window Solution - Best Solution

It is an approximation solution.

Let us suppose that we have again divided the timeline into buckets of size 50, i.e., the first interval is from t = 1 to 50, the second interval is from 51- 100, and the next is from 101 - 150, and so on.

Suppose we receive 60 requests in the interval t = 51 to 100 and 40 requests in the first half of the third interval ranging from t = 101 to 120.



Now we will use an approximation as follows.

Calculate what percentage of the interval (101,150) does the sub-range (101,120) constitute?

- Length of (101,120) is 20
- Length of (101,150) is 50
- So the percentage coverage would be (20/50)*100% = **40%**

So we have data for the 40% interval. Now to obtain data for the remaining 60% interval, we will use the last interval data and estimate it.

Thus 60% of the previous_count = 60 % of 60 = 36 requests.
The current_count is 40 requests.

Thus total count approximation for this interval is 36 + 40 = 76

Advantage: This reduces the amount of metadata that we need to keep. In the earlier deque solution, we need to keep track of all the requests separately. Now in this solution, we just need to keep the count of the prev_bucket and the cur_bucket. It can be stored using **map<string,pair<int,int> >**