**DESIGN A UNIQUE ID GENERATOR IN DISTRIBUTED SYSTEMS**

# **Requirement Gathering and estimation:**

1. What are the characteristics of unique IDs?

- IDs must be unique and sortable

1. For each new record, does ID increment by 1?

- The ID increments by time but not necessarily only increments by 1. IDs created in the evening are larger than those created in the morning on the same day.

1. Do ID only contain numerical values?

- Yes, that is correct.

1. Is there any restriction on the length of the requestId?

- IDs should fit into 64-bit

1. What is the scale of the system?  
   - The system should be able to generate 10,000 IDs / second

Requirements gathered:  
1. IDs must be unique.

2. IDs must be ordered by date and time

3. IDs are numerical in nature

4. IDs should fit in 64 bit.

5. Ability to generate 10,000 IDs/second

# **Proposing High level:**

Multiple options can be used to generate unique IDs in distributed system:

1. Multi-master replication
2. Random UUID
3. Ticket server
4. Twitter Snowflake approach

**Multi-master Replication:**Instead of increasing the next ID by 1, we increase it by k, where k is the number of database servers in use.

However, this strategy has some major drawbacks:  
1. Hard to scale with multiple Data centers

2. IDs do not go up with time across multiple servers.

3. It does not scale well when a server is added or removed

**Random UUID generator:**

UUID is a 128-bit number used to identify information in computer systems. UUID has a very low collision rate.  
After generating 1billion UUIDs every second for approx 100 years then the probability of creating a single duplicate would reach 50%.

UUIDs can be generated independently without coordination between servers.

Pros:

1. Generating IDs is simple. No coordination between servers is needed so there will not be any synchronization issues.
2. The system is easy to scale because each web server is responsible for generating IDs they consume. ID generators can easily scale with web servers.

Cons:

1. IDs are 128 bits long, but our requirement is 64 bits.
2. IDs do not go up with time
3. IDs could be non-numeric.

**Ticket Server:**The idea is to use a centralized auto\_increment feature in a single database server(Ticket Server)  
Pros:

1. Numeric IDs
2. Easy to implement, and works for small to medium-scale applications.

Cons:

1. Single point of failure

**Twitter Snowflake approach**

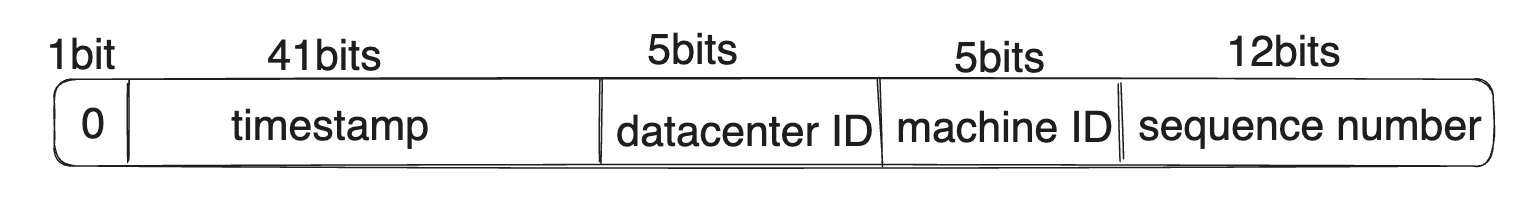
Divide and conquer is our friend. Instead of generating an ID directly, we divide an ID into different sections:  
1. Sign bit: 1 bit. This will always be 0. This is reserved for future uses. It can potentially be used to distinguish between signed and unsigned numbers.

2. Timestamp: 41bits. Milliseconds since the epoch or custom epoch. We use Twitter Snowflake epoch 1288834974657, equivalent to Nov 4, 2010, 01:42:54 UTC.

3. Datacenter ID: 5bits, which gives us 2^5 = 32 datacenters.

4. Machine ID: 5bits, which gives us 2^5 = 32 machines per datacenter.

5. Sequence number: 12 bits. For every ID generated on that machine/process, the sequence number is incremented by 1. The number is reset to 0 every millisecond.



# **Design Deep Dive**

Datacenter IDs and machine IDs are chosen at the startup time, generally fixed once the system is up running. Any changes in datacenter IDs and machine IDs require careful review since an accidental change in those values can lead to conflicts. Timestamp and sequence numbers are generated when the ID generator is running.

The maximum timestamp can be represented in 41 bits is:  
(2^41 -1) ~ (2.2 \* 10^12 milliseconds) ~ 69 years.

This means the ID generator will work for 69 years and having a custom epoch time close to today’s date delays the overflow time. After 69 years, we will need a new epoch time or adopt other techniques to migrate IDs.

Sequence number is 12 bits, which give us 2 ^ 12 = 4096 combinations. This field is 0 unless more than one ID is generated in a millisecond on the same server. In theory, a machine can support a maximum of 4096 new IDs per millisecond.

A few additional talking points:

• Clock synchronization: In our design, we assume ID generation servers have the same clock. This assumption might not be true when a server is running on multiple cores. The same challenge exists in multi-machine scenarios. however, it is important to understand the problem exists. Network Time Protocol is the most popular solution to this problem.

• Section length tuning: For example, fewer sequence numbers but more timestamp bits are effective for low concurrency and long-term applications.

• High availability: Since an ID generator is a mission-critical system, it must be highly available.