DESIGN RATE LIMITER

→ Why do we need Rate Limiter ?

In DDOS attacks the attacker sends unwanted requests to a server. And these servers have limited resources like RAM, disk space etc. So when 1 lakh requests/second, server goes down and genuine user won't be able to access it

Algorithms to design RL -

(i) Token Bucket
(ii) Leaking bucket
(iii) Fixed window counter
(iv) sliding window log
(v) sliding window counter

① TOKEN BUCKET

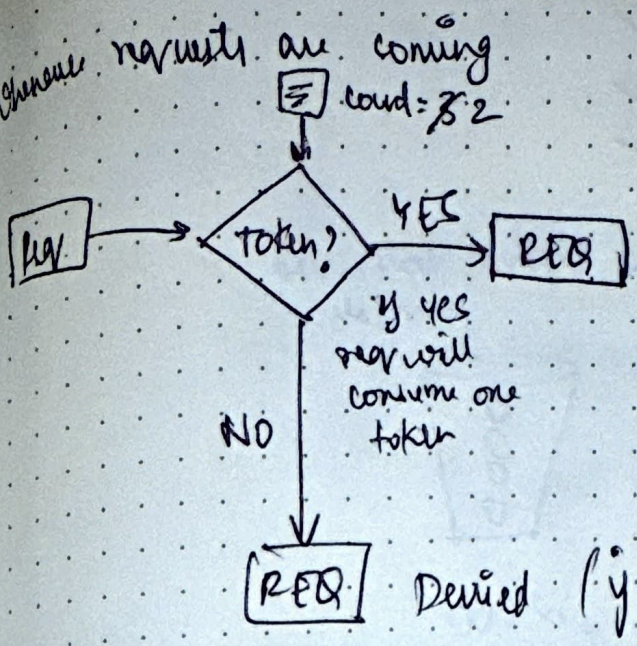
capacity of bucket to hold token

So if we try inserting more tokens than the capacity then the token will overflow.

There is also something called refiller 1 min = 2 token

↳ to keep adding after certain time

whenever requests are coming



We can configure this token bucket for each user like

per User rule → 3 counter / token / minute

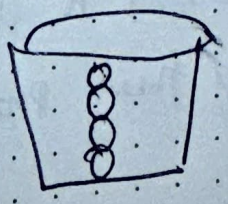
tweet POST API

→ Amazon and Stripe uses this for throttling the API requests.

⇒ Token bucket is a container that has pre-defined capacity. Token are put in preset rates periodically. Once the bucket is full, no more tokens can be added.

Say bucket size = 4, and refill rate is 4/min

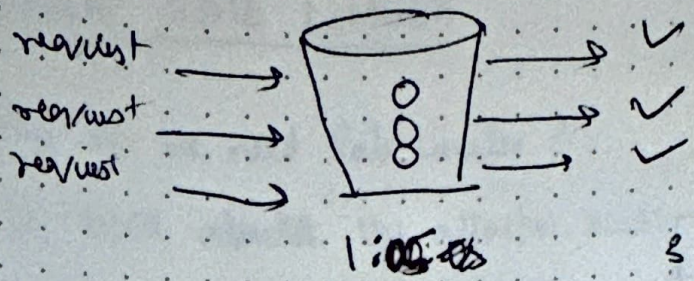
So at 1min the



1:00

request comes at 1:00

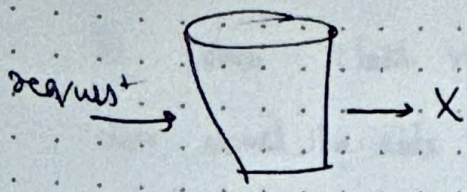
1 token will be consumed



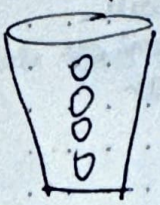
1:00

3 tokens consumed

refills



1:20



2:00

4 tokens are refilled

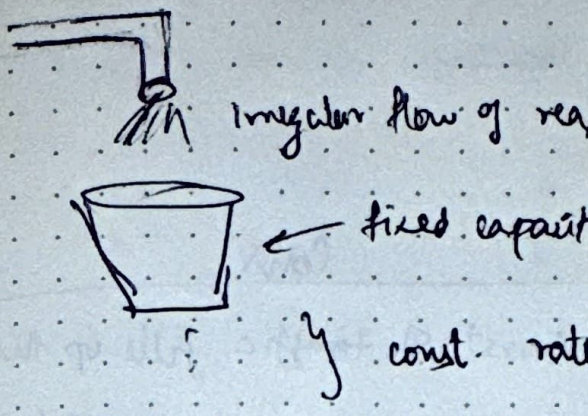
→ It is necessary to have different buckets for different end-points.

Eg :- If a user is allowed to make 1 post/sec, add 150 friends and like 5 posts/second then 3 buckets are required for each user.

→ If we need to throttle requests based on IP address, each IP address requires a bucket.

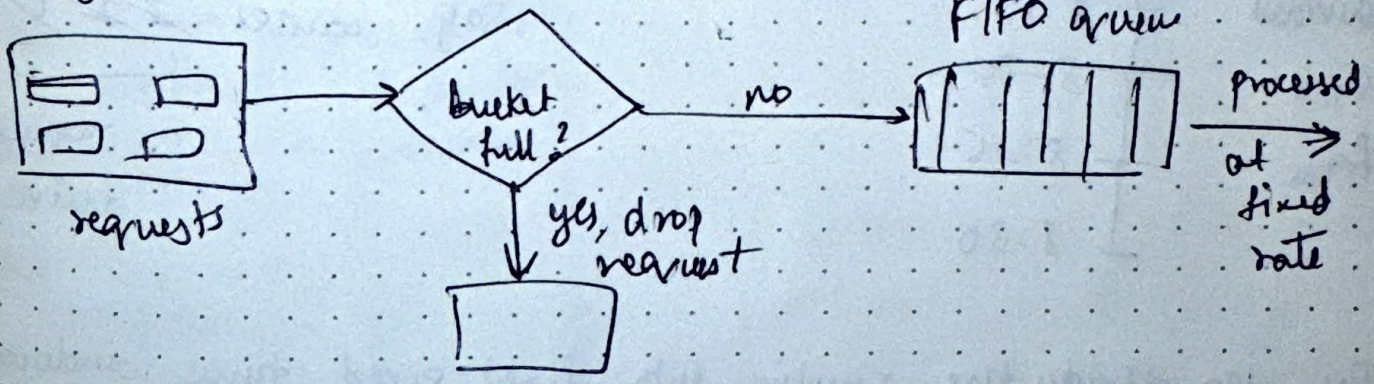
Pros	Cons
1. Easy to implement 2. Memory efficient 3. Allows a burst of traffic for short periods. request can go through as long as tokens left.	1. 2 parameters in the algorithm are bucket size and token refill rate. It is challenging to tune them properly.

Leaking BUCKET ALGORITHM



Similar to token bucket algorithm except that the requests are processed at a fixed rate. It is usually implemented. Usually implemented with a FIFO queue.

- When a request arrives, the system checks if the queue is full. If it is not full, the request is added to queue.
- Otherwise, the request is dropped.
- Request is pulled from the queue and processed through regular intervals.



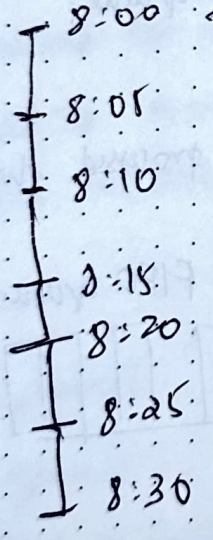
Leaking bucket takes these 2 parameters:

- Bucket Size - Equal to queue size
- outflow rate

Eg → Shopify uses this

Pros	Cons
Memory efficient requests are processed at a fixed rate	burst of traffic fills up the queue, & they are not processed, then recent requests will be rate limited 2 parameters not easy to tune properly

(3) FIXED WINDOW COUNTER



24 hr time-frame say divided into 5 min time frame

- Each window has a counter

Say counter = 3 2 1 0

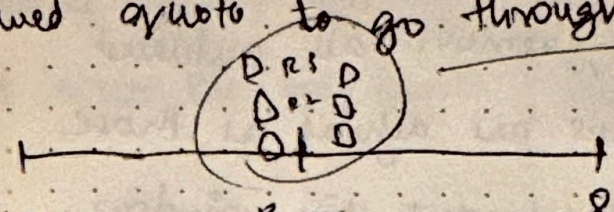
as request arrive

→ The algo divides the timeline into fixed-sized time windows and assigns a counter for each window.

→ Each request either increments or decrements it based on scenario.

→ once it reaches the threshold, new requests are dropped.

Major problem / disadvantage of this is say a burst of traffic at the edges of time window could cause more requests than allowed quota to go through → 6 requests but in 2 min more than our desired just 3 requests counter for a fixed window of 5 min.



So issue ←

That's why sliding window log was introduced.

SLIDING WINDOW LOG

say I would take 3req/min

In log, we log timestamp

size = ~~2~~ 3

REQ 1 → 10:00:10

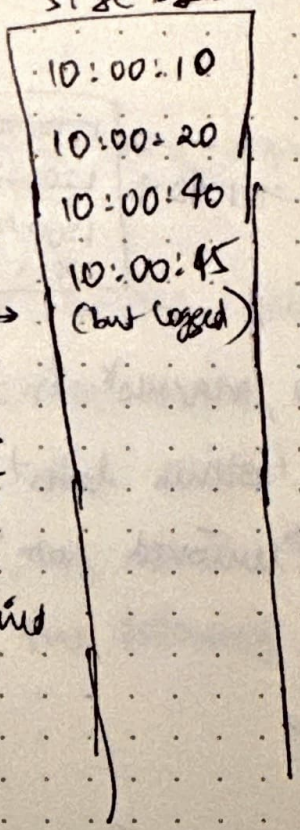
REQ 2 → 10:00:20

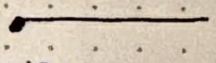
REQ 3 → 10:00:40

REQ 4 → 10:00:45

✗ denied

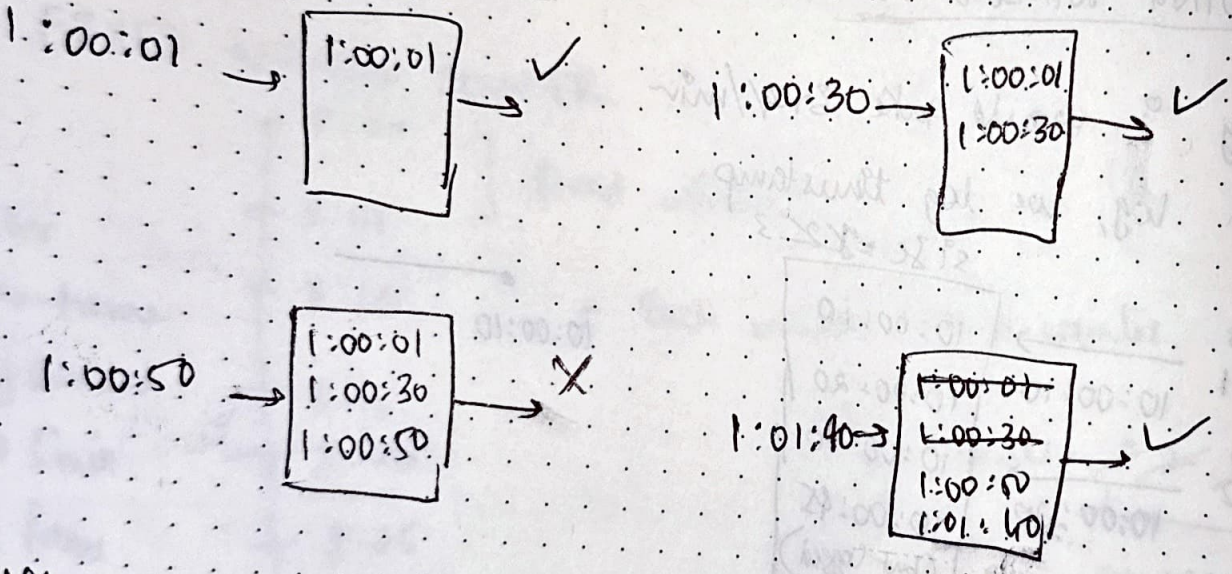
REQ 5 → 10:01:20





1. Algorithm keeps track of the timestamps. Timestamp data is usually kept in cache, such as sorted set of redis.
2. when a new request comes in, removes all outdated timestamps. Outdated timestamps are defined as those older than the start of the current time window.
3. add timestamp of the new req to the log.
4. If the log size is same or lower than the allowed count, req is accepted. Otherwise, rejected.

Allow 2 requests/min



When a new req arrives at 1:01:40, request in the range of [1:00:40 - 1:01:40] are within latest time frames. So 2 outdated time stamps are removed from the log.

~~Rate~~ Pros

① Rate limiting is very accurate. In any rolling window, requests will not exceed the rate limit.

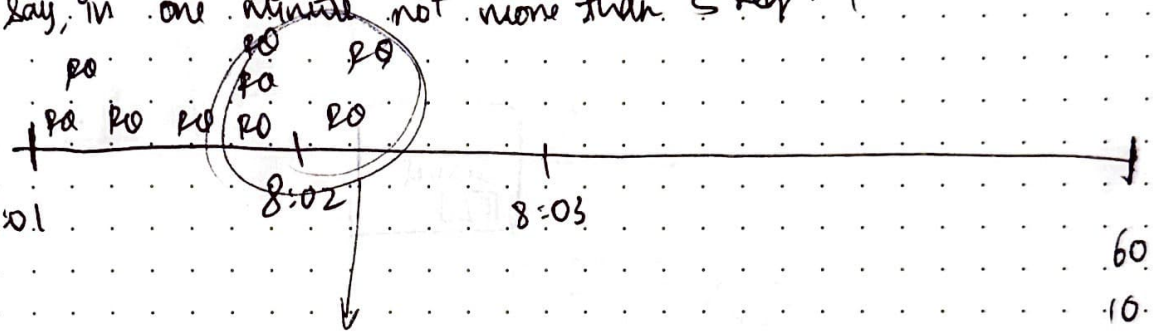
Cons

① Consumes a lot of memory because even if a request is rejected, its timestamp is stored in memory.

SLIDING WINDOW COUNTER

Fixed window counter + Sliding window log

Say, in one minute not more than 5 req. (5 req/min)



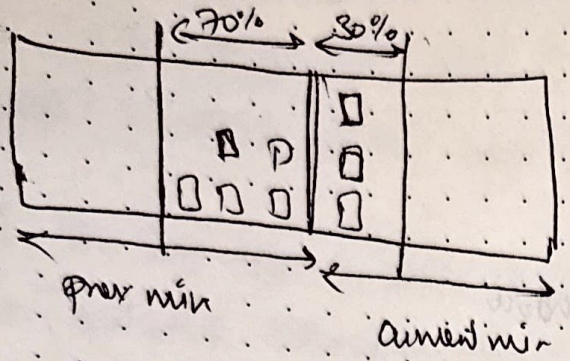
60 → 6 req
10 → ? 6

find no. of requests in this window.

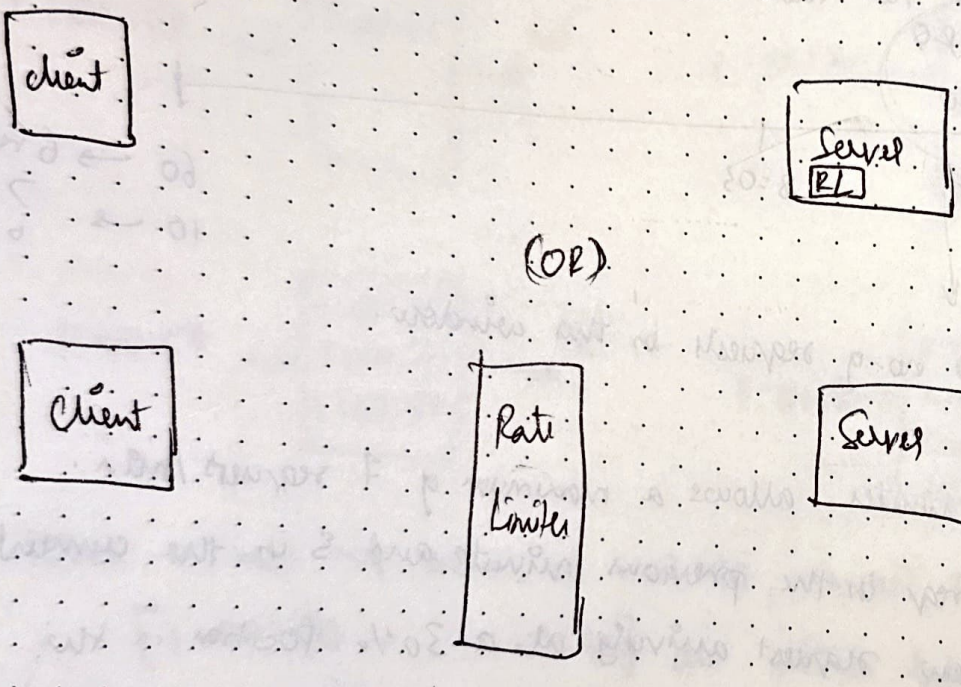
Assume the rate limiter allows a maximum of 7 requests/min. And there are 5 req in the previous minute and 3 in the current minute. For a new request arriving at a 30% position in the current minute, the number of requests in the rolling window is calculated using the following formula:-

Request in current window + (request in prev window) overlap percentage of the rolling window and previous window

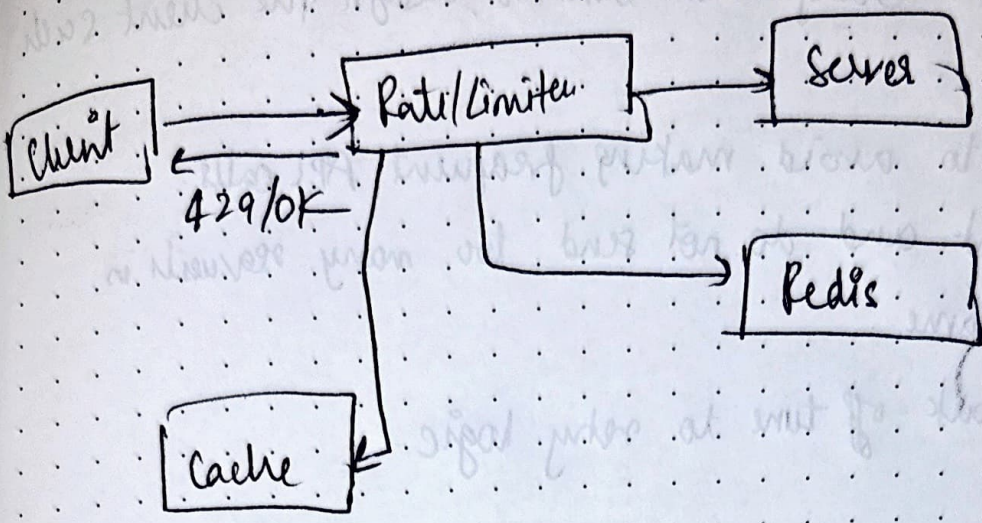
$3 + 5 * 0.7 = 6.5$ req. → (6)



High - Level architecture

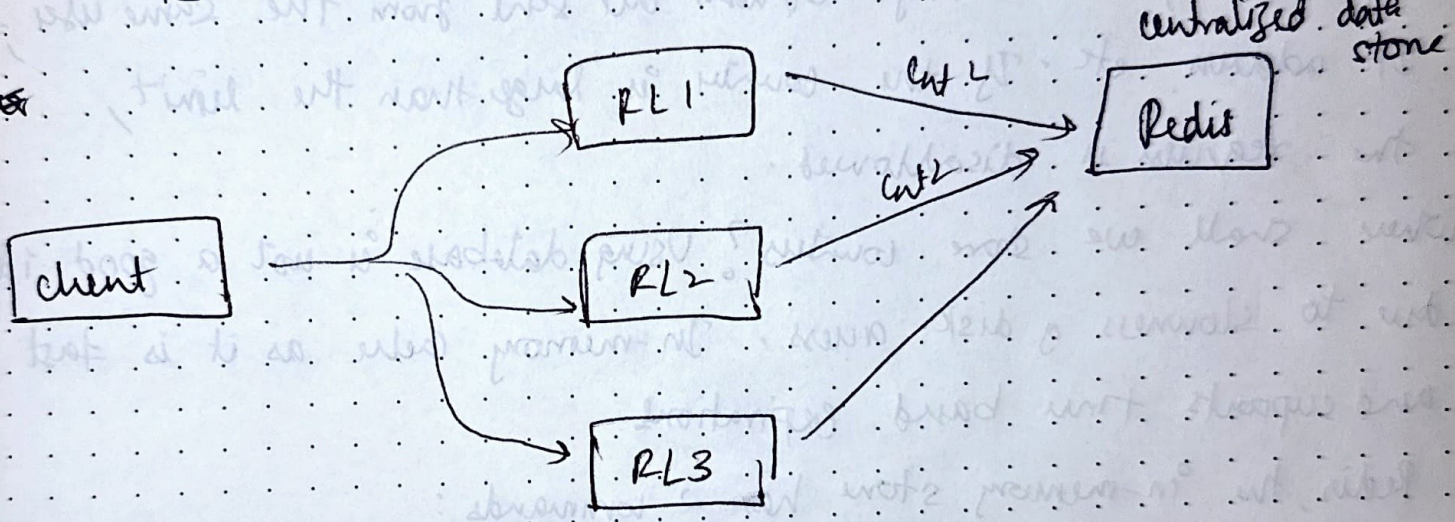


(or) put RL inside API Gateway.



for queue / timestamp sorted set

Config files present in cache



→ Redis does not maintain atomicity, what if both requests RL1 and RL2, that is updating cnt together ?

So for this issue, we can bring atomicity to redis, but with some latency.

To avoid the client from being rate-limited, design the client such that -

(i) use client cache to avoid making frequent API calls.

(ii) understand the limit and do not send too many requests in a short time frame

(iii) Add sufficient back-off time to retry logic.

Also,

The basic idea of rate limiting is that we need a counter to keep track of how many requests are sent from the same user, IP address etc. If the counter is large than the limit, the request is disallowed.

Where shall we store counters? Using database is not a good idea due to slowness of disk access. In-memory cache as it is fast and supports time based expirations.

Redis, the in-memory store has 2 commands:

INCR → increases stored counter by 1.

EXPIRE → sets timeout for the counter. If timeout expires, counter is automatically deleted