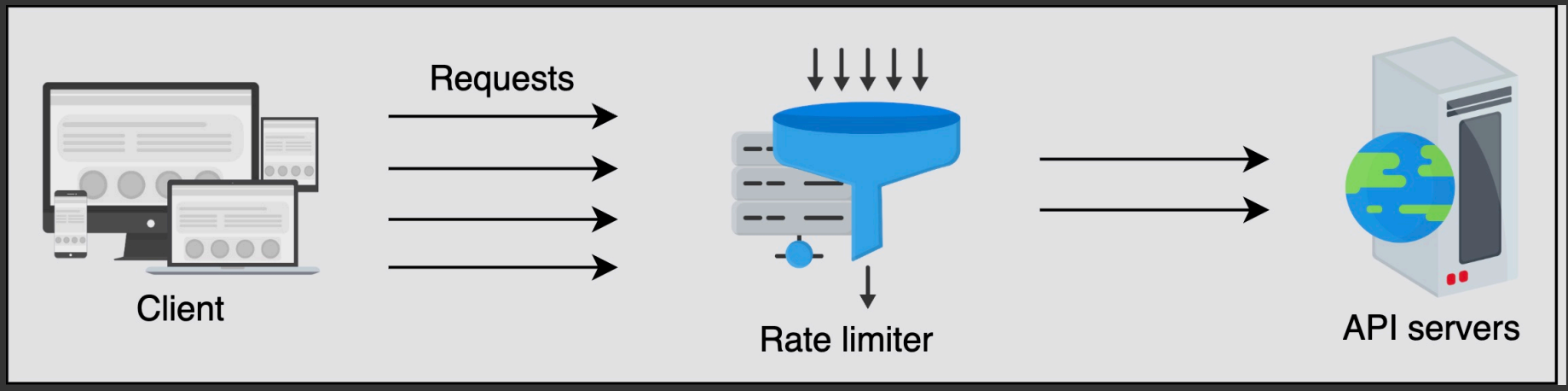
Rate limiter, → puts a limit on the number of requests a service fulfils.

Why do we need a rate limiter?

A rate limiter is generally used as a defensive layer for services to avoid their excessive usage, whether intended or unintended. It also protects services against abusive behaviors that target the application layer, such as denial-of-service (DOS) attacks and brute-force password attempts.



→ preventing resource starvation
→ managing policies & quotas
→ controlling data flow
→ Avoiding excess costs.

Requirements of a Rate Limiter's Design

Functional Requirement

```
To limit the number of requests a client can send to an API within a time window.
To make the limit of requests per window configurable.
To make sure that the client gets a message (error or notification) whenever the defined threshold is crossed within a single server or combination of servers.
```

Non-functional Requirement

```
Availability: Essentially, the rate limiter protects our system. Therefore, it should be highly available.
Low latency: Because all API requests pass through the rate limiter, it should work with a minimum latency without affecting the user experience.
Scalability: Our design should be highly scalable. It should be able to rate limit an increasing number of clients' requests over time.
```

Types of throttling

A rate limiter can perform three types of throttling.

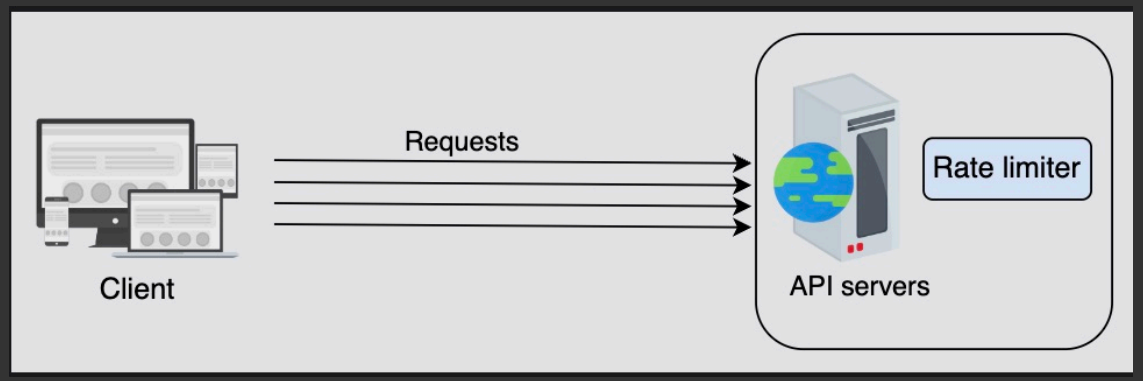
1. **Hard throttling:** This type of throttling puts a hard limit on the number of API requests. So, whenever a request exceeds the limit, it is discarded.
2. **Soft throttling:** Under soft throttling, the number of requests can exceed the predefined limit by a certain percentage. For example, if our system has a predefined limit of 500 messages per minute with a 5% exceed in the limit, we can let the client send 525 requests per minute.
3. **Elastic or dynamic throttling:** In this throttling, the number of requests can cross the predefined limit if the system has excess resources available. However, there is no specific percentage defined for the upper limit. For example, if our system allows 500 requests per minute, it can let the user send more than 500 requests when free resources are available.

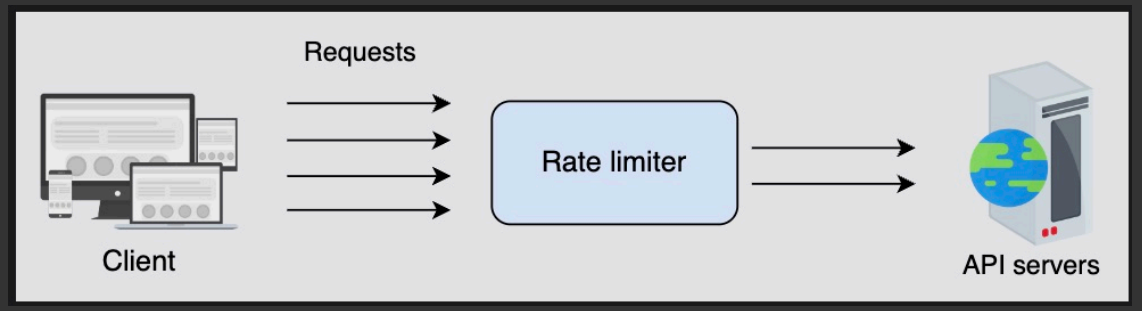
where to place the rate limiter

→ On the client side →

→ On the server side

→ As middleware





1. **A rate limiter with a centralized database:** In this approach, rate limiters interact with a centralized database, preferably Redis or Cassandra. The advantage of this model is that the counters are stored in centralized databases. Therefore, a client can't exceed the predefined limit. However, there are a few drawbacks to this approach. It causes an increase in latency if an enormous number of requests hit the centralized database. Another extensive problem is the potential for race conditions in highly concurrer requests (or associated lock contention).
2. **A rate limiter with a distributed database:** Using an independent cluster of nodes is another approach where the rate-limiting state is in a distributed database. In this approach, each node has to track the rat limit. The problem with this approach is that a client could exceed a rate limit—at least momentarily, while the state is being collected from everyone—when sending requests to different nodes (rate-limiters). To enforce the limit, we must set up sticky sessions in the load balancer to send each consume to exactly one node. However, this approach lacks fault tolerance and poses scaling problems when the nodes get overloaded.

Can a load balancer be used as a rate limiter?

Hide Answer
Load balancers play a critical role in preventing an application server from being overwhelmed by an excessive number of requests. They achieve this by either declining requests based on predefined limits or directing them to a queue for deferred processing. It's essential to emphasize that load balancers treat all incoming requests impartially, without recognizing the diverse complexities and processing durations associated with different operations. For instance, certain operations within a web service may be rapid, while others may be more time-intensive. When there is a need to control the number of requests for specific operations, a more effective approach is to implement this control at the application server level using a rate limiter. The rate limiter excels at understanding the intricacies of individual operations and can selectively impose limitations as required.

Assume a scenario in which a client intends to send requests for a particular service using two virtual machines (VMs), where one is using a VPN to a different region. Suppose that the throttling identifier works based on user credentials. Therefore, the user ID would be the same for both sessions. Moreover, let's assume that requests from different VMs can hit different data centers. How would the throttling work to prevent the user from exceeding the rate limit in this scenario?

Hide Answer
To rate limit the incoming requests, we have two different choices to place the rate limiter.

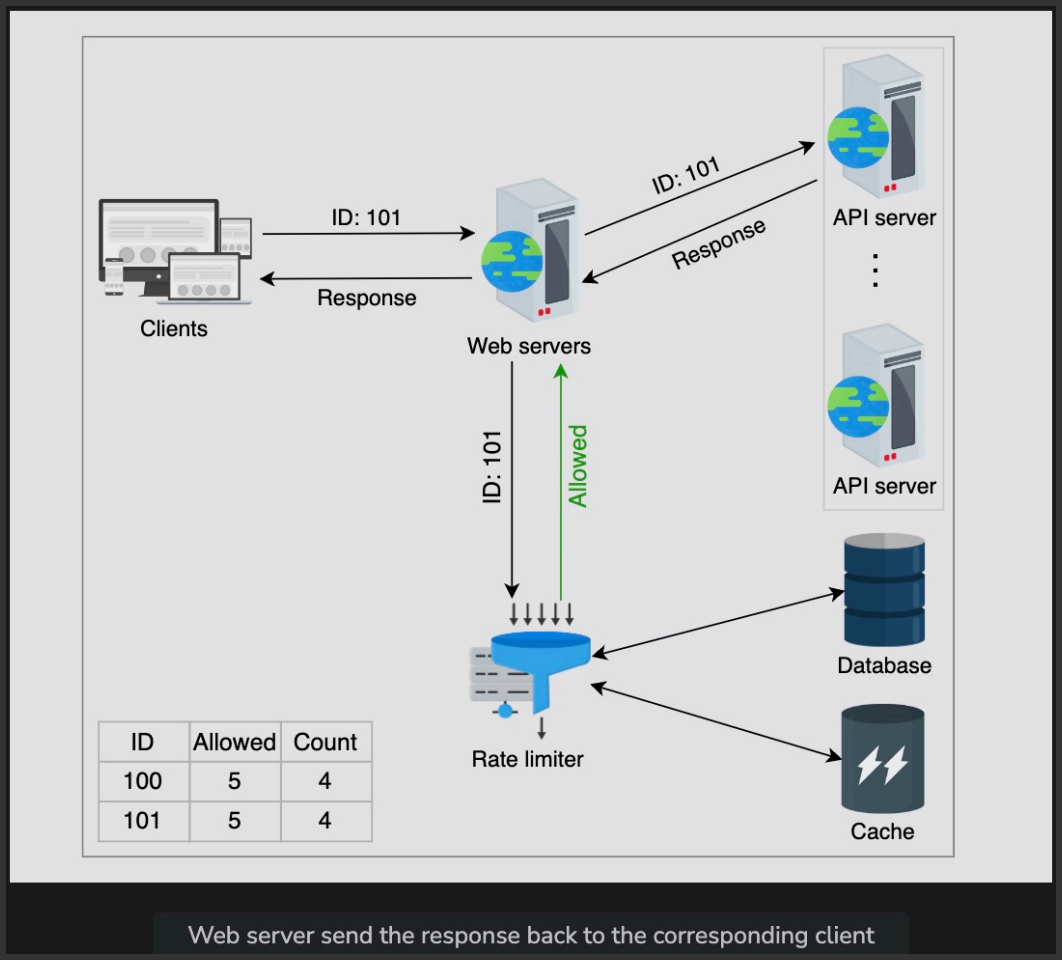
Rate limiter per data center: One way to throttle the incoming requests from the user is to use rate limiting per data centers. Each datac enter will have its own rate-limiter, which limits the incoming requests. In this approach, the rate (count or rate limit) is relatively lower. Therefore, a limited number of requests are allowed per unit time. Moreover, this approach provides lower latency since the requests are normally directed to the nearest data centers located geographically. Often, latency within a data center is less than one millisecond and multiple redundant paths are available in case of some link failure.

A shared rate limiter across data centers: Another approach is to use a shared rate limiter across multiple data centers. This way, requests received from both VMs will be throttled by the single rate limiter. The number of requests allowed in this case is higher. However, this approach is relatively slower, as prior to directing a request to any nearest data center, it will pass through the shared rate limiter. Latency is often high and variable across geographically distributed data centers and not a lot of redundant paths are available.

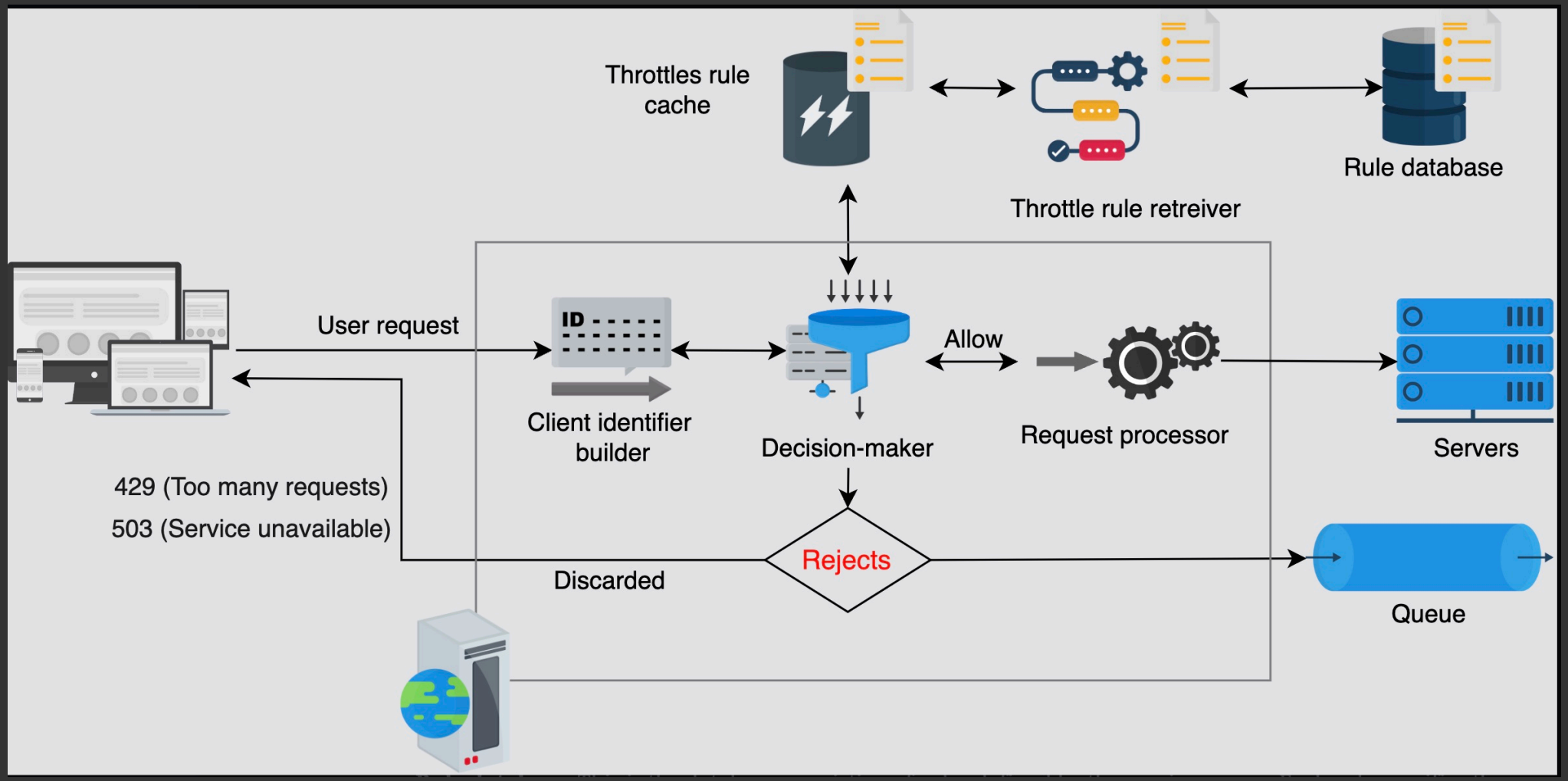
Design of Rate limiter

High level design

In the above rate-limiting rule, the unit is set to day and the request_per_unit is set to 5. These parameters define that the system can allow five marketing messages per day.



Detailed design



Rule database: This is the database, consisting of rules defined by the service owner. Each rule specifies the number of requests allowed for a particular client per unit of time.

Rules retriever: This is a background process that periodically checks for any modifications to the rules in the database. The rule cache is updated if there are any modifications made to the existing rules.

Throttle rules cache: The cache consists of rules retrieved from the rule database. The cache serves a rate-limiter request faster than persistent storage. As a result, it increases the performance of the system. So, when the rate limiter receives a request against an ID (key), it checks the ID against the rules in the cache.

Decision-maker: This component is responsible for making decisions against the rules in the cache. This component works based on one of the rate-limiting algorithms that are discussed in the next lesson.

Client identifier builder: This component generates a unique ID for a request received from a client. This could be a remote IP address, login ID, or a combination of several other attributes, due to which a sequencer can't be used here. This ID is considered as a key to store the user data in the key-value database. So, this key is passed to the decision-maker for further service decisions.

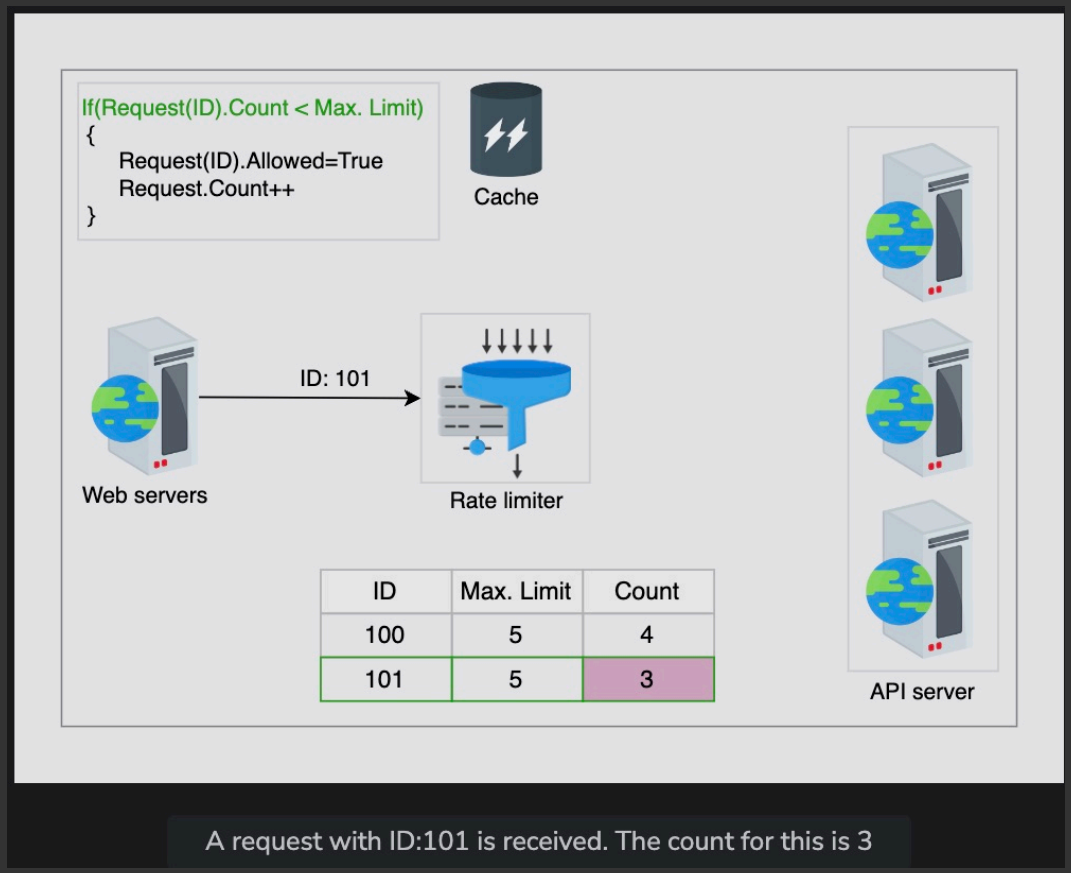
When a request is received, the client identifier builder identifies the request and forwards it to the decision-maker. The decision-maker determines the services required by request, then checks the cache against the number of requests allowed, as well as the rules provided by the service owner. If the request does not exceed the count limit, it is forwarded to the request processor, which is responsible for serving the request.

The decision-maker takes decisions based on the throttling algorithms. The throttling can be hard, soft, or elastic. Based on soft or elastic throttling, requests are allowed more than the defined limit. These requests are either served or kept in the queue and served later, upon the availability of resources. Similarly, if hard throttling is used, requests are rejected, and a response error is sent back to the client.

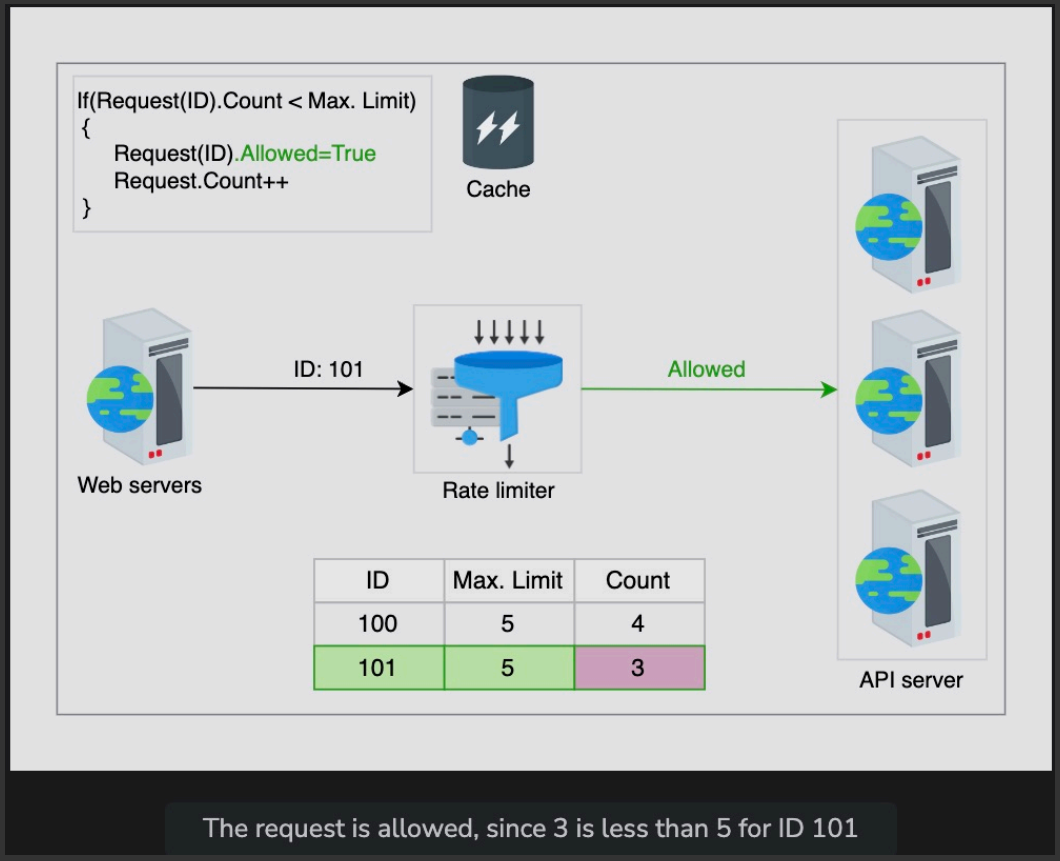
Race condition

There is a possibility of a race condition in a situation of high concurrency request patterns. It happens when the "get-then-set" approach is followed, wherein the current counter is retrieved, incremented, and then pushed back to the database. While following this approach, some additional requests can come through that could leave the incremented counter invalid. This allows a client to send a very high rate of requests, bypassing the rate-limiting controls. To avoid this problem, the locking mechanism can be used, where one process can update the counter at a time while others wait for the lock to be released. Since this approach can cause a potential bottleneck, it significantly degrades performance and does not scale well.

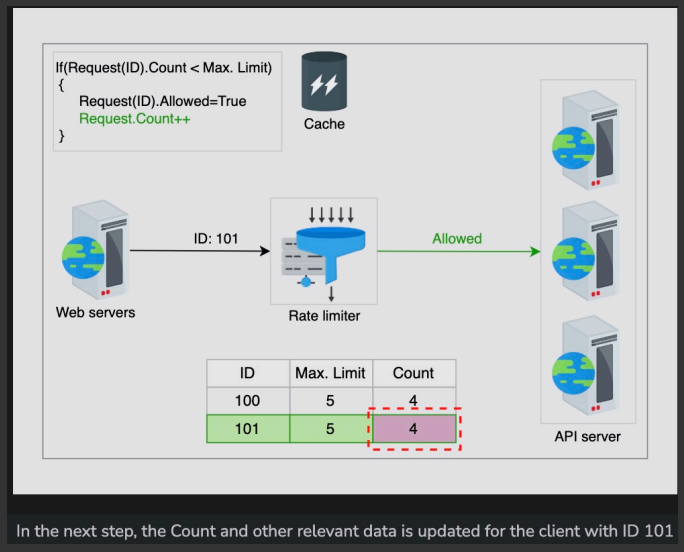
Another method that could be used is the "set-then-get" approach, wherein a value is incremented in a very performant fashion, avoiding the locking approach. This approach works if there's minimum contention. However, one might use other approaches where the allowed quota is divided into multiple places and divide the load on them, or use sharded counters to scale an approach.



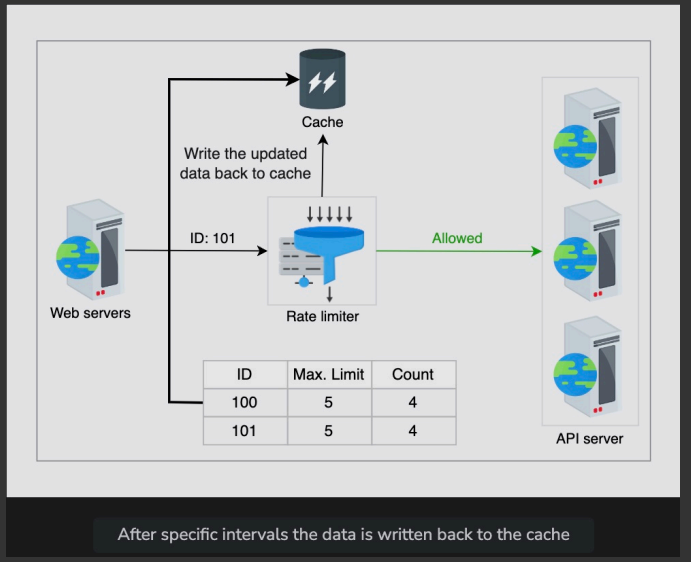
A request with ID:101 is received. The count for this is 3



The request is allowed, since 3 is less than 5 for ID 101

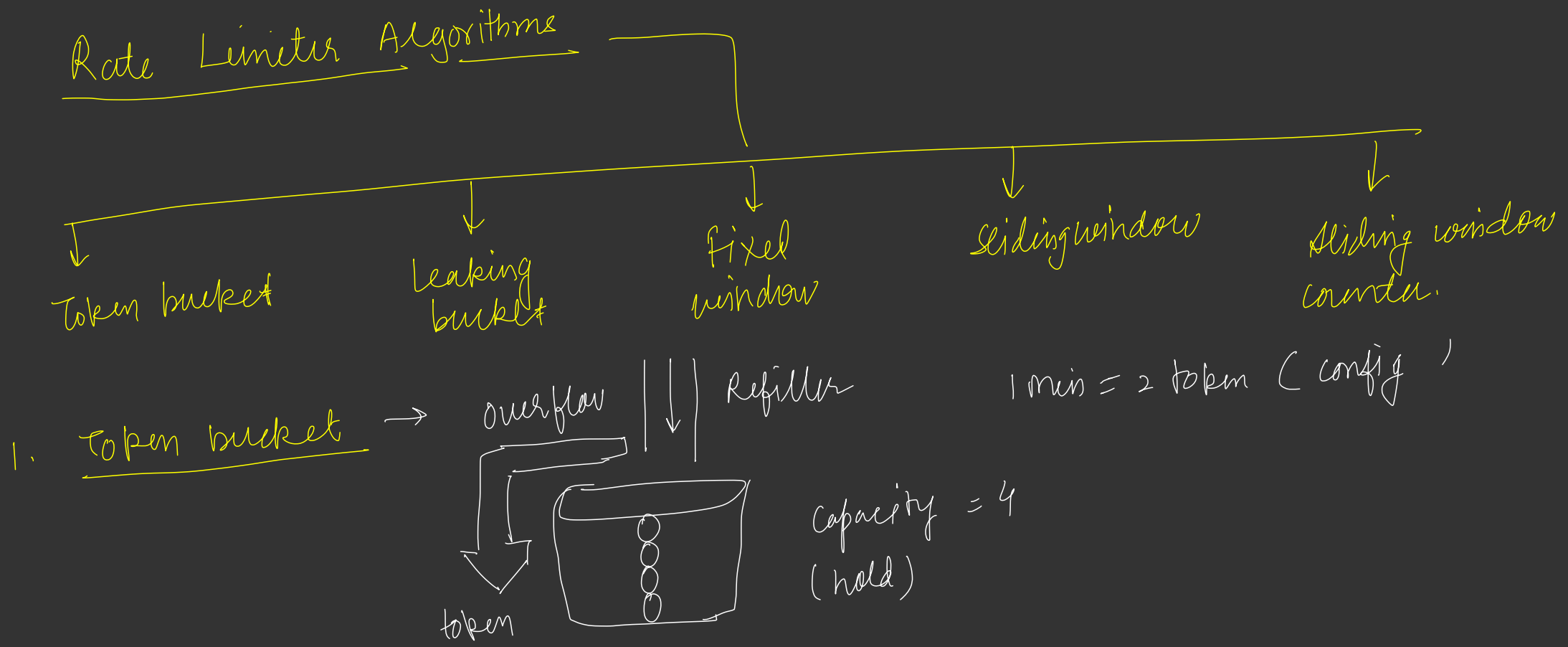


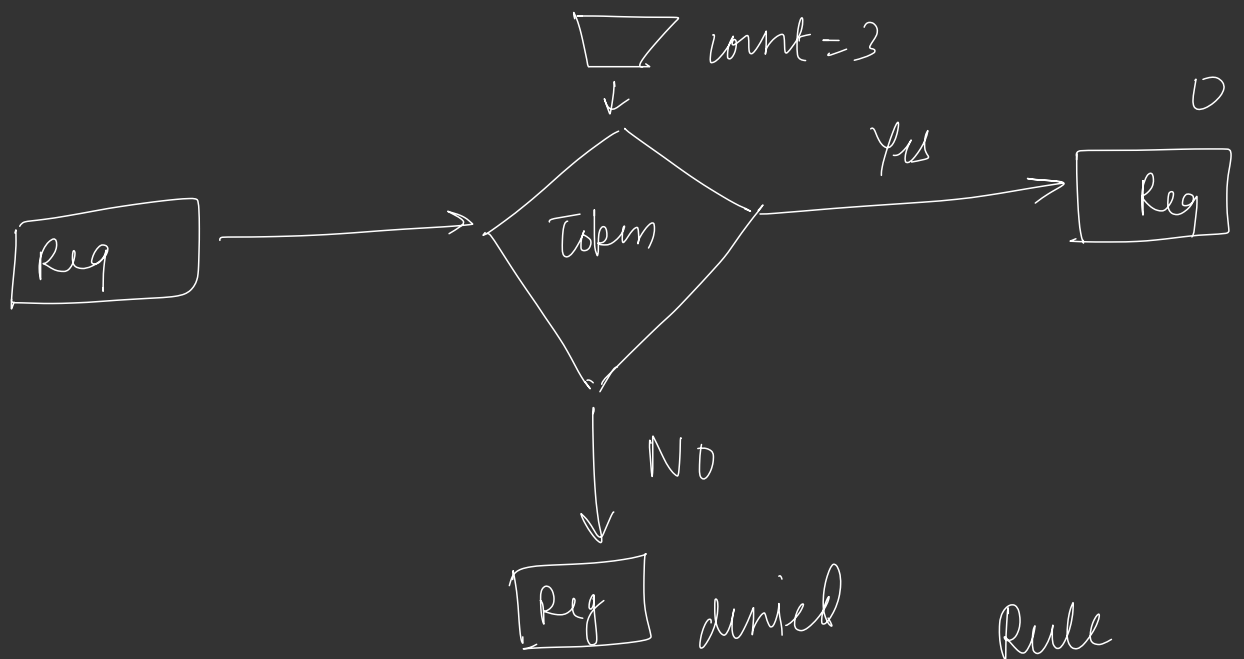
In the next step, the Count and other relevant data is updated for the client with ID 101



After specific intervals the data is written back to the cache

- **Availability:** If a rate limiter fails, multiple rate limiters will be available to handle the incoming requests. So, a single point of failure is eliminated.
- **Low latency:** Our system retrieves and updates the data of each incoming request from the cache instead of the database. First, the incoming requests are forwarded if they do not exceed the rate limit, and then the cache and database are updated.
- **Scalability:** The number of rate limiters can be increased or decreased based on the number of incoming requests within the defined limit.





Rule

[each user: 3 counter / token / minute
Post API]

/user

Post func → UI: { counter: 2, time: 10:01:00 }

Post func → UI: { counter: 1 : time: 10:01:25 }

Post func → UI : { counter: 0 · time: 10:01:35 }

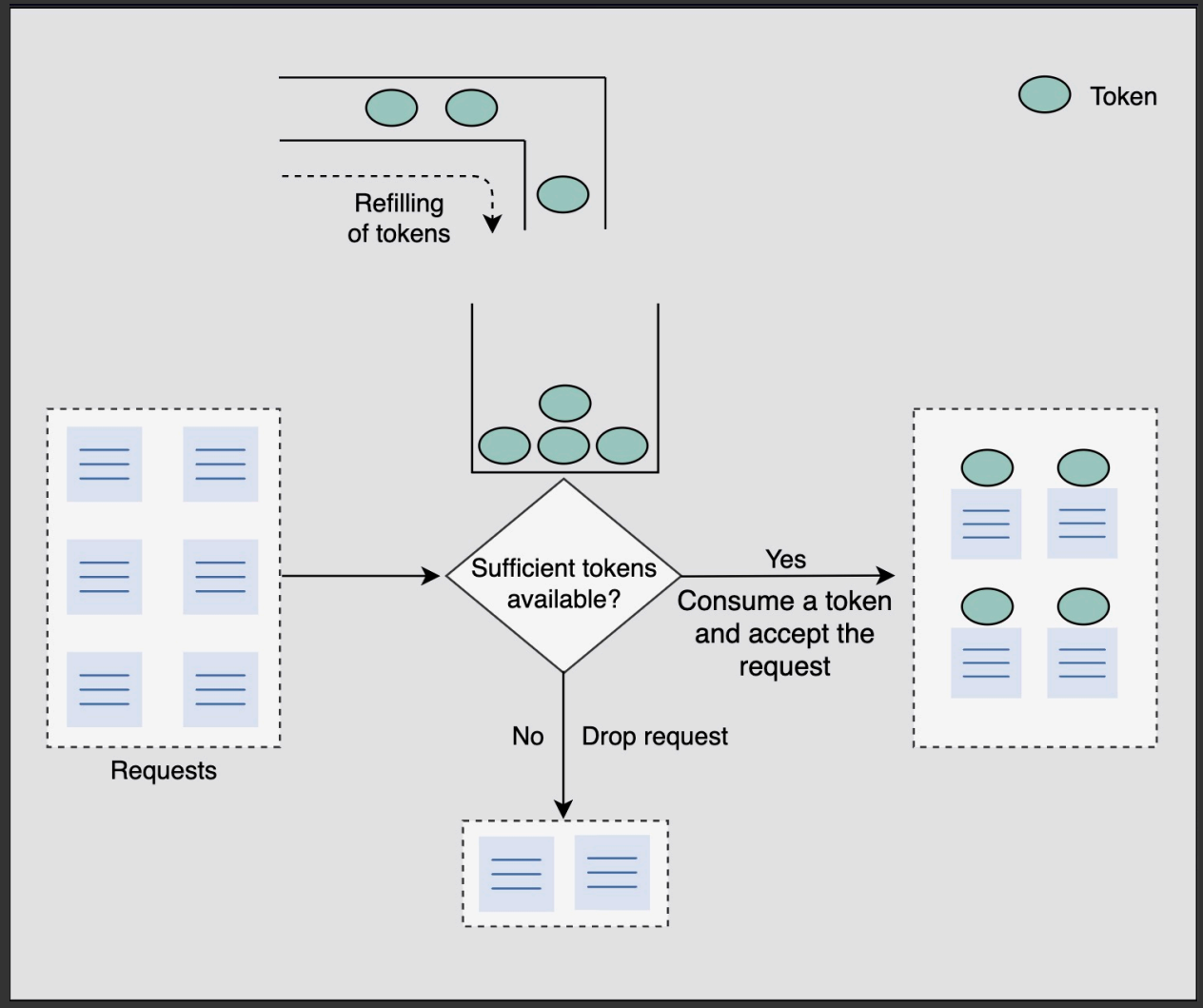
Post func UI : {

X HTTP → 429
access request in time limit

Refiller
{ 1 min : 2 token }

Assume that we have a predefined rate limit of R and the total capacity of the bucket is C.

1. The algorithm adds a new token to the bucket after every $\frac{1}{R}$ seconds.
2. The algorithm discards the new incoming tokens when the number of tokens in the bucket is equal to the total capacity C of the bucket.
3. If there are N incoming requests and the bucket has at least N tokens, the tokens are consumed, and requests are forwarded for further processing.
4. If there are N incoming requests and the bucket has a lower number of tokens, then the number of requests accepted equals the number of available tokens in the bucket.



Advantages

- This algorithm can cause a burst of traffic as long as there are enough tokens in the bucket.
- It is space efficient. The memory needed for the algorithm is nominal due to limited states.

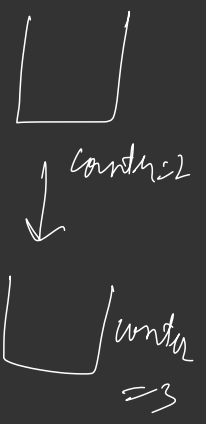
Disadvantages

- Choosing an optimal value for the essential parameters is a difficult task.

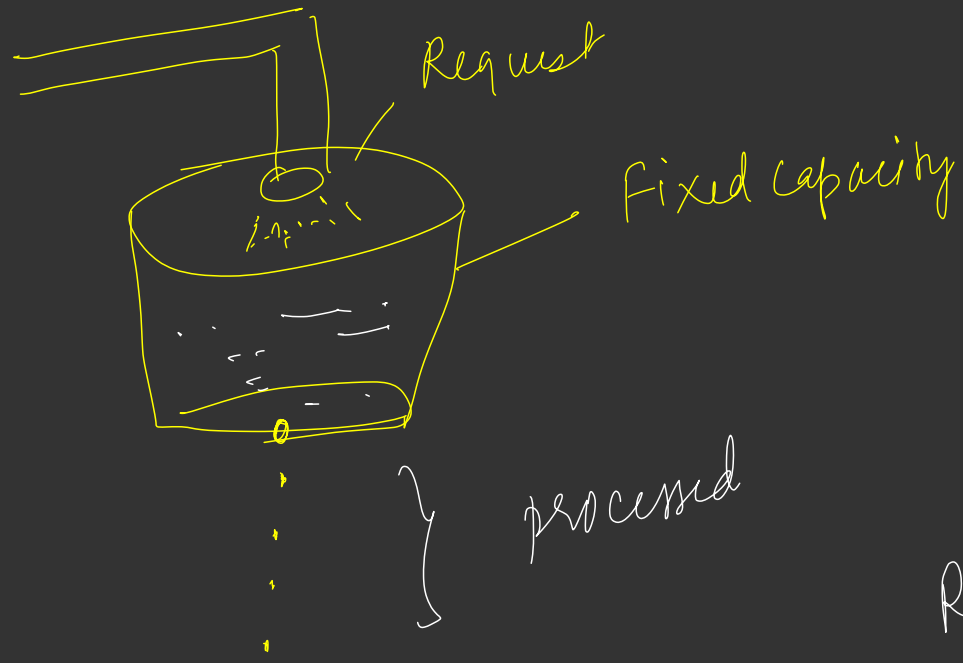
Post → v_1 { counter : 2 , 10:02 }

10:03

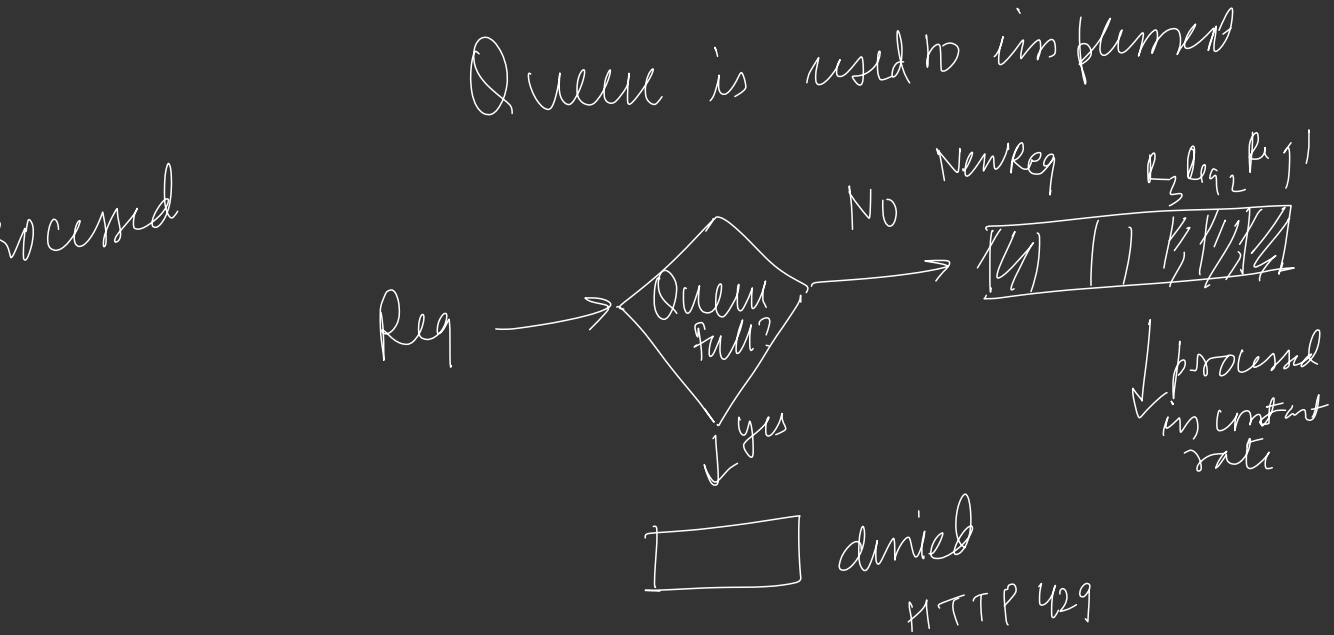
10:01 → 10:02 → 10:03

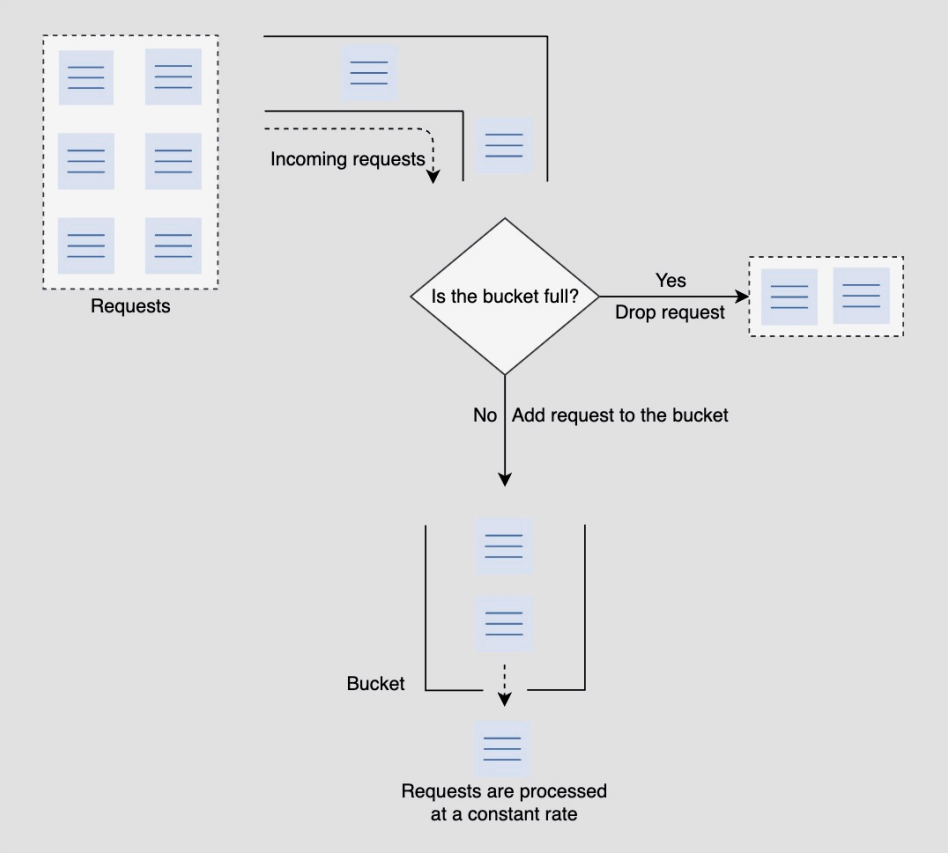


2. Leaking bucket



Queue is used to implement





How the leaking bucket algorithm works

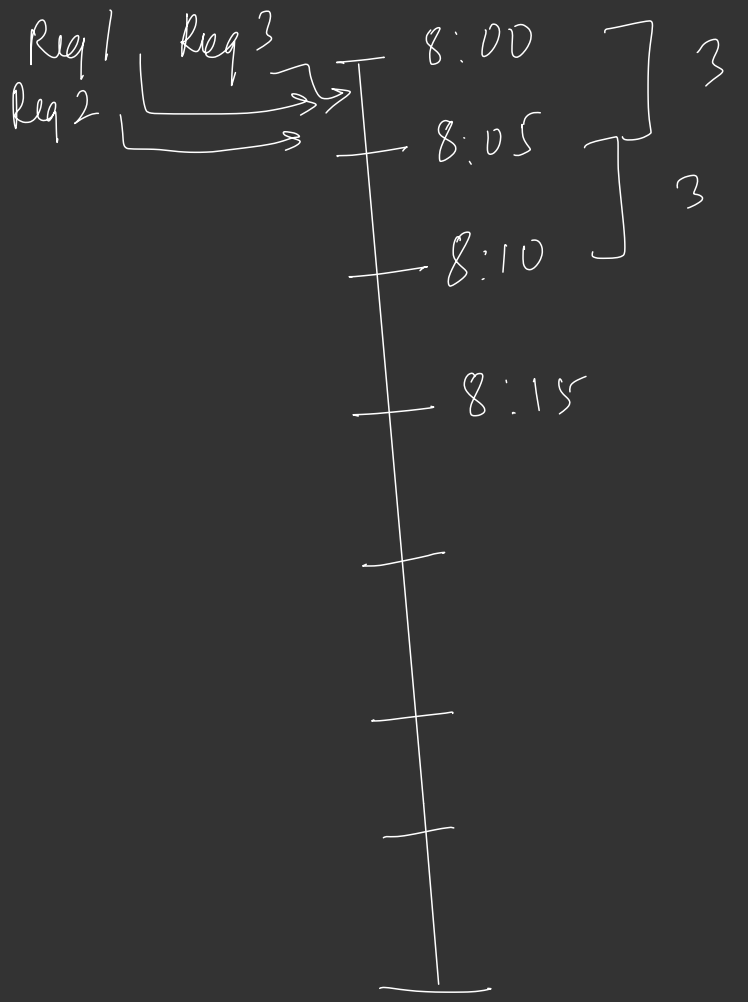
Advantages

- Due to a constant outflow rate (R_{out}), it avoids the burst of requests, unlike the token bucket algorithm.
- This algorithm is also space efficient since it requires just three states: inflow rate (R_{in}), outflow rate (R_{out}), and bucket capacity (C).
- Since requests are processed at a fixed rate, it is suitable for applications with a stable outflow rate.

Disadvantages

- A burst of requests can fill the bucket, and if not processed in the specified time, recent requests can take a hit.
- Determining an optimal bucket size and outflow rate is a challenge.

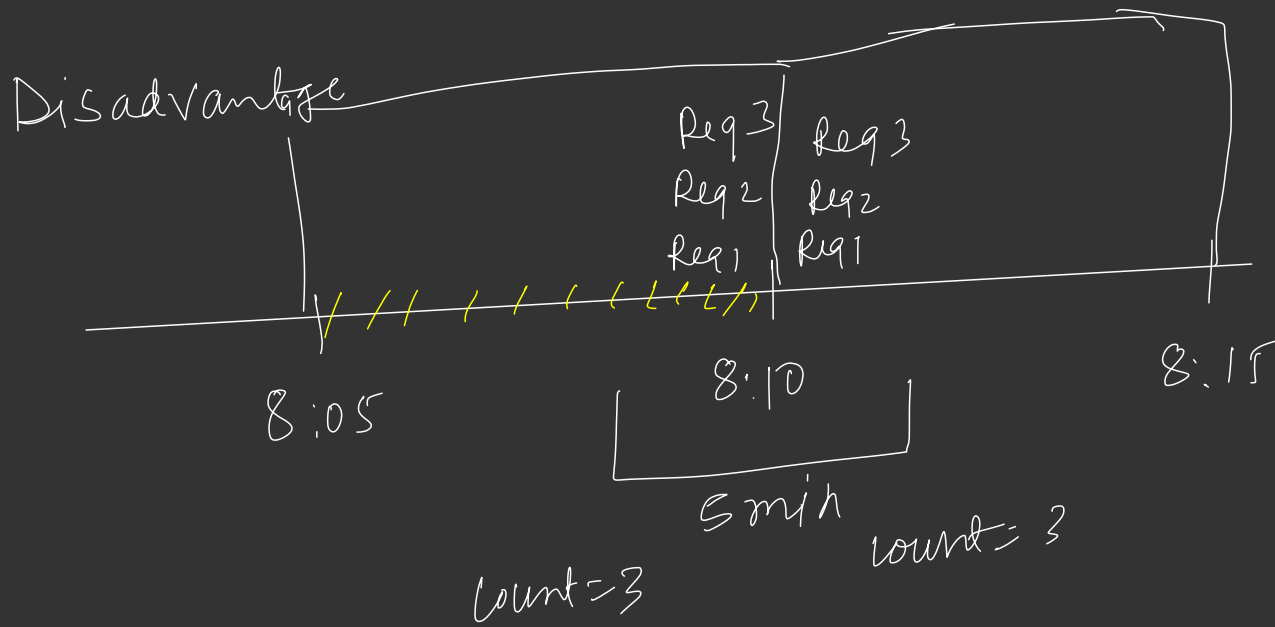
3. Fixed Window Counter



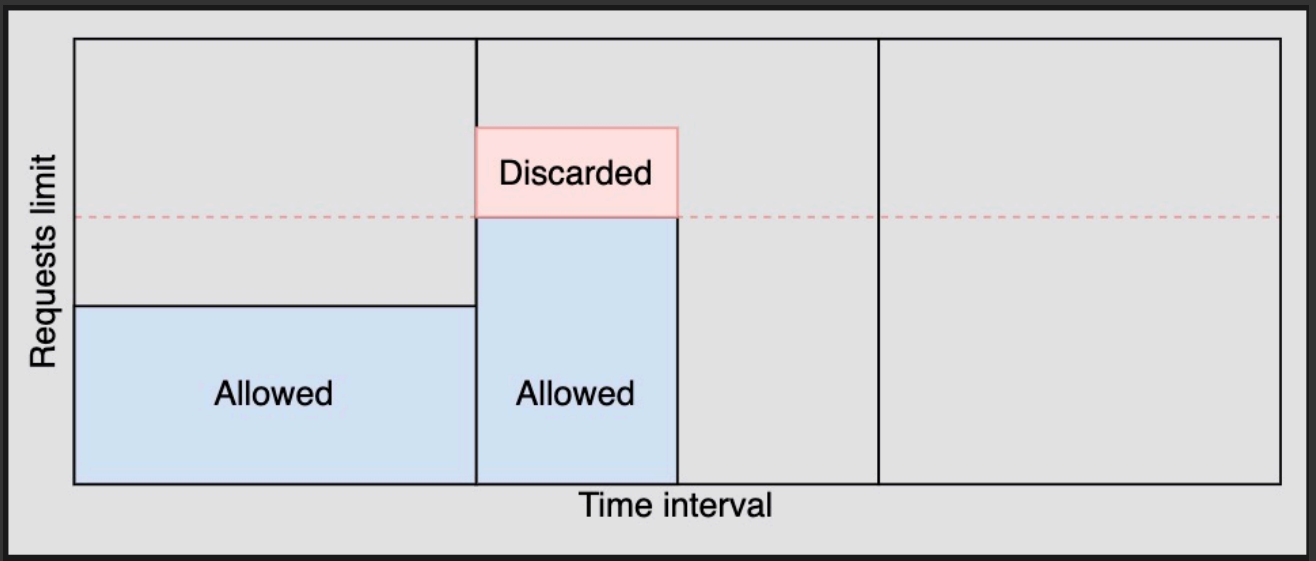
fixed window = 5 min

counter = 3

counter 3 2 1 0



→ Near to the end of window we can have double number of request.



Fixed window counter algorithm: Discard the request exceeding the limit

Essential parameters

The fixed window counter algorithm requires the following parameters:

- **Window size** (W)**:** It represents the size of the time window. It can be a minute, an hour, or any other suitable time slice.
- **Rate limit** (R)**:** It shows the number of requests allowed per time window.
- **Requests count** (N)**:** This parameter shows the number of incoming requests per window. The incoming requests are allowed if N is less than or equal to R.

Advantages

- It is also space efficient due to constraints on the rate of requests.
- As compared to token bucket-style algorithms (that discard the new requests if there aren't enough tokens), this algorithm services the new requests.

Disadvantages

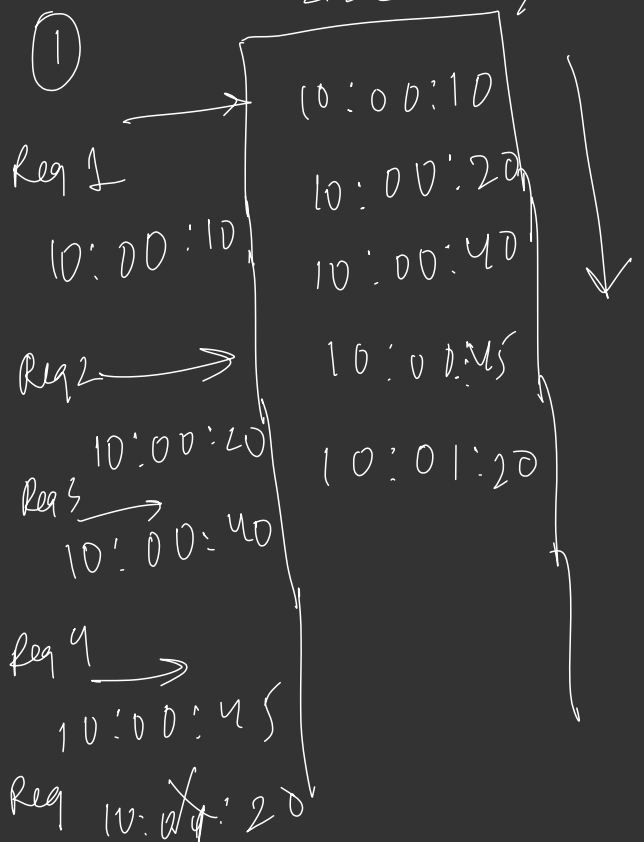
- A consistent burst of traffic (twice the number of allowed requests per window) at the window edges could cause a potential decrease in performance.

4. Sliding Window log

- timestamp

size = 3

3 Req / min



10:00:10

Dis adv

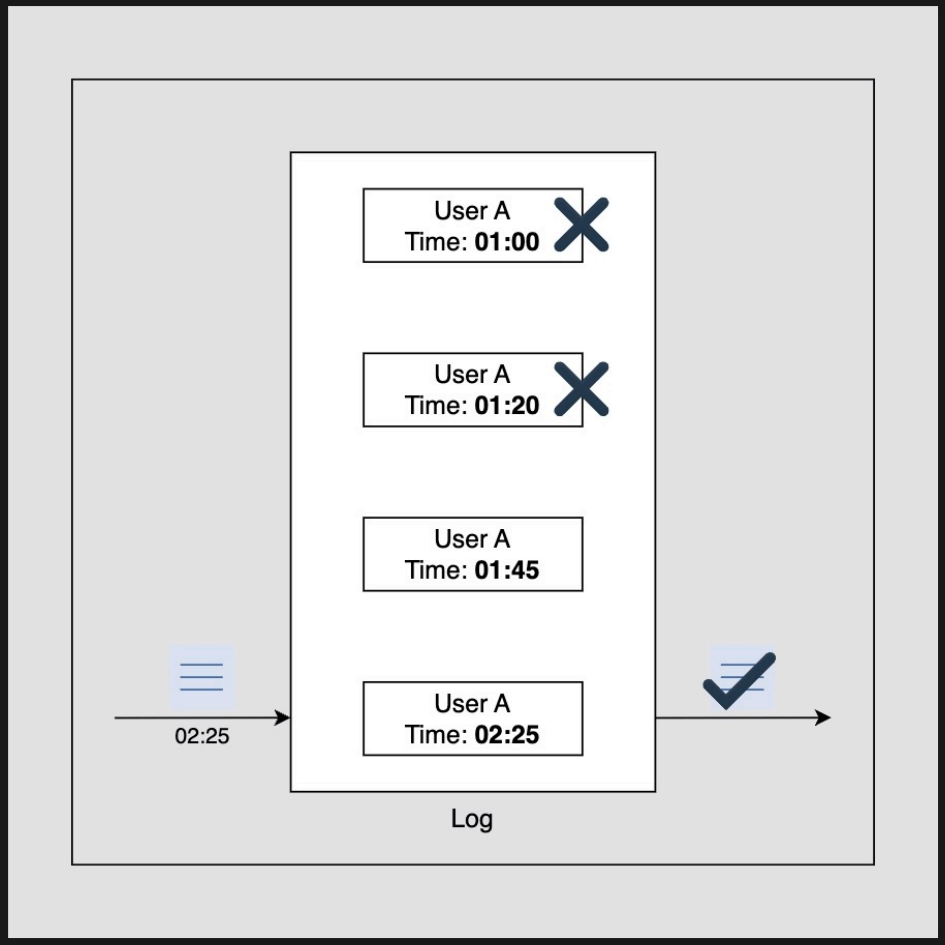
1. even request is denied then also storing time stamp

Advantages

- The algorithm doesn't suffer from the boundary conditions of fixed windows.

Disadvantages

- It consumes extra memory for storing additional information, the time stamps of incoming requests. It keeps the time stamps to provide a dynamic window, even if the request is rejected.

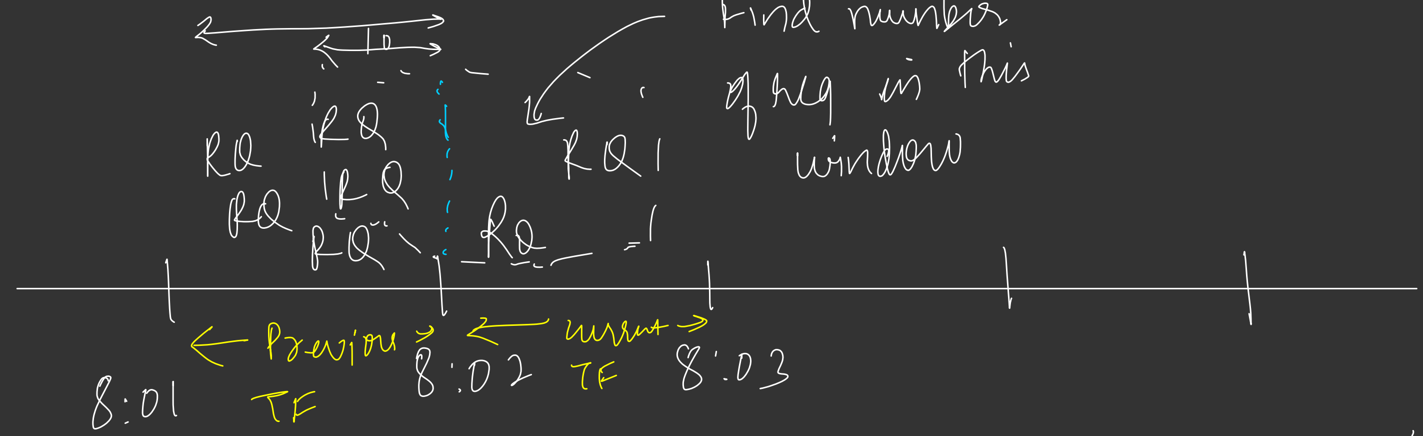


A new request comes in at 02:25, and we start our new window from there. We keep one last window (from 01:25 to 02:25) in the log. The old data is removed from the log, and the size is reduced accordingly

5 Sliding Window counter

Fixed Window + Sliding Window log

60 counter



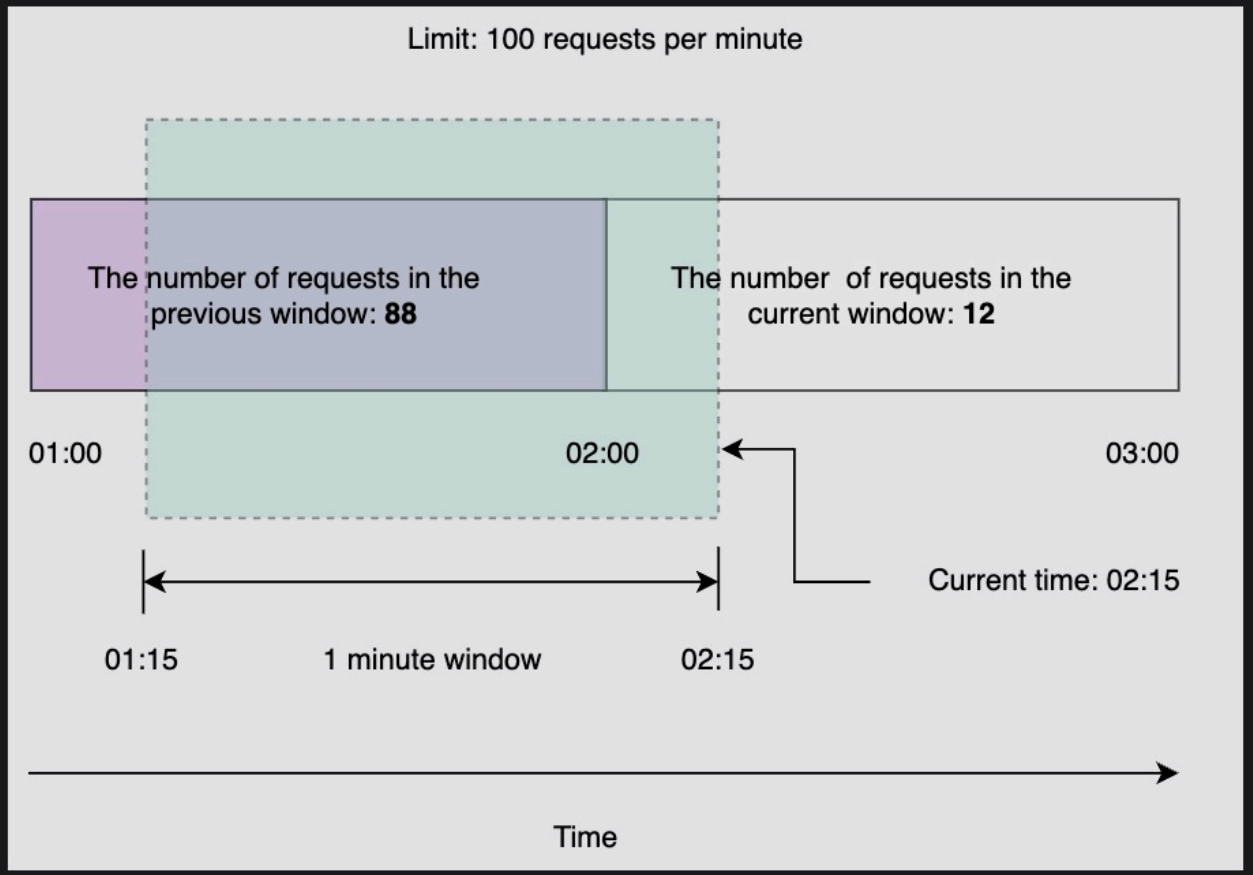
counter=10 counter=10 counter=10

60 → 6

10 → $\frac{6}{60} \times 10$

$2 + \frac{6}{60} \times 10$

$\Rightarrow 3$ ✓



A sliding window counter algorithm, where the green shaded area shows the rolling window of 1 minute

Advantages

- The algorithm is also space efficient due to limited states: the number of requests in the current window, the number of requests in the previous window, the overlapping percentage, and so on.
- It smooths out the bursts of requests and processes them with an approximate average rate based on the previous window.

Disadvantages

- This algorithm assumes that the number of requests in the previous window is evenly distributed, which may not always be possible.

A Comparison of Rate-limiting Algorithms

Algorithm	Space efficient	Allows burst?
Token bucket	Yes	Yes, it allows a burst of traffic within defined limit.
Leaking bucket	Yes	No
Fixed window counter	Yes	Yes, it allows bursts at the edge of the time window and can exceed the defined limit.
Sliding window log	No, maintaining the log requires extra storage.	No
Sliding window counter	Yes, but it requires relatively more space than other space efficient algorithms.	Smooths out the burst