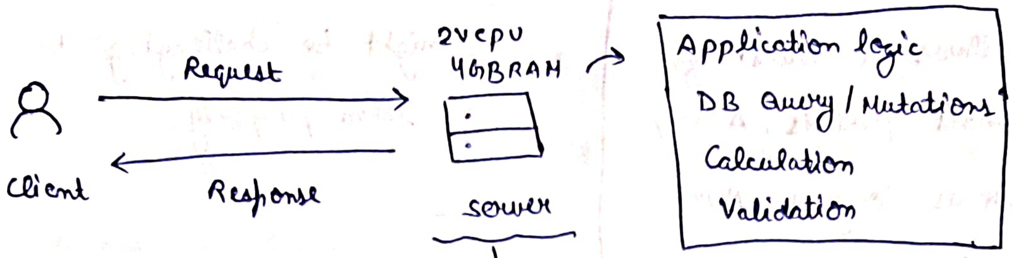
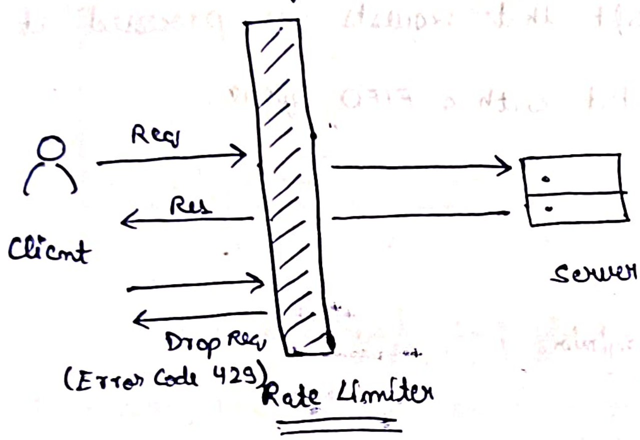
# Rate Limiting

## Request - Response Cycle




There is a limitation of this machine to process requests (let's say 100 req/min)

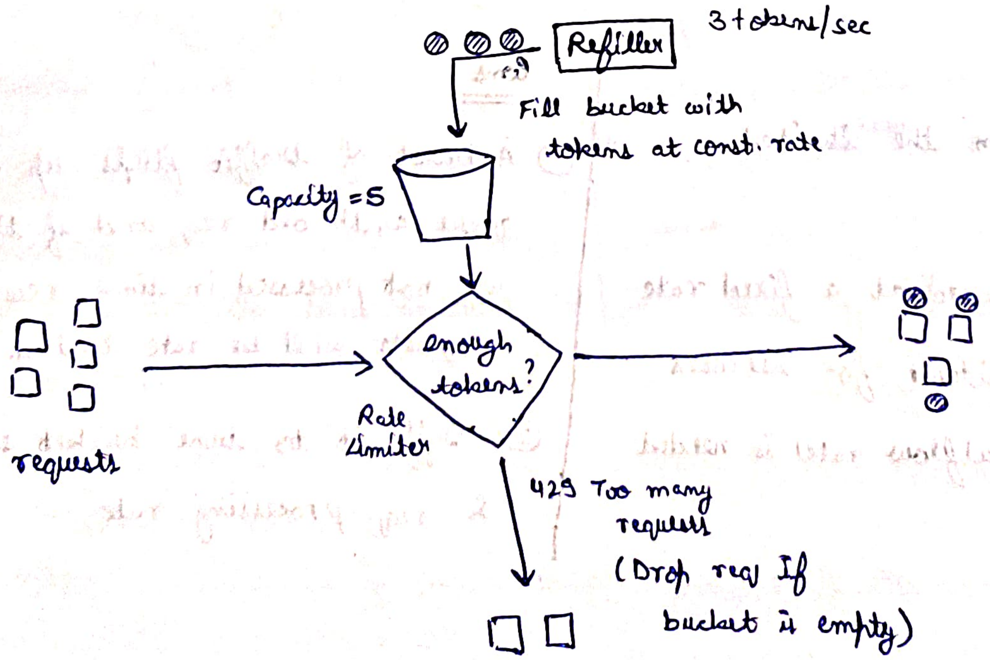
If the requests will be 150 or 200/min, then the system may crash




rate limiter is used to control the rate of traffic sent by a client or a service

## Algorithms for Rate Limiting

### ① Token Bucket Algorithm




Proceed req if tokens available in bucket & remove consumed token from bucket

**Pros:-**

① Easy to implement

② Memory efficient

③ Token bucket allows a burst of traffic for short periods. A req. can go through as long as there are tokens left.

**Cons**

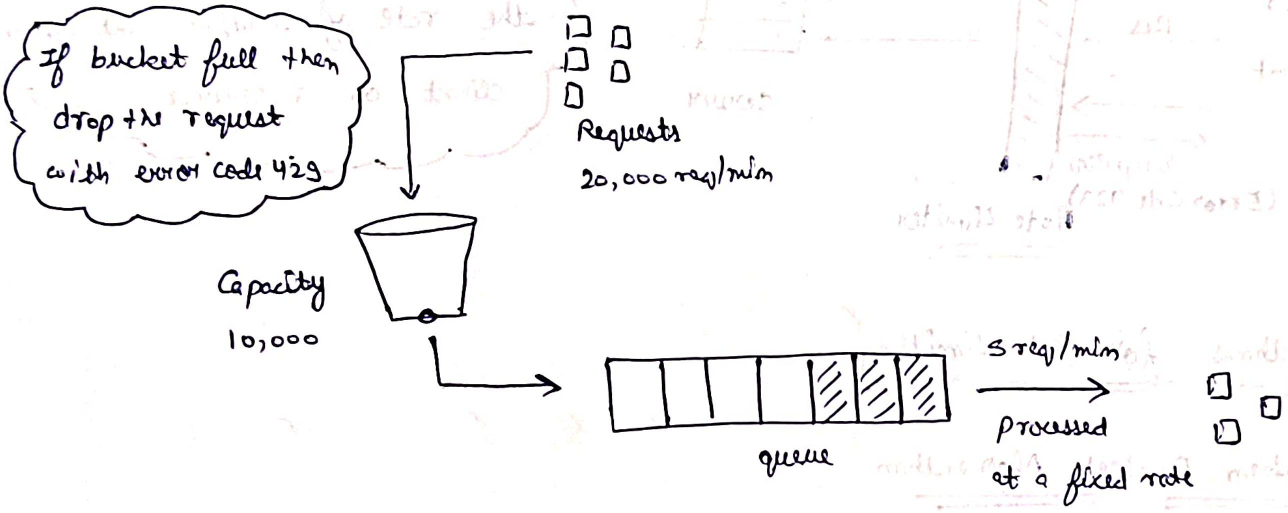
Two parameters :- Bucket size & Refill rate

↓

might be challenging to tune them properly.

## ② Leaky Bucket Algorithm

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




**Pros:-**

① Memory efficient given the limited queue size

② Requests are processed at a fixed rate therefore it is suitable for use cases that a stable outflow rate is needed

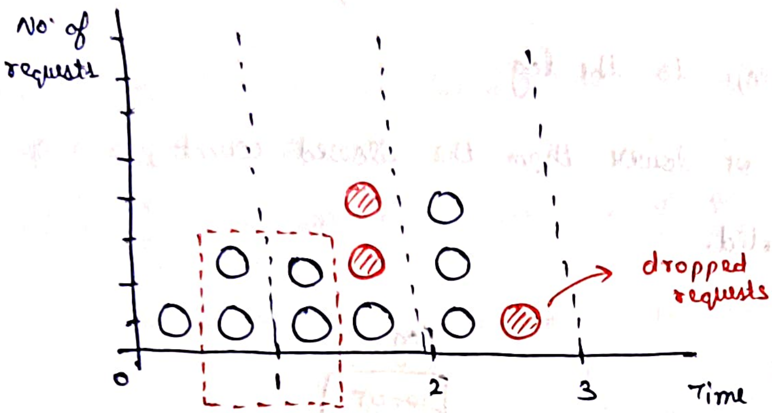
**Cons**

① A burst of traffic fills up the queue with old req. and if they are not processed in time, recent requests will be rate limited.

② Difficult to tune bucket size & req. processing rate

## ③ Fixed Window Counter Algorithm

- The algo. divides the timeline into fix-sized time windows & assign a counter for each window.
- Each req. increments the counter by one.
- Once the counter reaches the pre-defined threshold, new req. are dropped until a new time window starts.




3 req/sec

Counter for each block

↳ You are saying that for every second there should only be atmost 3 requests, but if we look at this timeframe (0.5sec - 1.5sec) → It is handling 4 requests in a second.

↳ Problem with this Algorithm

**Pros:-**

① Memory efficient

② Easy to understand

③ Resetting available quota at the end of a unit time window fits certain use cases

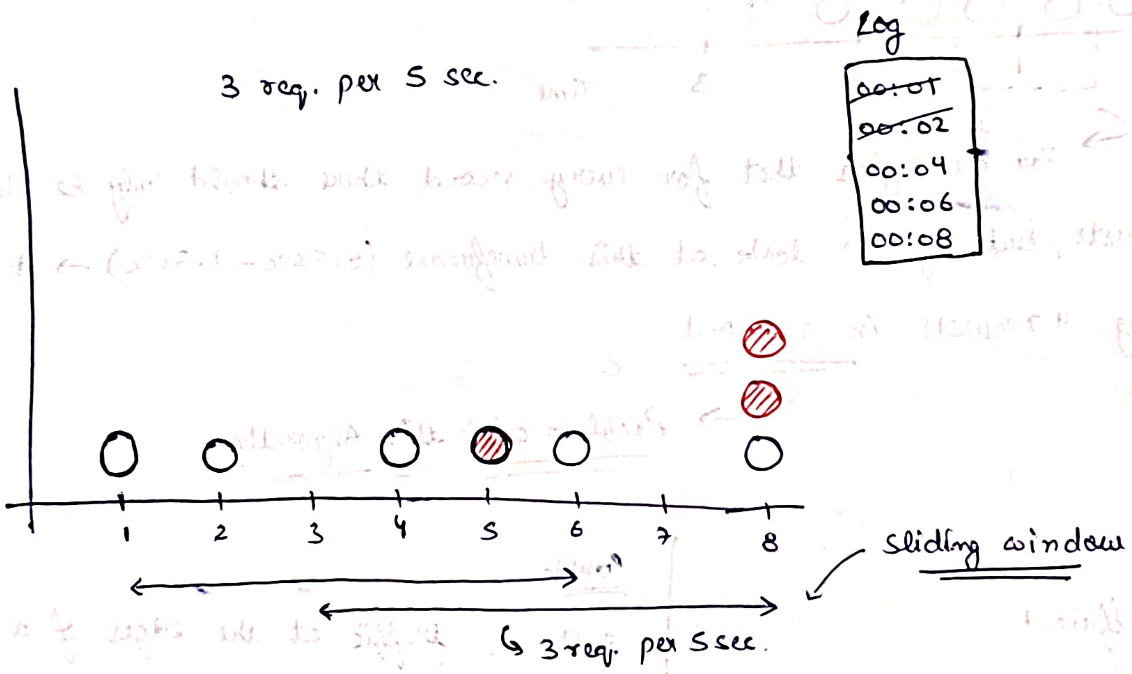
**Cons:-**

Spike in traffic at the edges of a window could cause more requests than the allowed quota to go through.

④ Sliding Window Log Algorithm

↳ fixes the issue of fixed window counter algorithm

- The algo. keeps track of req. timestamps. Timestamp data is usually kept in cache such as sorted sets of Redis.
- When a new req. comes in, remove all the outdated timestamps. Outdated timestamps are defined as those older than the start of the curr. time window
- Add timestamp of the new req. to the log.
- If the log size is the same or lower than the allowed count, a req. is accepted. Otherwise, it is rejected.




Pros:-

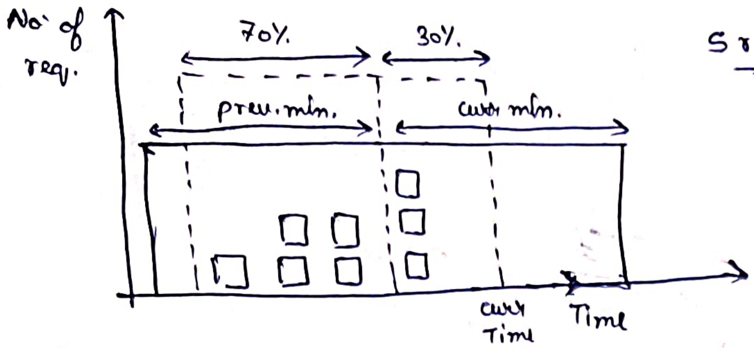
Rate limiting implemented by this algo. is very accurate. In any rolling window, req. will not exceed the rate limit.

Cons:-

The algo. consumes a lot of memory because even if a req. is rejected, it's timestamp might still be stored in memory.

## ⑤ Sliding Window Counter Algorithm

↳ hybrid approach of fixed window counter & sliding window log




5 req/min

Assume the rate limiter allows a max. of 7 req/min & there are 5 req. in the prev. minute & 3 in the curr. min. For a new req. that arrives at a 30% position in the curr. min., the no. of req. in the rolling window is :-

Req. in the curr. window + Req. in the prev. window * overlap % of the rolling window & prev. window

⇒ 3 + 5 * 0.7% = 6.5 request (so this can go through the rate limiter as the limit is 7 req/min)