**DESIGN DOCUMENT FOR API RATE LIMITING SYSTEM**

**Prepared By:** Musa Shaikh

**Date:** 29/07/2021

**API RATE LIMITER:**

**What is api rate limiter**

API limiting, which is also known as rate limiting, is an essential component of Internet security, as DoS attacks can tank a server with unlimited API requests. Rate limiting also helps make your API scalable. If your API blows up in popularity, there can be unexpected spikes in traffic, causing severe lag time.

**Why api rate limiter?**

1. Rate limiting protects your APIs from inadvertent or malicious overuse by limiting how often each user can call the API. Without rate limiting, each user may make a request as often as they like, leading to “spikes” of requests that starve other consumers.
2. Consider you have product deployed on the server. It has multiple apis and you want to make money out of it, for that you want users to give access to those apis for some time to try.

**Advantage of rate limiting: -**

1. User Experience: - good quality of service for every consumer.
2. Security
3. Operational cost

**Different kinds of rate limiting: -**

1. User based: - for a particular user how many requests can process or for a particular time duration.
2. Concurrent: - for a particular user how many parallel connections are allowed.
3. Location/IP: - A device with a specific ip address or region how many requests to process.

**Requirements and Goals of the System**

* 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
* Highly available
* Not introducing substantial latencies

**Algorithm: -**

**Below are several rate limiting algorithm**

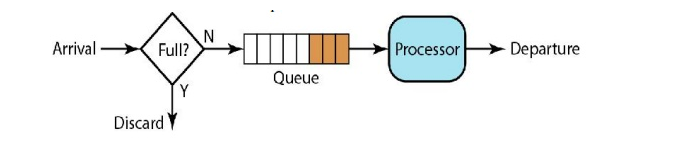
1. **Token bucket: -** Suppose there are a few tokens in a bucket. When a request comes, a token must be taken from the bucket for it to be processed. If there is no token available in the bucket, the request will be rejected, and the requester must retry later. The token bucket is also refilled per time unit. In the token bucket algorithm, the request processing rate is not capped, which means it only guarantees an average processing rate will not exceeds the maximum rate. But in some period, the real-time processing rate can be higher than maximum.

**Pros: -**

It is memory efficient as we are saving less amount of data per user.

**Cons: -**In distributed environment it causes race condition, because for same user two requests are coming on two different servers, it will try to update the same record in token bucket.

1. **Leaky Bucket: -** It provides a simple, intuitive approach to rate limiting via a queue, which you can think of as a bucket holding the requests. When registering a request, the system appends it to the end of the queue. Processing for the first item on the queue occurs at a regular interval or first in, first out (FIFO). If the queue is full, then additional requests are discarded (or leaked).



**Pros:** -

* it smooths out bursts of requests and processes them at an approximately average rate.
* It is memory efficient for each user, given the limited queue size.

**Cons: -**

* A burst of traffic can fill up the queue with old requests and starve more recent requests from being processed.

1. **Sliding Log**

Here we will append the request timestamps to array, for each new request we will remove the outdated requests which did not come under the new time window and will check if number of request in the new window is exceeding the rate limit. If it

Ex:

Suppose we have set the api rate limit to 10Req/Min

User U1: [“10:00:01”, “10:00:15”, “10:00:35”, “10:01:05”, “10:01:31”, “10:01:35”].

Consider above request came for user U1.

When a new request comes at 10:02:05, so the current time window will be (10:01:05 to 10:02:05), outdate request in the array will be [“10:00:01”, “10:00:15”, “10:00:35”] that we will remove.

After removing outdated requests, we will have three requests in the new time window, which is less than the rate limit, so we can process the new request.



1. **Database**

Persistent memory stores like mysql is a bad idea because the time taken for disk seeks is high enough to hamper the rate limiter granularity.

Let’s say that we are using a single mysql instance to store the request counts and mysql takes 1ms to process 1 request, which means we can achieve a throughput of 1000 req/sec. But a rate limiter using in-memory cache takes around 100nanosec (a main-memory access) to process 1 request, which implies a granularity of around 10Mreq/sec can be achieved.

Cassandra is a distributed key-value store. It 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 in memory and distributed database. Memory size is limited by the maximum memory of your server. We can also shard the data to several Redis instances, running on multiple servers

1. **Rate limiter in distributed environment**

Suppose we have distributed setup with multiple nodes, where application and rate limiters are distributed across multiple servers.

With above algorithm will face

1. Inconsistency
2. Race condition

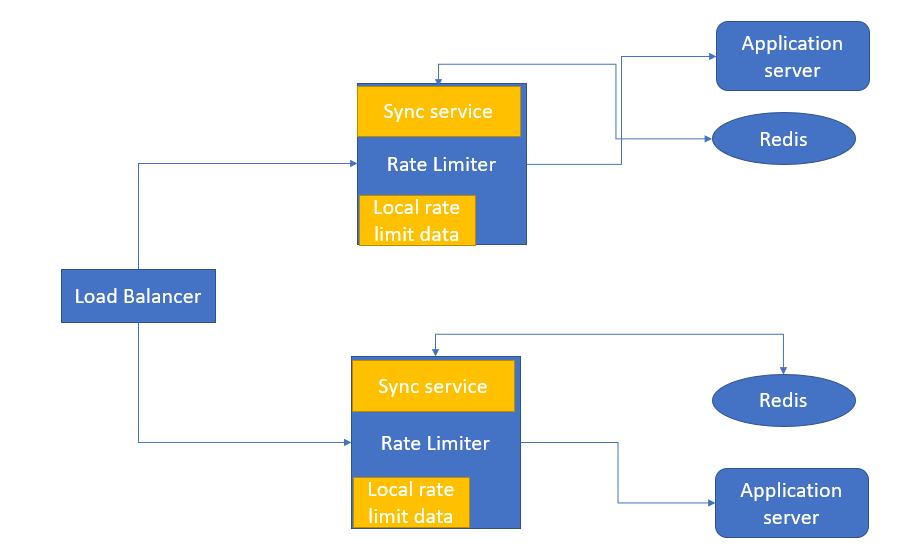
**Inconsistency in rate limiter handle through sticky session.**

In sticky session a user will redirect all the requests to same server.

The problems with sticky session are, it is not well balanced and fault tolerant.

**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).



**Distributed rate limiting system.**

**Sync Service: -** This service will update rate limit data from Redis database to rate limiter local and vice versa. So that for each new request rate limiter does not need to fetch data from Redis.