Rate Limiter

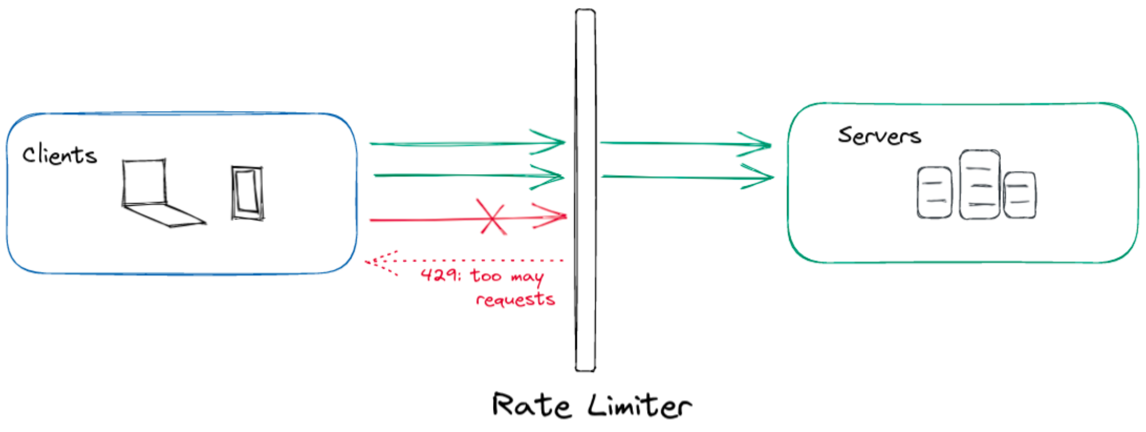
Use Cases:

1. It protects the server from DOS (denial-of-service) attacks, by limiting the number of requests made from a particular Ip address or user. Ex, Twitter limits the number of tweets to 300 per 3 hours. Basically, It reduces the load over servers.
2. Reduce the cost for the company, as some may be using external API which required some fees.

Types of rate limiters:

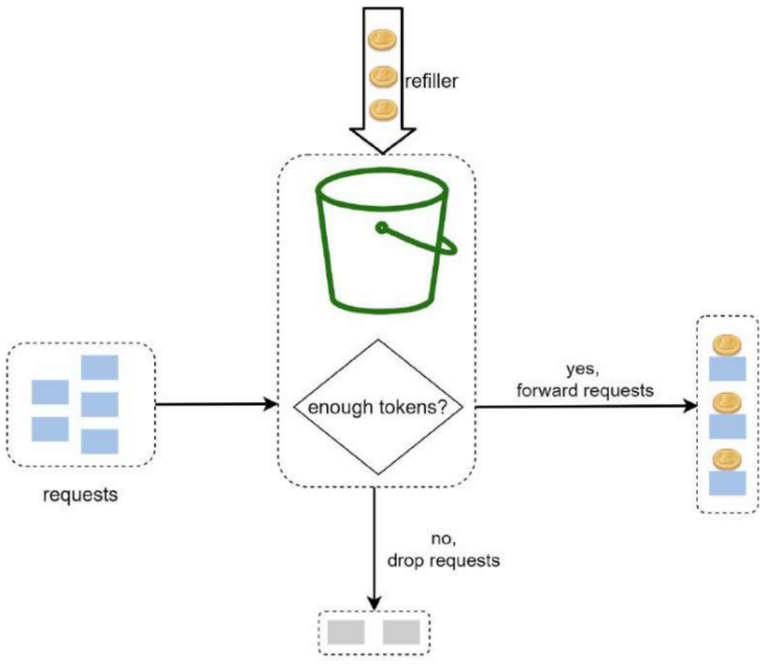
1. **Client-side:** In this, we limit the number of requests sent by the user. Not good as we might not have full control on the client side.
2. **Server-side:** In this, we limit the number of request to a particular server, if more request comes, either block them or transfer it to other free servers.
3. **Middleware rate limiter:** In cloud architecture, the user requests may go to any of the servers available, due to which using a rate limiter at all of the servers is a costly and not efficient way, that's why we can have our rate limiter on **API gateway** as a middleware.

   💡 API gateway is a fully managed service that supports rate limiting, SSL termination, authentication, IP whitelisting, servicing static content, etc.



Algorithms for rate limiting

1. Token bucket algorithm



In this algorithm, we have fixed-sized buckets in our rate limiter system. The bucket is filled with tokens and each request to the server consumes one token. If there is no token present in the bucket, the request is dropped.

A bucket is also refilled with tokens at a predefined rate.

So this algorithm has two parameters: 1. Bucket size, 2. Refilling rate.

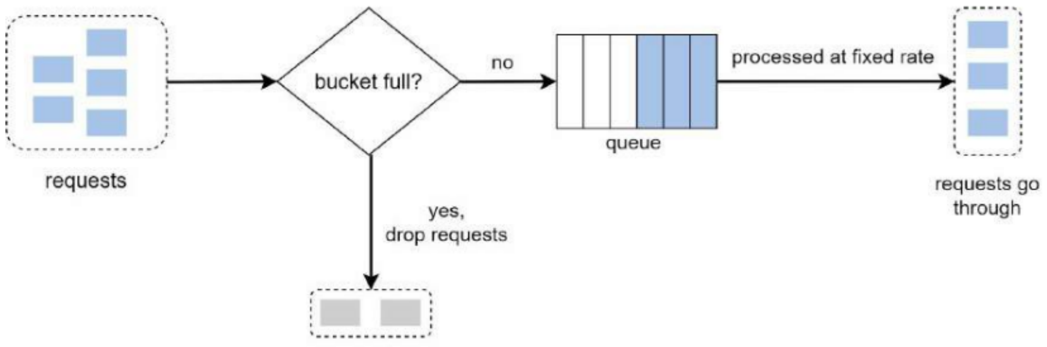
Pros :

1. Easy to implement and memory efficient.

Cons:

1. challenging to tune both parameters properly.

2. Leaking bucket algorithm



In this algorithm, there is a bucket that is filled with requests. And there is a queue that process requests at a constant rate. So this algorithm basically converts the bursty traffic into constant rate traffic.

3. Fixed window counter algorithm

In this algorithm, we have a fixed time window, in which we put a maximum request limit. All extra requests are dropped until the next time window starts.

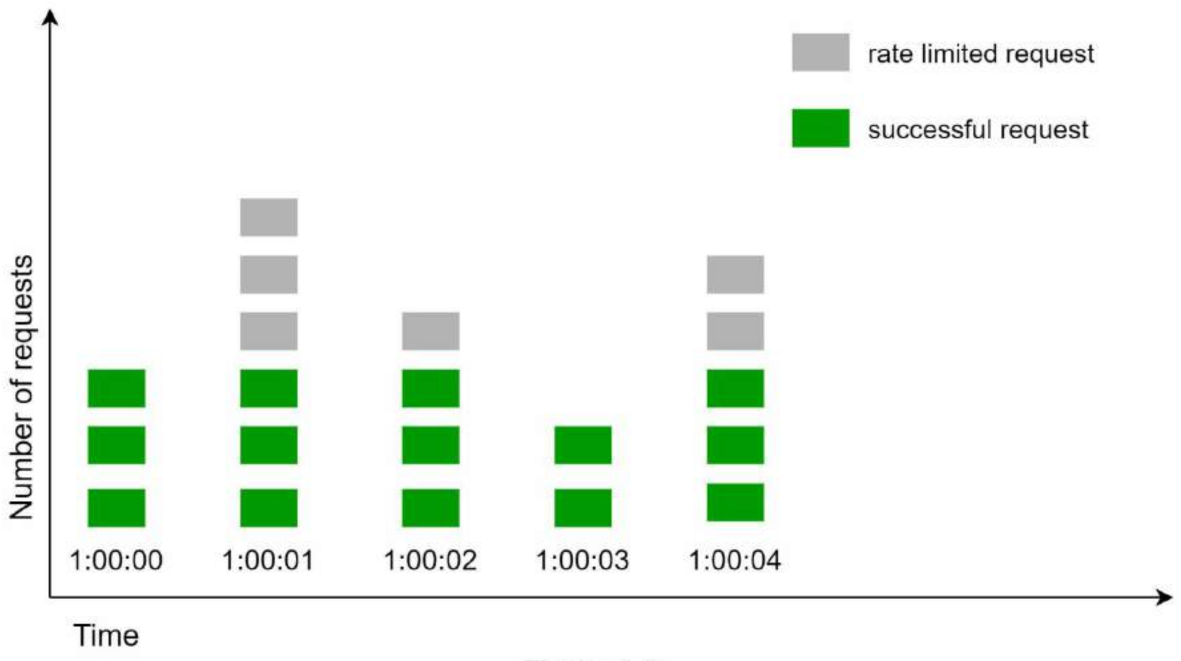


Figure 4-8

Problem:

If all requests for the previous window were served in the second half, and all requests for the current window are served in the first half, it effectively results in serving twice as many requests within our time window. ⇒ **Spike in traffic!**

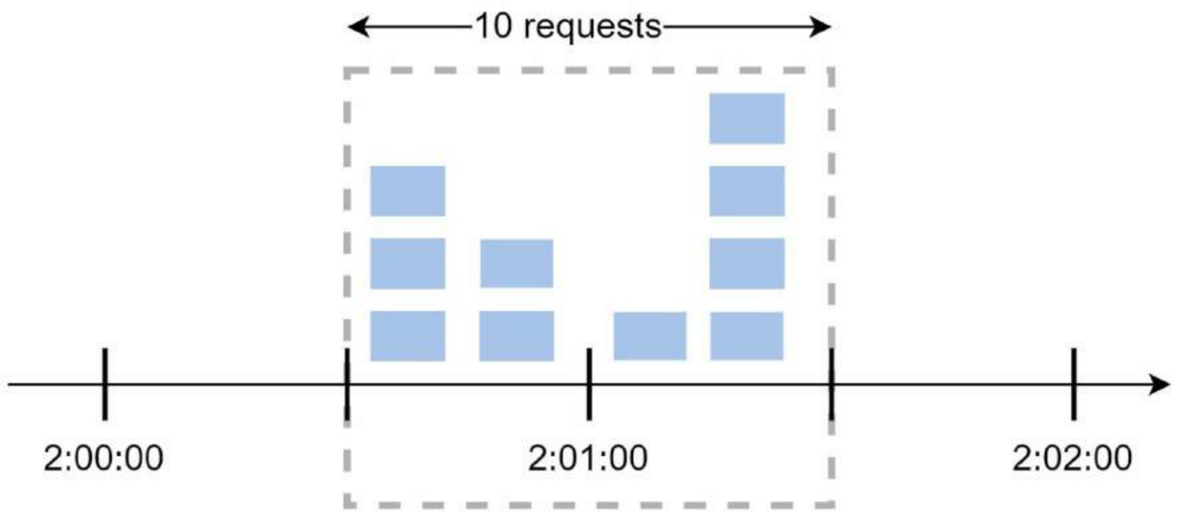


Figure 4-9

4. Sliding window log algorithm

In this algorithm, along with user request we keep track of request timestamp. This algorithm has two parameters: 1. Time window size, 2. Max requests count. So when new request comes, It will get accepted only if:

1. After removing all requests logs with timestamp < current timestamp - time window size, if number of requests in window is < max request count. **Else dropped.**

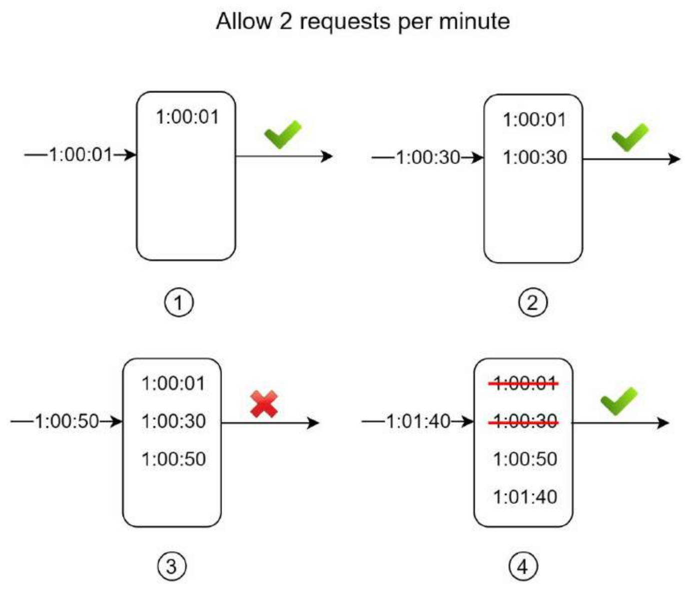


Figure 4-10

Pros:

1. It is better than fixed window algoirthm, because it does not have a window border issues.

Cons:

1. It’s not memory efficient as we need to need track of all the logs in time window.

4. Sliding window log algorithm

It’s a mix of above two algorithms, here,

1. we keep fixed windows.
2. To become memory effecient, we dont use timestamps of each request instead we keep just the count of request served in previous fixed window.

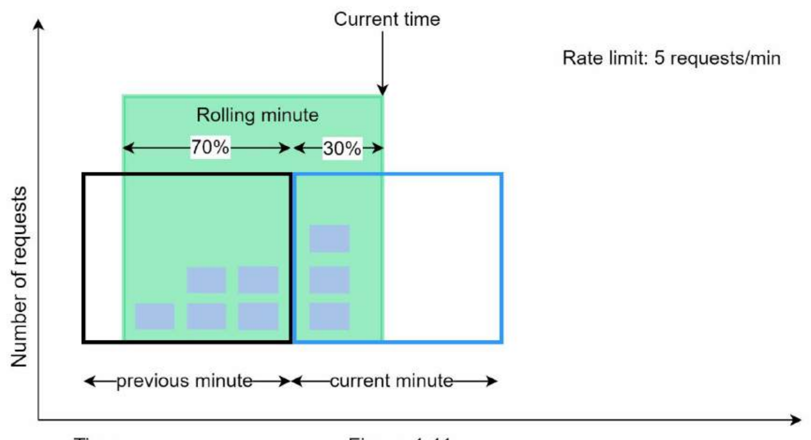


Figure 4-11

3. Now it we have decided **rate limit to be 7 requests/min**, and in current time window if we have served 3 requests. Now if new request comes, to avoid the border problem of fixed window, we calculate the current slidign window size liek this,

- Requests in current window + requests in the previous window * overlap percentage of the rolling window and previous window

4. So in our case, 3 + 5*(0.7) = 6.5 = 6 requests, since its less than 7 we accept new request.
5. Here, we assume that requests in previous window are evenly distributed over time but it’s not bad as **only 0.003% of requests are wrongly allowed or rate limited among 400 million requests.**

Rate Limiter in Distributed Systems

Two challenges:

1. Race condition

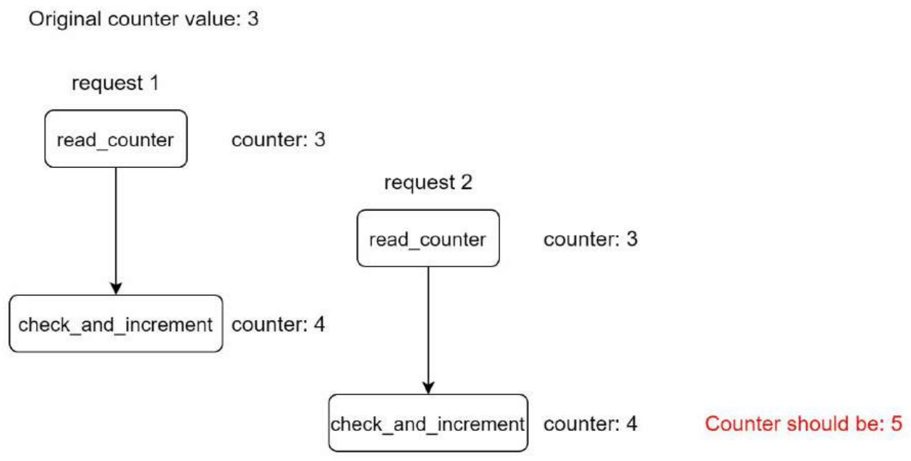


Figure 4-14

2. Synchronization

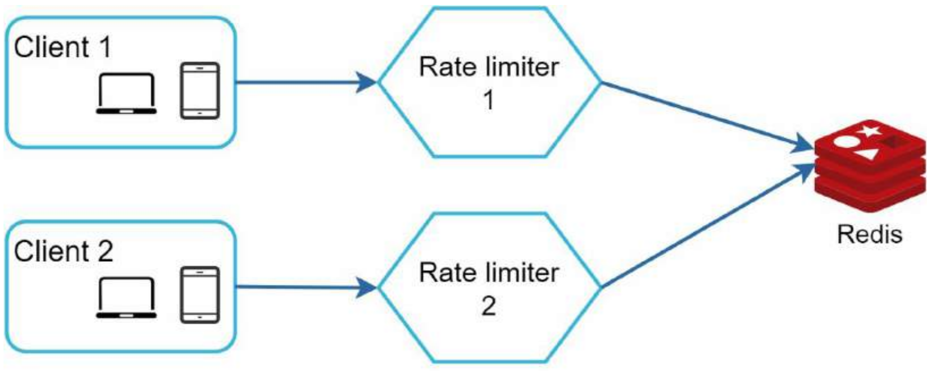


Figure 4-16