Contents

[Introduction 1](#_Toc76154492)

[Why Rate Limit? 1](#_Toc76154493)

[Benefits: 1](#_Toc76154494)

[Requirements 1](#_Toc76154495)

[Algorithms 2](#_Toc76154496)

[**Token bucket:** 2](#_Toc76154497)

[Leaky bucket: 2](#_Toc76154498)

[Fixed window: 2](#_Toc76154499)

[Sliding window: 2](#_Toc76154500)

[**Sliding window log:** 2](#_Toc76154501)

[**Sliding window counter:** 3](#_Toc76154502)

[Architecture 3](#_Toc76154503)

[Detailed design 3](#_Toc76154504)

[Rules setting and response after being throttled. 3](#_Toc76154505)

[Race condition 4](#_Toc76154506)

[Rate Limiting in Distributed Systems 4](#_Toc76154507)

[Synchronization Policies 4](#_Toc76154508)

[Race Conditions 5](#_Toc76154509)

[Optimizing for Performance 5](#_Toc76154510)

# Introduction

In computer network, rate limiting is used to limit the rate of the requests. Rate limiter is widely used in production to protect underlying services and resources by limiting the number of client requests allowed to be sent over a specified period

Rate limiting refers to preventing the frequency of an operation from exceeding some constraint. In large-scale systems, rate limiting is commonly used to protect underlying services and resources

# Why Rate Limit?

You implement rate limiting primarily for one of three reasons:

1. To prevent a denial of service (intentional or otherwise) through resource exhaustion,
2. To limit the impact (or potential) of cascading failure,
3. To restrict or meter resource usage.

# Benefits:

1. Prevent resource starvation caused by Denial of Service (DoS) attack.
2. Prevent servers from being overloaded. For example, applying rate limiting per user to provide fair and reasonable use, without affecting other users.
3. Controlling flow. For example, preventing a single worker from accumulating a queue of unprocessed items while other workers are idle.

# Requirements

In this article, we are going to assume the requirements of a rate limiter are:

1. Not a client-side rate limiter (reason: easy to bypass or be forged.)
2. Work in a distributed environment
3. High fault tolerance
4. High accuracy
5. Low latency

# Algorithms

Although maintaining a rate can be as simple as counting an occurrence, there are several different algorithms to do it.

# **Token bucket:**

The bucket has a capacity of tokens and the tokens will be refilled at some rate. Each request will attempt to withdraw a token from the bucket, if there are no tokens in the bucket, the service has reached its limit, otherwise, the request goes through.

## ****Leaky bucket**:**

A leaky bucket is similar to a token bucket, but the rate is limited by the amount that can drip or leak out of the bucket. The bucket is like a queue or buffer, requests are processed at a fixed rate. Requests will be added to the bucket as long as the bucket is not full, any extra request spills over the bucket edge is discarded.

## ****Fixed window**:**

Windows are split upfront and each window has a counter. Each request increases the counter by one. Once the counter reaches the threshold, new requests are dropped until a new time window starts. This algorithm is easy to implement but they are subject to spikes at the edges of the window.

## ****Sliding window**:**

The sliding window algorithm can mitigate the problem mentioned in the fixed window algorithm. The idea of sliding window is to keep track of all requests timestamp and calculate whether the counter exceeds in the past fixed period when a request arrives, this algorithm is called sliding window log. Based on this, an optimized algorithm called “sliding window counter” requires fewer operations on the timestamps. I will talk about the difference and implementation detail.

### **Sliding window log:**

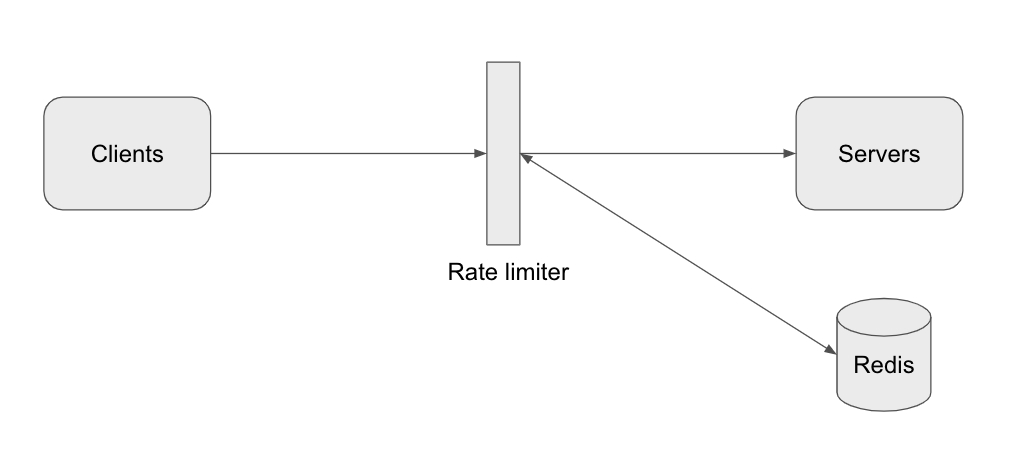
As discussed above, this algorithm keeps track of all request timestamps. When a new request comes in, remove all the outdated timestamps and then count the remaining timestamps. If the counter exceeds the threshold, reject the request. To implement this sliding window log algorithm, we can take advantage of the sorted sets of Redis. We can use ZREMRANGEBYSCORE to count the timestamps. We can even set a TTL equal to the rate-limiting window and let Redis take care of the timestamp removal.

### **Sliding window counter:**

Instead of storing and managing all timestamps, this sliding window counter algorithm is the combination of fixed window and sliding window, and it keeps the benefits of the sliding window while just requires just 2 variables to keep track of the rate and counter. The idea of this algorithm is to use the information from the previous counter to extrapolate an accurate approximation of the request rate of current window. It smoothest the traffic spike

# Architecture

We have discussed the algorithms, now move to high-level design. We are going to store the data in the database, a in-memory storage is likely a good choice because the rate limiter should respond as quickly as possible. Redis is the best option here because it is fast, has a simple API to manipulate the data and supports a time-based expiration strategy.

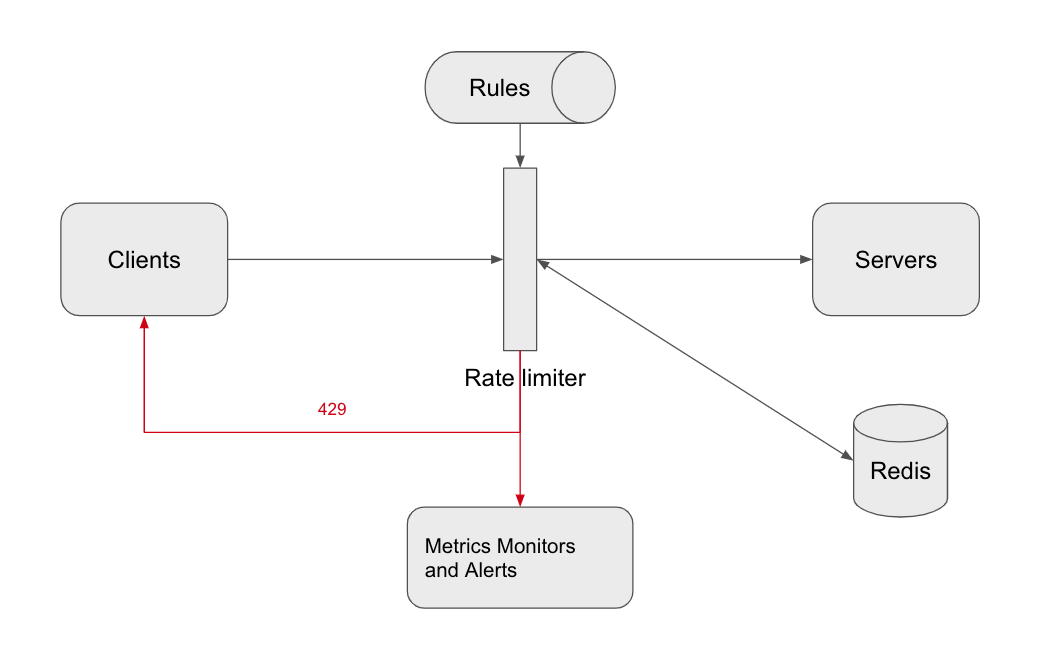


# Detailed design

### Rules setting and response after being throttled.

To support a flexible setting, the rate limiter loads the rules configuration when it starts or has a mechanism to allow uses to update the rules. Here we introduce a config database to persist the configuration.

Once a request is rate limited, the rate limiter should able to notify its client. Suppose we use HTTP protocol, code 429 can be returned and in the response, headers like X-Ratelimit-Remaining and X-Ratelimit-Retry-After can be added to tell the clients more info about the rate limit.



# Race condition

Race condition might occur in a highly concurrent environment with the sliding window log algorithm because it needs to remove, add and count the timestamps. The data becomes stale while multi threads/instances are processing at the same time. However, we can take advantage of the MULTI command. In this mode, all Redis operations can be performed as an atomic action. 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, which prevents the race condition.

# Rate Limiting in Distributed Systems

### Synchronization Policies

If you want to enforce a global rate limit when using a [cluster of multiple nodes](https://konghq.com/learning-center/api-gateway/api-gateways-for-high-availability-clusters), you must set up a policy to [enforce it](https://konghq.com/blog/using-instaclustr-and-cassandra-with-kong/). If each node were to track its rate limit, a consumer could exceed a global rate limit when sending requests to different nodes. The greater the number of nodes, the more likely the user will exceed the global limit.

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

A better solution that allows more flexible load-balancing rules is to use a centralized data store such as Redis o . A centralized data store will collect the counts for each window and consumer. The two main problems with this approach are increased latency making requests to the data store and race conditions, which we will discuss next.

A picture containing graphical user interface

Description automatically generated

# Race Conditions

One of the most extensive 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 issue 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. This model’s problem is that additional requests can come through in the time it takes to perform a full cycle of read-increment-store, each attempting to store the increment counter with an invalid (lower) counter value. This allows a consumer to send a very high rate of requests to bypass rate limiting controls.

A screenshot of a computer

Description automatically generated with low confidence

**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. A lock would quickly become a significant performance bottleneck and does not scale well, mainly 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.

# Optimizing for Performance

The increased [latency](https://konghq.com/blog/observability-kubernetes-kong/) is another disadvantage of using a centralized data store when checking the rate limit counters. Unfortunately, even checking a fast data store like Redis would result in milliseconds of additional latency for every request.

Make checks locally in memory to make these rate limit determinations with minimal latency. To make local checks, relax the rate check conditions and use an eventually consistent model. For example, each node can create a data sync cycle that will synchronize with the centralized data store. Each node periodically pushes a counter increment for each consumer and window to the datastore. These pushes atomically update the values. The node can then retrieve the updated values to update its in-memory version. This cycle of converge → diverge → reconverge among nodes in the cluster is eventually consistent.

A picture containing text

Description automatically generated

The periodic rate at which nodes converge should be configurable. Shorter sync intervals will result in less divergence of data points when spreading traffic across multiple nodes in the cluster (e.g., when sitting behind a round robin balancer). Whereas longer sync intervals put less read/write pressure on the datastore and less overhead on each node to fetch new synced values.