RATE LIMITER

A rate Limiter is used to control the rate of traffic sent by a client or a Service.

A rate Limiter limits the number of client requests allowed to be sent over a specific period. If the API request count exceeds the threshold, defined by the rate Limiter, all access Calls will be blocked.

For example:-

*. A user can ~~oth~~ only post up to 10 times in one minutes or allowed 2 posts per second.

* We can create upto 10 accounts per day from the same ip address.

*. We can claim reward up to 5 times per week ~~form~~ from the same device.

Benefits of rate limiter

*. It prevents DoS (~~Denial~~ Denial of service) attack. Like Twitter limits 300 tweets per 3 hours. Google Doc API limits up to 300 read request per user per 60 Seconds.

A rate Limiter prevents DoS attacks either intentional or unintentional by blocking the access Calls.

* Reduce costs:- by limiting access requests. It is important for the company which is using third party API and charge on per call basis. it limits the api calls and reduce the costs.

* Rate limiter prevents the server being overLoaded. To reduce the Server load the rate limiter is used to ~~access out~~ filter out excess request caused by bots or user's misbehaviour.

So, now the next point is, should we write a seperate service for rate limiter or should it be implemented in the same application code?

It depends on the application, tech-team and tech stack where exactly we want to implement rate limiter.

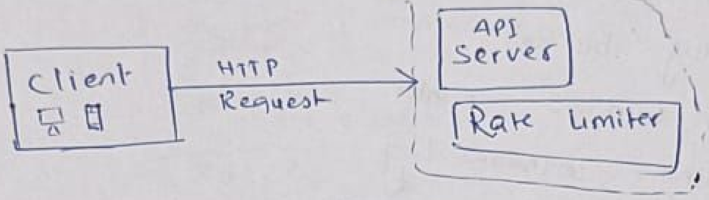
Next point is, where to put rate to limiter server side or client side?

We can implement the rate limiter either at server| client side. We see the scenerio ad and put the rate limiter accordingly.

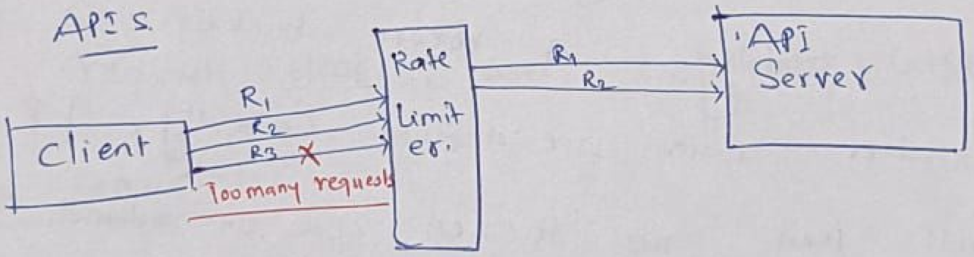
Server side | client side / middleware.

* Client Side implementation is unreliable place to enforce the rate limiting because client request can easily be forced by ~~mulin~~ milicious actors.

*. Server side implementation:- Placing the rate limiter at server side is not a good design because for every request call client connect to server.



Besides the client and server side implementation, there is an alternative way.
instead of putting the rate limiter at api server, we create the rate limiter at middle ware, which throttles request to your APIs.



As shown in above diagram. ~~test~~ lets assume API server allows 2 request per second. Client sends 3 requests in 1 second. 1st 2 request will pass through the rate limiter and goes to API server and 3rd request stops at rate limiter and return HTTP response (status) code 429. that indicates the user sends too many requests.

Suppose you are using microservice based architecture and you are using authentication Serves at ~~[illegible]~~ middleware. We will place the rate limiter parallel to it.

Algorithm for Rate Limiting

- Token bucket
- Leaking bucket
- Fixed window counter
- Sliding window log
- Sliding window Counter.

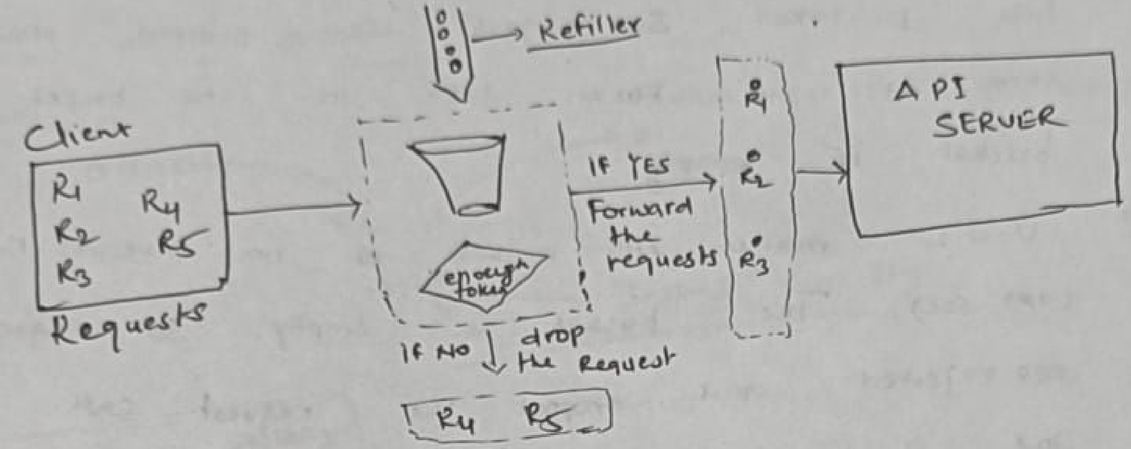
Token bucket Algorithm

Token bucket algorithm is widely used for rate limiting. A token bucket is a Container with pre-defined capicity. If bucket is full then no tocken can be added. If bucket is empty for that user, it reject for that request. Each request consume one token.

Every user has a seperate bucket, When any request made for that user, we remove 1 token from the bucket. After the particular time interval we refil the bucket with appropriate token.

~~Lates~~
Lats take the example :-

if bucket capicity is 4. and we refil the bucket with 2 token per second.



The bucket has a token with predefined capicity. when any request comes it takes the token from bucket and proceed further. if there is no token in the bucket then request drops, user can retry later.

Usecase Example :-

Suppose the bucket capicity = 3 and rules are 3 requests per user per minute.

Step1 :- User1 make a call at 00 time interval. ~~Current~~ currently bucket is full, so user1 can take a token and request will proceed further. Now the bucket capicity will be update ~~capic~~ token = 2.

Step2: User1 makes 2nd request: at time interval 12:00:10 (10th Second), the bucket has 2 token, so request can proceed further. Now, 1 token left in the bucket.

Step3: User1 makes 3rd request. At time interval 12:00:30 (30th sec), the bucket has 1 token, So request can proceed. Now, there is no token left in the bucket. and bucket is empty.

Step4. User 1 makes 4th request, At time interval 12:00:40 (40th sec). The bucket is empty. So request ~~are~~ rejected and dropped the request call. And it return the code 429 (status code) that indecate user ~~can~~ has sent too many request in the given amount of time.

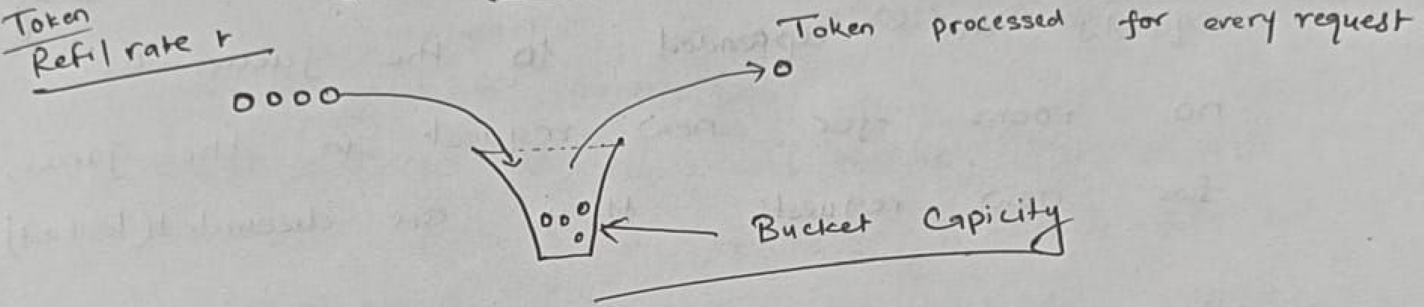
Step5 After 1 minute, the bucket will refil with 3 token. and the same steps will be continue.

In Simple word,
In The token bucket algorithm, we process a token from the bucket for every request. New token are added into bucket with rate r. The bucket can hold maximum b token. If request comes and bucket is full then it discarded.

The token bucket algorithm takes two parameter:

→ Bucket Size - Maximum number of token allowed in the bucket

→ Refil. — No of token put into the bucket every Second.



Code in golang

Pros:

→ The algorithm is easy to implement.

→ It is memory efficient.

→ Token bucket allows burst of traffic for Sort periods

Cons:

→ Two parameters are needed bucket Size and token refil rate. It would be challenging to tune them properly.

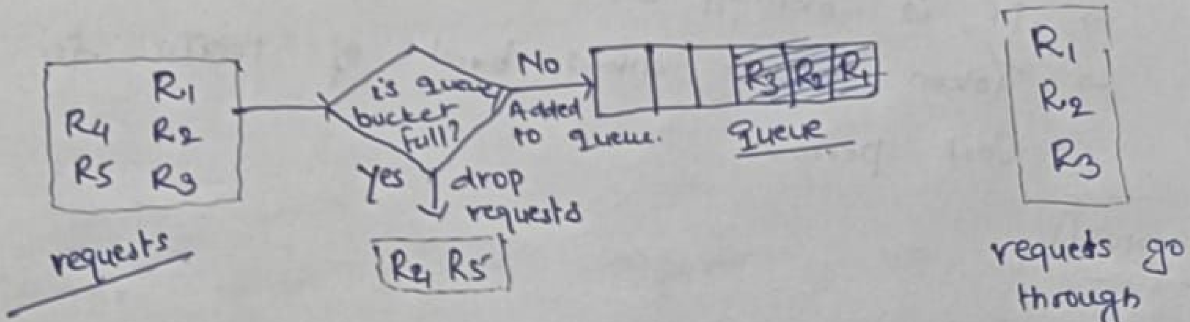
Leaking bucket algorithm

A leaking bucket algorithm is simple and intuitive way to implement rate limiting using a queue. It is using First-in-first-out (FIFO) queue. If queue is not full, incomming requests are appended to the queue, if no room for new request in the queue, for new requests they are discarded (leaked).

→ When the request arrives, the system checks if queue is full. If it is not full then request is added to the queue.

→ If queue is full, request is dropped.

→ requests are pulled from the queue and processed at regular intervals.



The leaking bucket algorithm takes two parameter.

→ Bucket size → It is equal to queue size. The queue holds the requests to be processed at fixed rate.

→ Outflow Rate :- It defines how many requests to be processed at fixed rate.

Pros:-

→ Memory efficient given the Limited queue Size.
→ Requests are processed at a fixed rate, So it is suitable where stable outflow rate is needed.

Cons:

→ Two parameters, so defficult to manage and tune them properly.
→ In case of burst of traffic, the queue fills with old requests if they are not processed in time, new requests will be rate limited.

Fixed window counter algorithm :-

→ The algorithm devide the timeline into the fixed-sized time windows, assign the counter for each window.

→ Each request increment the Counter by one.
→ If counter reaches the pre-defined threshold, new requests are dropped untila the new ~~t~~ time window starts.

Pros:
→ Memory efficient
→ Easy to understand
→ Resetting available quota at the end of the unit time window fits certain use case.

Cons:-
→ If Spike in trafic at the ~~ago~~ edge of the window, could cause more request then allows the quota.

Sliding window log algorithm:

The major issue in the fixed window counter algorithm, It allows more requests to go through at the edge of the window. The sliding window log algorithm fix that issue.

→ The algorithm keep track the request timestamp. timestamp kept in cache such as sorted set of redis.

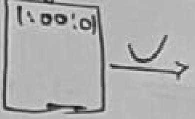
→ When new request comes in, it removes the all outdated request. Outdated timestams means these older then the start of current time window.

→ Add timestamp of new request to log.

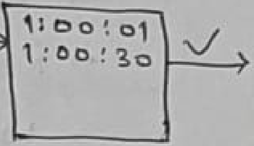
→ If the log size is less then or equal to the allowed count, request is accepted otherwise request is rejected.

For example

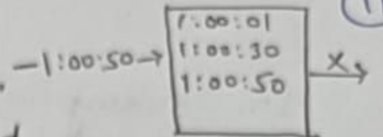
→ New request comes at 1:00:01, at this time log is empty, thus request is allowed → 1:00:01



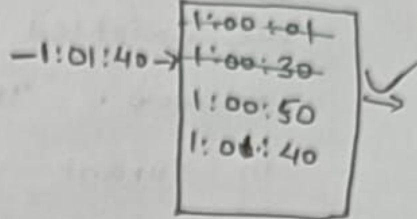
→ New requet at 1:00:30, the timestamp is inserted. After insertion, check → 1:00:30 the Logsize is not greater then the allowed count. here, log size is 2, that is equal to the allowed count. So, the request is allowed.



→ New request at 1:00:50, timestamp inserted. log size(3) is greater then the allowed time (2), so request is rejected.



→ New request at 1:01:40. The requests in the range [1:00:40, 1:01:40] are within the latest timeframe, but requests sent before 1:00:40 are outdated so, two outdated timestamps (1:00.01, 1:00:30) are deleted from the log. ~~After~~ removing ~~th~~ it the logsize becomes 2 so request is accepted and increament the counter.



Pros:

→ It is very accurate. In case of rolling window. requests will not be exceed the Limit.

Cons.

→ It Consume Lots of memory because even if request is rejected the timestamp might still in the memory.

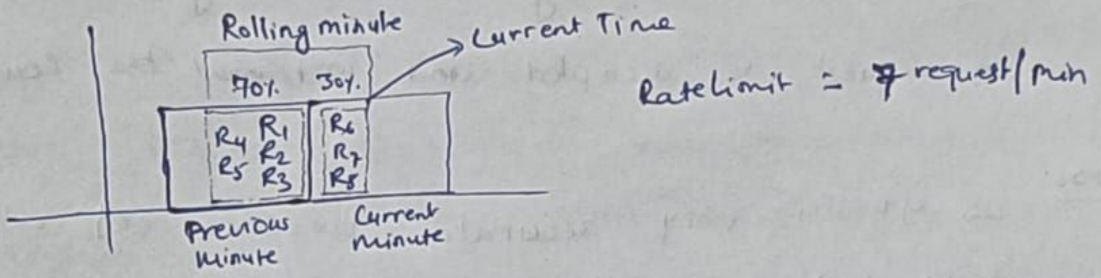
Sliding Window Counter algorithm

This algorithm is the hybrid approach with the Combination of fixed window counter and sliding window log.

Fixed window counter is not robust solution because it can't correctly handle brust request. Sliding window log is powerfull solution but it enforce a hard time Limit on every time window. So, it is not memory efficient Solution.

In Sliding window we don't know how many requests are in it. for this solution, we have to calculate the weighted counter for the previous fixed time window. Then estimate how many requests are in current Sliding window.

Assume, the rate Limiter allows a maximum of 7 requests per minutes. 5 requests in Previous minutes and 3 requests in current minutes.



A new request arrives at 30% position in current minute. The no of request in rolling window is calculated using :-

→ Request of current window + Request of previous window * over lap percentage of rolling window and Previous window

$3 + 5 \times 70\% = 3 + 5 \times 0.7 = 6.5$ requests. we can ~~or~~ rounded up to 6 requests.

Since the rate limiter allows 7 requests/minute. So limit will be reached after receiving 1 More request

Pros

memory efficient
no fixed time window

Cons.

1, calculate the approximations of the actual rate.

Rate Limiting Rules are :-

```
Domain : messaging
Descriptors :
  - key : message_Type
    Value : marketing
    rate limit -
      unit - day
      request_per_unit : 5
```

→ In this example System is Configured to allow maximum 5 marketing messages per day.

Another example :

```
domain: auth
descriptors:
  - key: auth_type
    value: Login
    rate Limit :
      unit : minute
      request_per_unit : 5
```

This rules shows the clients are not allowed to login more then 5 times in 1 minute.

— The rules are generally written in the configuration file and saved in the disk.

In case of the request is rate limited, API returns HTTP response code 429 (too many requests)

Detailed Design of rate Limiting :-

→ We write the rate Limiter rule and store it to disk.

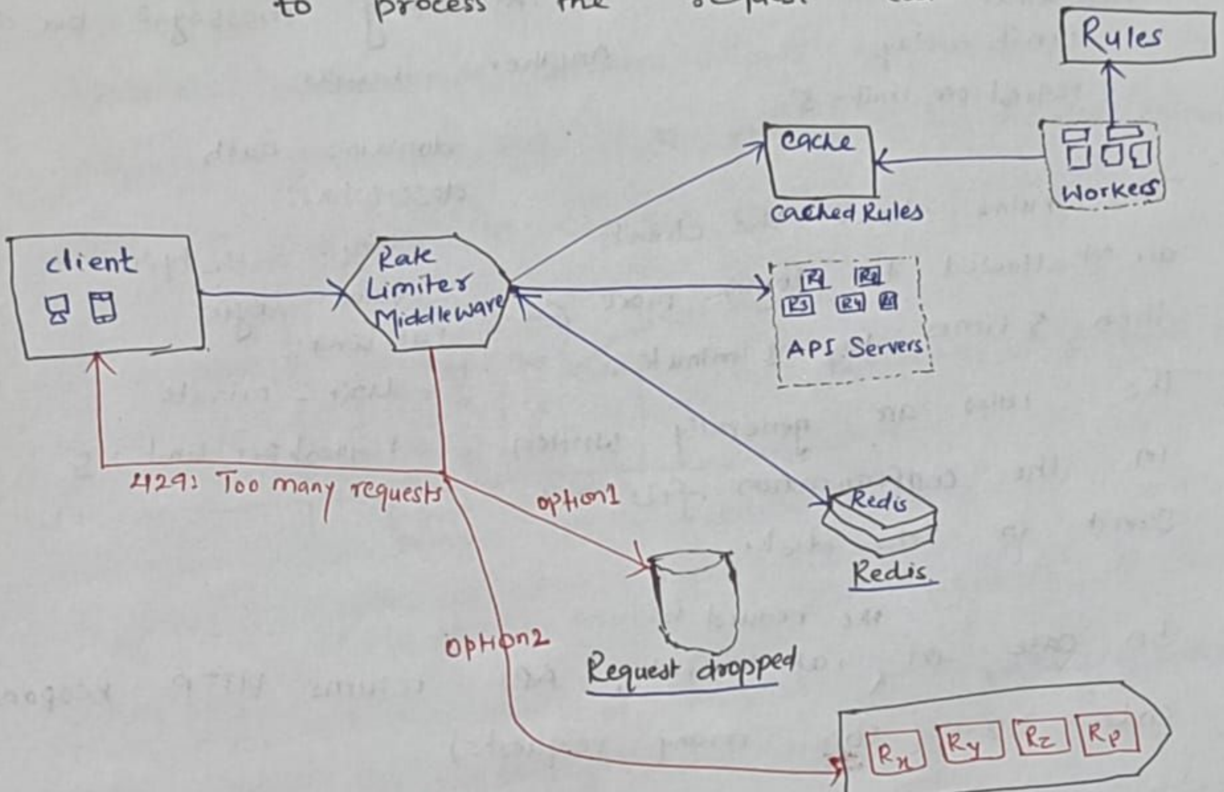
→ Workers frequently pulls the rule from the disk and store it to cache.

→ The rate limiter middleware fetches the rules from the cache and fetches the data such as timestamp, Counter etc from the redis. Then it does the computation when any request comes and decide ~~weather~~ whether the request to be processed or rate limited.

—

→ In case of rate Limiting the request - We have two options :-

- → To drop the request and send a status code 429: too many request back to the client.
- → Push the request to the message queue to process the request later.



Scalling for distributed ~~enviro~~ environment

Now, we have to scale the system for a distributed environment to support multiple servers and concurrent threads.

We have mainly two challenges.

1.) Race Condition

2.) Synchronization issue.

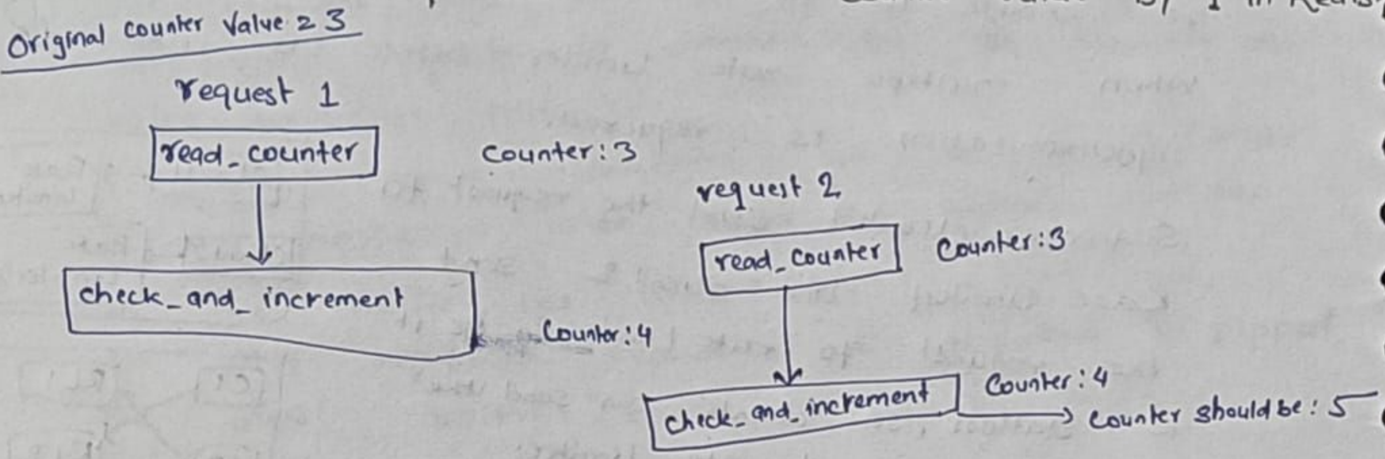
Race Condition :-

As per the diagram:-

→ Read the counter value from Redis

→ Check if (counter +1) exceeds the threshold.

→ If not, increment the counter value by 1 in Redis.



Assume, the counter value in redis = 3. If two concurrent requests come at the same time, and read the counter value before either of them writes the value back. Each will increment the counter by one, and write it back without checking the other thread. Both requests (threads) believe they have the correct counter value 4. However the correct counter value should be 5.

Locks are the solution for solving the race condition. But Lock will significantly slow down the system. That might impect the performance of the system.

Two straitegies are commonly use to solve the problem. Lua scripts and sorted sets ~~of~~ data structure in Redis. Redis Lua scripts:-

→ This may result the better performance.

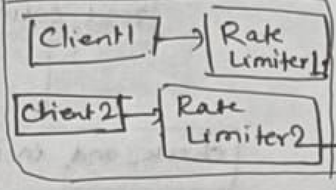
→ Also, all steps within a script are executed in an atomic way. No other redis command can run while a script is executing.

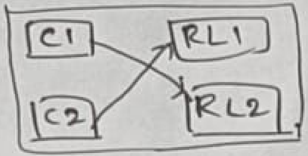
Synchronization :-

Synchronization is another important factor to Consider in distributed environment. To support millions of user 1 rate Limiter might not be enough to handle the traffic.

When multiple rate Limiter server is used synchronization is required.

Suppose, client 1 send the request to Rate Limiter1 and client 2 send the request to rate Limiter 2. As it is stateless, so clients can send the request to different rate Limiter.

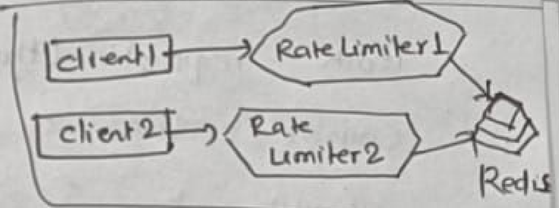




If no synchronization, then rateLimiter1 does not Contains the data about client 2 and rateLimiter - 2 doesn't contains the data about client1.

Ones possible solution is to use the sticky session that allows a client to send the traffic to the same rate Limiter, It is not scalable / flexible, so it is not advisable.

A better apporach is to use the centralize data store like redis.



Performance Optimization

Two areas to be improved.

1.) For the multi-data center setup:- the latency is so high for the users located far away from the data-center. Most cloude service provider build many edge server Locations around the world. So, traffic will ~~atom~~ automaticaly routed to the closest edge server to reduce the latency.

example - cloudflare has 194 edge Servers geographically Locations.

2.) Synchronization the data with an eventual consistency Model.

~~Mo~~ Mo~~nitoring~~
Monitoring :-

~~After the~~ After the rate limiter is put inplace, we have to gather analylitics data to check whether the rate-limiter is effective. Like, Rate Limiting algorithm / rules are effective.

For example :- If rate limiting rules are very strick, Many ~~the~~ valid requests are dropped. In this case, we can relax the rules little bit.

anothe example :- sometimes we see the rate limiter becomes ineffective, when ~~there~~ there are sudden increase in the traffic, like flash sales. In this case, we may ~~replaced~~ replace the algorithm to support the worst traffic. Token bucke is fit here.

Wrap-up

4 Different rate limiter, pros & cons. Algorithms are :- Token bucket, Leaking bucket, Fixed window, Sliding window logs, Sliding window counter.

we discussed, system architecture, rate limiter in distributed environment, performance optimization, Monitoring.

Hard - The no of request cant exceed the threshold.
Soft - Request can exceed the threshold for short time.

Rate limiting on different levels :- We talked on application level (HTTP, level 7). We can apply rate limited on IP level (Layer 3, IP) using IPTables.

~~Avoid~~ Avoid Rate Limitting :- Design your client with best practice
→ Use client cache to avoid making frequent API cals.
→ Understand the limit, and do not send many request in short time frame.
→ Catch errors & exceptions, so client can gracefully recover from exceptions.
→ Add sufficient back off time to retry the logic.