**Rate Limiter:**

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

A rate limiter, at a high-level, limits the number of events an entity (user, device, IP, etc.) 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

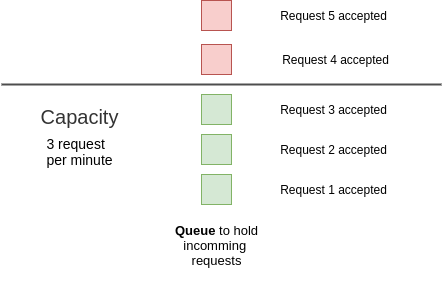
# **Rate Limiting Algorithms**

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

## **1) 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.



**The advantage of this algorithm:**

1. It smoothens burst of requests by processing them at a constant rate.
2. Easy to implement.
3. The size of the [queue](https://nlogn.in/queue-data-structue-tutorial-and-implementation/)(buffer) used will be constant, hence it is memory efficient.

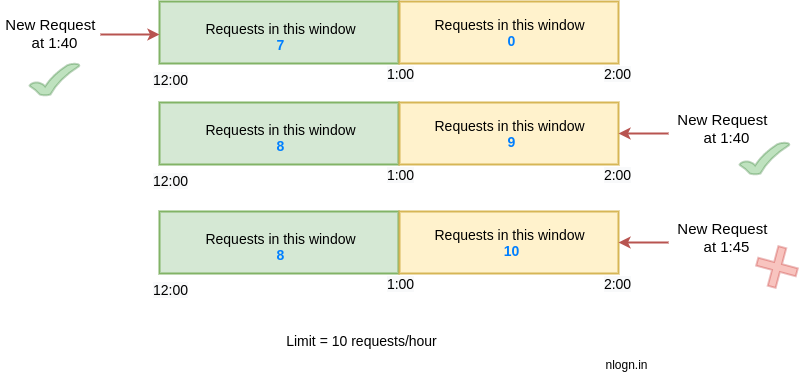
**The disadvantage of the leaky bucket algorithm:**

1. A burst of traffic can fill-up the queue with old requests in a time slot and the new request might starve.
2. It provides no guarantee that requests will be processed in a fixed amount of time.

## **2) Fixed Window**

Fixed window rate limiting algorithm, the timeline is divided into a fixed window(say 1min or 1 hour, etc.) and each window is provided with a counter(to count a number of requests in a particular window). If the value of the counter exceeds the limit, the remaining requests are dropped.  
The counter resets after every window.

Suppose we have a rate limit of 10 requests/hour and have a data model like below.



With the current method in the above example, if a new request arrives at 12:40, we get the count from the bucket(12:00–1:00) which is 7, and if less than our request limit, hence this request will be processed and count of the current window will become 8.

Now assume a case for window (1:00–2:00), a request arrives at 1:40 and the value of the counter in this window is 9, which is less than permissible limit(10), so this request will be accepted and the value of the counter will become 10. Now no more requests in the window (1:00–2:00) will be accepted.

**The advantage of this method is:**

1. It is easy to implement.
2. Less memory requirement since we are storing the only count in a given time window.
3. It ensures more recent requests get processed without being starved by old requests (as the counter resets after every window).

**Cons of this algorithm are:**

1. A single burst of traffic that occurs near the boundary of a window can result in twice the rate of requests being processed. *Suppose, that counter is empty 10 requests spikes arrive at 12:59, they will be accepted and again a 10 requests spike arrives at 1:00 since this is a new window and the counter will be set to 0 for this window. So even these requests will be accepted and sever is now handling 20 requests> 10 requests/ hour limit.*
2. Many consumers waiting for a reset window(ex during peak hour like black Friday sale) can stampede our server at the same time.

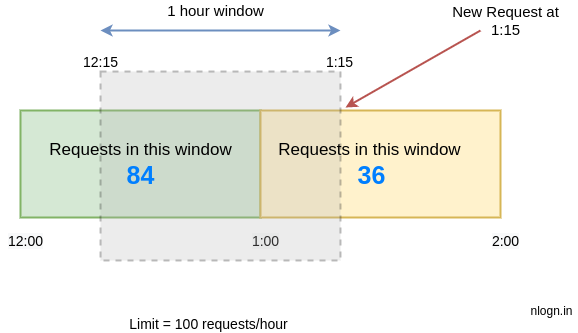
## **3) Sliding Window**

It is based on a fixed window algorithm. Here instead of completely the counter after every window, we use the information from the previous counter to estimate the size of the current request rate for the current window.

In the sliding window, instead of fixed window size, we have a rolling window of time to smooth bursts.

The windows are typically defined by the floor of the current timestamp, so 12:03:15 with a 60-second window length would be in the 12:03:00 window.

Let’s say we want to limit 100 requests per hour on an API and assume there are 84 requests in the time window [12:00–1:00) and 36 requests current window [1:00 to 2:00) which started 15 minutes ago.



Now imagine a new request arrives at 1:15. To decide, whether we should accept this request or deny it will be based on the approximation.

The approximation rate will be calculated like this:

limit = 100 requests/hour

rate = 84 \* ((60-15)/60) + 36

= 84 \* 0.75 + 36

= 99

rate < 100

hence, we will accept this request.

Since the requests in the current windRate Limiting in Distributed Systems

Global Rate Limit

when you are using a cluster of multiple nodes

Sticky Sessions in Load Balancer

The simplest way to enforce the limit is to set up sticky sessions in your load balancer so that each consumer gets sent to exactly one node.

Pros:

Simple

Cons:

The disadvantages include a lack of fault tolerance and scaling problems when nodes get overloaded.

Centralized Data Store

Use a centralized data store such as Redis or Cassandra, to store the counts for each window and consumer

Pros:

More flexible for load-balancing rule

Cons:

increased latency making requests to the data store,

race conditions

race condition:

One of the largest problems with a centralized data store is the potential for race conditions in high concurrency request patterns.

This happens when you use a naïve “get-then-set” approach, wherein you retrieve the current rate limit counter, increment it, and then push it back to the datastore.

One way to avoid this problem is to put a “lock” around the key in question, preventing any other processes from accessing or writing to the counter. This would quickly become a major performance bottleneck, and does not scale well, particularly when using remote servers like Redis as the backing datastore.

A better approach is to use a “set-then-get” mindset, relying on atomic operators that implement locks in a very performant fashion, allowing you to quickly increment and check counter values without letting the atomic operations get in the way.ow [12:15–1:15) are 99 which is less than our limit of 100 requests/hour, hence this request will be accepted. But any new request during the next second will not be accepted.

This algorithm assumes a constant request rate in the (any) previous window, which is not true as there can be request spikes too during a minute and no request during another hour. Hence the result is only an approximated value.

**Pros of the sliding window:**

1. It smoothens the traffic spikes problem we had in the fixed window method, it is easy to implement.
2. It results in an approximate value, but the value is very closer to an accurate valuRate Limiting in Distributed Systems
3. Global Rate Limit
4. when you are using a cluster of multiple nodes
5. Sticky Sessions in Load Balancer
6. The simplest way to enforce the limit is to set up sticky sessions in your load balancer so that each consumer gets sent to exactly one node.
7. Pros:
8. Simple
9. Cons:
10. The disadvantages include a lack of fault tolerance and scaling problems when nodes get overloaded.
11. Centralized Data Store
12. Use a centralized data store such as Redis or Cassandra, to store the counts for each window and consumer
13. Pros:
14. More flexible for load-balancing rule
15. Cons:
16. increased latency making requests to the data store,
17. race conditions
18. race condition:
19. One of the largest problems with a centralized data store is the potential for race conditions in high concurrency request patterns.
20. This happens when you use a naïve “get-then-set” approach, wherein you retrieve the current rate limit counter, increment it, and then push it back to the datastore.
21. One way to avoid this problem is to put a “lock” around the key in question, preventing any other processes from accessing or writing to the counter. This would quickly become a major performance bottleneck, and does not scale well, particularly when using remote servers like Redis as the backing datastore.
22. A better approach is to use a “set-then-get” mindset, relying on atomic operators that implement locks in a very performant fashion, allowing you to quickly increment and check counter values without letting the atomic operations get in the way.e ( an analysis on 400 million requests from 270,000 distinct sources shows only 0.003% of requests have been wrongly allowed)
23. It has very little memory usage: we need to store only 2 numbers per counteRate Limiting in Distributed Systems
24. Global Rate Limit
25. when you are using a cluster of multiple nodes
26. Sticky Sessions in Load Balancer
27. The simplest way to enforce the limit is to set up sticky sessions in your load balancer so that each consumer gets sent to exactly one node.
28. Pros:
29. Simple
30. Cons:
31. The disadvantages include a lack of fault tolerance and scaling problems when nodes get overloaded.
32. Centralized Data Store
33. Use a centralized data store such as Redis or Cassandra, to store the counts for each window and consumer
34. Pros:
35. More flexible for load-balancing rule
36. Cons:
37. increased latency making requests to the data store,
38. race conditions
39. race condition:
40. One of the largest problems with a centralized data store is the potential for race conditions in high concurrency request patterns.
41. This happens when you use a naïve “get-then-set” approach, wherein you retrieve the current rate limit counter, increment it, and then push it back to the datastore.
42. One way to avoid this problem is to put a “lock” around the key in question, preventing any other processes from accessing or writing to the counter. This would quickly become a major performance bottleneck, and does not scale well, particularly when using remote servers like RRate Limiting in Distributed Systems
43. Global Rate Limit
44. when you are using a cluster of multiple nodes
45. Sticky Sessions in Load Balancer
46. The simplest way to enforce the limit is to set up sticky sessions in your load balancer so that each consumer gets sent to exactly one node.
47. Pros:
48. Simple
49. Cons:
50. The disadvantages include a lack of fault tolerance and scaling problems when nodes get overloaded.
51. Centralized Data Store
52. Use a centralized data store such as Redis or Cassandra, to store the counts for each window and consumer
53. Pros:
54. More flexible for load-balancing rule
55. Cons:
56. increased latency making requests to the data store,
57. race conditions
58. race condition:
59. One of the largest problems with a centralized data store is the potential for race conditions in high concurrency request patterns.
60. This happens when you use a naïve “get-then-set” approach, wherein you retrieve the current rate limit counter, increment it, and then push it back to the datastore.
61. One way to avoid this problem is to put a “lock” around the key in question, preventing any other processes from accessing or writing to the counter. This would quickly become a major performance bottleneck, and does not scale well, particularly when using remote servers like Redis as the backing datastore.
62. A better approach is to use a “set-then-get” mindset, relying on atomic operators that implement locks in a very performant fashion, allowing you to quickly increment and check counter values without letting the atomic operations get in the way.edis as the backing datastore.
63. A better approach is to use a “set-then-get” mindset, relying on atomic operators that implement locks in a very performant fashion, allowing you to quickly increment and check counter values without letting the atomic operations get in the way.r.

# Rate Limiting in Distributed Systems

## Global Rate Limit

when you are using a cluster of multiple nodes

### Sticky Sessions in Load Balancer

The simplest way to enforce the limit is to set up sticky sessions in your load balancer so that each consumer gets sent to exactly one node.

Pros:

* Simple

Cons:

* The disadvantages include a lack of fault tolerance and scaling problems when nodes get overloaded.

### Centralized Data Store

Use a centralized data store such as Redis or Cassandra, to store the counts for each window and consumer

Pros:

More flexible for load-balancing rule

Cons:

* increased latency making requests to the data store,
* race conditions

race condition:

One of the largest problems with a centralized data store is the potential for [race conditions](https://en.wikipedia.org/wiki/Race_condition) in [high concurrency](https://en.wikipedia.org/wiki/Concurrency_(computer_science)) request patterns.

This happens when you use a naïve “get-then-set” approach, wherein you retrieve the current rate limit counter, increment it, and then push it back to the datastore.

One way to avoid this problem is to put a “lock” around the key in question, preventing any other processes from accessing or writing to the counter. This would quickly become a major performance bottleneck, and does not scale well, particularly when using remote servers like Redis as the backing datastore.

A better approach is to use a “set-then-get” mindset, relying on atomic operators that implement locks in a very performant fashion, allowing you to quickly increment and check counter values without letting the atomic operations get in the way.