**DESIGN DOCUMENT FOR API RATE LIMITING SYSTEM**

**Prepared By:** Arjuna Pandian Nehru

**Date:** 29/07/2021

****API Rate Limitation**** means ****API**** usage quota is defined per user per interval(hour, minute, seconds, day, year). It’s a limitation on frequency of usage of API based on some constraints.

IMG_256

**Rate Limitation - WHY??**

- A single user can hit a lot of requests and that can starve the server computation resources. This can also affect other users’ response times.

- To prevent overuse of services by defining maximum quota. This can also help in reducing malicious attacks.

**How to Implement/Design Rate Limiter?**

There are different levels at which one can design rate limiters. Listing a few…

- Rate limiter in a single machine, single threaded scenario

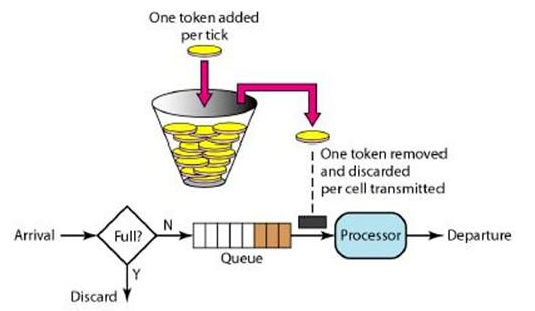
- Rate limiter in a single machine, multi threaded scenario — Handling race conditions

- Rate limiter in a distributed scenario —Distributed Cache Usage like Redis.

- Rate limiter from client side —Prevent network calls from client to server for all the excess requests.

There are different designs to implement rate limiters, some of which are described below…

**Token Bucket:** In this for each user, a token will be filled in some storage(preferred some in memory database) for particular user. Token contains a timestamp with quota limits. At every request this limit is checked whether the given user has a quota left or not.



User\_id1 : 06: 01: 00 -10

(Considering Rate limit as 10 request per min)

Suppose the user requests the api at 06: 01: 08 then in db , entry will be updated to 06: 01: 00 -9. So within this minute we can hit the api 10 times. Once this limit goes to 0 and users request the api, then we can error like too many requests or some quota exhausted.

Note: At every minute token is updated/refilled in storage.

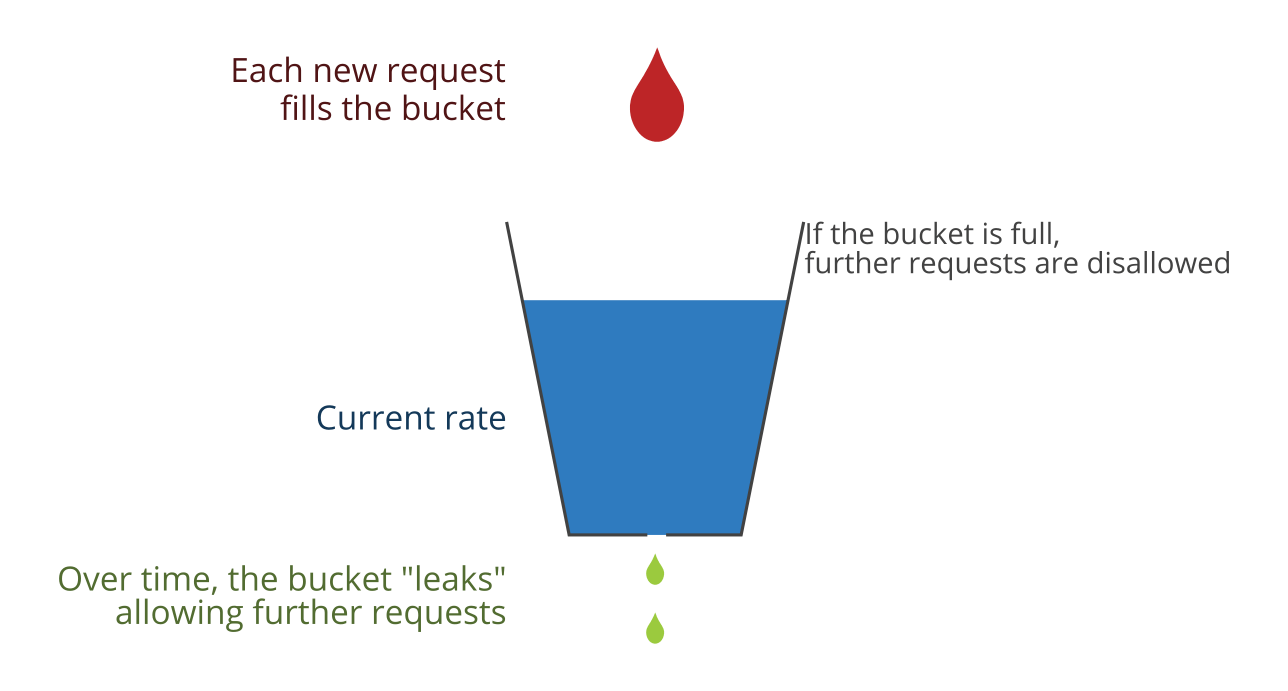
Token Bucket is also considered as memory efficient as a single entry needs to be maintained per user in storage. Token Bucket fails with distributed systems.

**Leaky Bucket:** Here a Queue is maintained that will hold the requests requested. Queue as usual will serve in FIFO. We can also compare this approach with pub sub. Requests will be appended(published) at the end and requests are getting processed(subscribed) from the start.

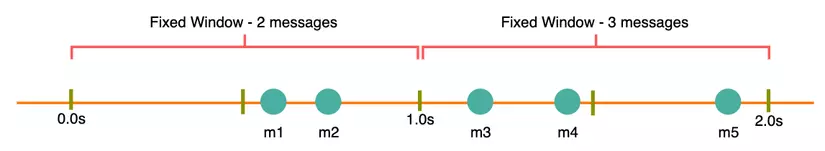
- Here Queue size will be fixed and requests that come after the queue is full, will be discarded/leaked. Leaky Bucket strategy will process the request at some average/constant rate.

- When compared to Token Bucket, Leaky Bucket is able to work efficiently with Distributed System also.

- If some requests take longer than expected or got stuck. Then our server will face the starving for new requests.



**Fix Window Counter:** In this we will keep the window for each user based on the interval. That time will increment the count with each request. Once the threshold is reached, it will reject the requests.



Suppose we allow the 100 request per hour for user:

So the window we will create like:

User 1: 07: 00: 00–0

Then till 07:59:59 we will be able to serve 100 requests.

User 1: 07: 59: 00–100

Problem with this approach- is that sometimes too much load comes at the edges of the time window. With that service will be overused for that particular slot and we can say consistency of load distributed over the entire window is disrupted.

**Sliding Logs:** This algorithm is synced with real time verification of requests within interval. In this we keep on appending requests with timestamps in some storage. Once we got new request, we will check how many requests we had served in a given interval based on Rate Limiter. If requests exceed the threshold we will reject otherwise we will serve the request.

User 1: [ Timestamp1, Timestamp2, ……, Timestampn]

API Rate Limitation Rule - 20 requests per minute

A new request came, the first system will check how many requests are there in the last 1 minute. If that count breaches the threshold then will reject it.

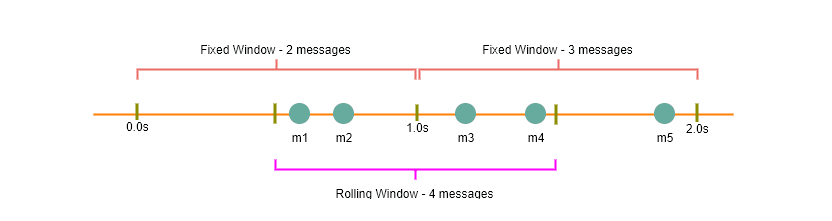
This algorithm works precisely and also able to overcome the boundary conditions of a fixed window. As it will check the request count based on the current timestamp only instead of the fixed window.

Drawbacks:

1.Memory Inefficient : Lot of space is required to keep the timestamp for each request.

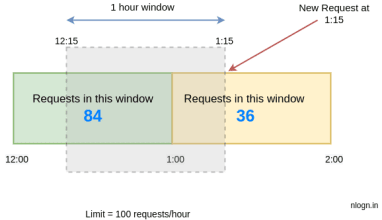
2.Computation Time is wasted on calculation of requests served in a given interval(Need to calculate based on current timestamp).

**Sliding Window Counter:** This is an optimised version of Sliding Logs where we will reduce the space requirements. Here while storing the logs for timestamp, if we get many requests at a particular timestamp.Then we will increase the update the same entry or we can store like Timestamp\_count(request count)



Challenges and Potential solutions of Rate Limiter with Distributed systems:

Till now we have discussed a few ways of implementing Rate Limiter. But these solutions will fail when we work with distributed systems. With distributed systems at the same time requests can come to any server.



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.

A general way is to use two hashmap as counters. To check if a window is valid, we use a map needs to store (char, count) for chars in t. And use counter window for the number of chars of t to be found in s. If window contains all the keys in needs, and the value of these keys is greater than or equal to the value in needs, we know that window meets the requirements and can start moving the left pointer.

Refinement pseudocode above.

string s, t;

// Two pointers

int left = 0, right = 0;

string res = s;

// Initialize the map

unordered\_map<char, int> window;

unordered\_map<char, int> needs;

for (char c : t) needs[c]++;

// The number of characters that meet the requirement

int match = 0;

while (right < s.size()) {

char c1 = s[right];

if (needs.count(c1)) {

window[c1]++; // Add to window

if (window[c1] == needs[c1])

// The number of occurrences of the character c1 meets the requirement

match++;

}

right++;

// When we found a valid window

while (match == needs.size()) {

// Update res here if finding minimum

res = minLen(res, window);

// Increase left pointer to make it invalid/valid again

char c2 = s[left];

if (needs.count(c2)) {

window[c2]--; // Remove from window

if (window[c2] < needs[c2])

// The number of occurrences of the character c2 no longer meets the requirement

match--;

}

left++;

}

}

return res;

Few Challenges and solutions:

1.Inconsistency

2.Race Condition

1. Inconsistency:

Requests are made from different regions with underlying Servers. Each Server will have a Rate Limited Module. Suppose at the same moment the request hit the Rate Limiter Module, in response it will check the limit in some storage. Now suppose from the available quota only 1 is left.

But each server got information that limited is still available. So in this case both requests are executed and our system fails to handle it.

Potential solution: Design system with sticky sessions approach. In this the same user will always serve the particular server always. So Inconsistency will be eliminated as single call to storage needs to be done at a time.

Problem with above solution: Users can expect latency as it should be handled by only a single server.

1. Race Condition:

As the same data can be available for multiple servers for a given user. Now from server one a read + update operation is requested. To get it done , first that data is locked(for reading also) till operation is completed. Now when the second server will try to read the data , it has to wait till the lock is released by the first server. So this will create a Race Condition.

Here distributed system Rate Limitation works but there is a lot of latency experience by users as servers need to wait for locks release.

**How can we make API Rate Limiter Performance Optimized?**

1.We can give relaxation to the system to allow some failure percentage. By this we can avoid locks and also avoid sticky sessions.

2.By using server local storage or session that will regularly query the storage(database) and sync the contents locally. This avoids making database calls for each request.

**Reference:**

1. Youtube Tech Dummies Channel System Design Series
2. https://blog.getambassador.io/rate-limiting-a-useful-tool-with-distributed-systems-6be2b1a4f5f4