**Rate Limiter:**

## **What is a Rate Limiter?**

A rate limiter, at a high-level, limits the number of events an entity can perform in a particular time window.

In general, a rate limiter caps how many requests a sender can issue in a specific time window. It then blocks requests once the cap is reached.

## **Why do we need API rate limiting?**

Protect services for abusive behaviors, brute-force password attempts, security, reduce costs, revenue

## How to do Rate Limiting?

# **Requirements and Goals of the System**

### Functional Requirements:

* Limit the number of requests an entity can send to an API within a time window
* (Distributed Scenario) The APIs are accessible through a cluster, so the rate limit should be considered across different servers

### Non-Functional Requirements: (Availability, Latency, Scalability)

* Highly available
* Not introducing substantial latencies

# **Rate Limiting Algorithms**

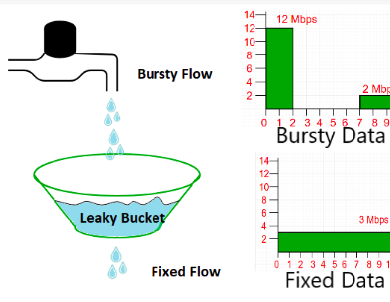
There are several different algorithms for implementing rate limiting. Let’s talk about each of them.

1**) Token based Bucket**

The token bucket algorithm is based on an analogy of a fixed capacity bucket into which tokens, normally representing a unit of bytes or a single packet of predetermined size, are added at a fixed rate. When a packet is to be checked for conformance to the defined limits, the bucket is inspected to see if it contains sufficient tokens at that time. If so, the appropriate number of tokens, e.g. equivalent to the length of the packet in bytes, are removed ("cashed in"), and the packet is passed, e.g., for transmission. The packet does not conform if there are insufficient tokens in the bucket, and the contents of the bucket are not changed. Non-conformant packets can be treated in various ways:

* They may be dropped.
* They may be enqueued for subsequent transmission when sufficient tokens have accumulated in the bucket.
* They may be transmitted, but marked as being non-conformant, possibly to be dropped subsequently if the network is overloaded.

A conforming flow can thus contain traffic with an average rate up to the rate at which tokens are added to the bucket, and have a burstiness determined by the depth of the bucket. This burstiness may be expressed in terms of either a jitter tolerance, i.e. how much sooner a packet might conform (e.g. arrive or be transmitted) than would be expected from the limit on the average rate, or a burst tolerance or maximum burst size, i.e. how much more than the average level of traffic might conform in some finite period.



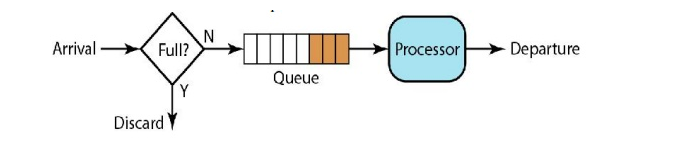
Advantage :

**Token Bucket for concurrency contro**l

## **2) Leaky Bucket**

It is the simplest algorithm to implement a rate limiter. It uses a bucket or [queue](https://nlogn.in/queue-data-structue-tutorial-and-implementation/) to hold the incoming requests. Whenever a new request arrives, it is appended to the rear of the queue, until the queue is not full.

The requests are processed at fixed time intervals in the first come first serve (FCFS) manner, i.e. old requests are the one to be executed first. If the [queue](https://nlogn.in/queue-data-structue-tutorial-and-implementation/) is full, the remaining are dropped or leaked with a proper message or notification to the client.



### Pros:

* Easy to implement
* Best used to control processing rate

### Cons**:**

* Leaky bucket algorithms cannot effectively take advantages of resources. The leaking rate is fixed, so it cannot process a bulky traffic that exceeds its threshold even though there are plenty of resources. (Token Bucket Algorithm can do that)

## **Sliding Window Log**

Sliding window log algorithm keeps a log of request timestamps for each user. When a request comes, we first pop all outdated timestamps before appending the new request time to the log. Then we decide whether this request should be processed depending on whether the log size has exceeded the limit. For example, suppose the rate limit is 2 requests per minute:



import java.util.LinkedList**;**

import java.util.Queue**;**

**public** **class** **SlidingWindowLog** **extends** **RateLimiter** **{**

**private** **final** **Queue<Long>** log **=** **new** **LinkedList<>();**

**protected** **SlidingWindowLog(int** maxRequestPerSec**)** **{**

**super(**maxRequestPerSec**);**

**}**

@Override

**boolean** **allow()** **{**

**long** curTime **=** **System.**currentTimeMillis**();**

**long** boundary **=** curTime **-** 1000**;**

**synchronized** **(**log**)** **{**

**while** **(!**log**.**isEmpty**()** **&&** log**.**element**()** **<=** boundary**)** **{**

log**.**poll**();**

**}**

log**.**add**(**curTime**);**

**return** log**.**size**()** **<=** maxRequestPerSec**;**

**}**

**}**

**}**

Redis sorted set we can use here in key value data structure : {user1:[‘00:11’,’00:12’]--soon }

each time we have only maximum maxRequestper sec so that ,if min will change than whole windlow will change based on timestamp They will discard present timestamp move on to next window

# **c) Database uses with highly available system**

While both are NoSQL databases, Redis is an in-memory data store that supports many different data types, used as a database, cache, and message broker. On the other hand, Cassandra is a distributed key-value store. Because Redis stores volume data in memory, its transactional response times are much faster than Cassandra that persists data to disk by performing traditional read-write transactions,

cassandra is the preferred option for use cases that require writing data to disk. It has a better Fault Tolerance than Redis .

Redis is preferred for use cases where massive amounts of rapidly changing data are processed in-memory before persisting to disk

so in this case we will use redis .

**Inconsistency in rate limiter handle through sticky session.**

**also we can prevent the race condition using redis – sorted sets**

The advantage of this approach is that **all Redis operations can be performed as an atomic action**, using the multi command. This means that if two processes both try to perform an action for the same user, there’s no way for them to not have the latest information, preventing the problem outlined above. It also allows us to use one limiter for both rates we wanted to track (i.e. no more than 10 messages per minute or 2 per 3 seconds).

However, there is one caveat to this approach – one we were comfortable with but may not fit others’ requirements. In our algorithm, you can see that whether or not an action is blocked is determined after all Redis operations are completed. This means that **blocked actions still count as actions**. So, if a user continually exceeds the rate limit, none of their actions will be allowed (after the first few), instead of allowing occasional actions through.