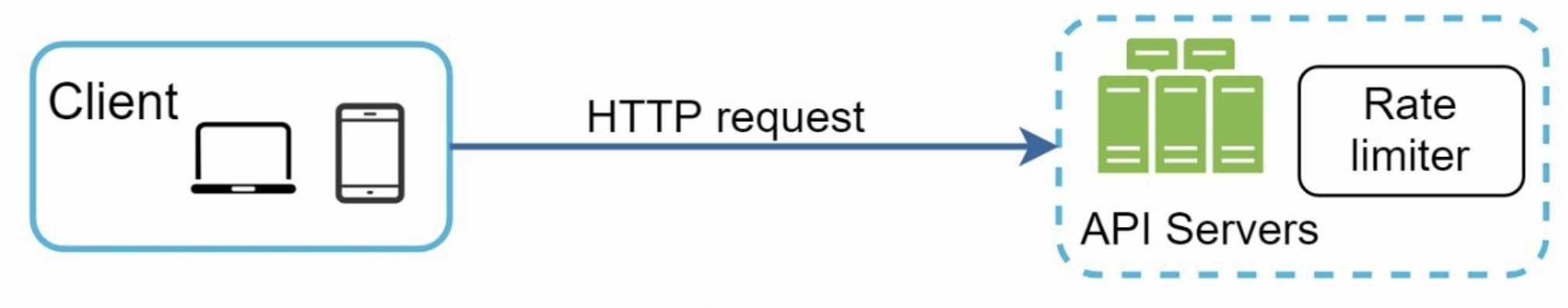
**Rate Limiting**

A rate limiter is used to control the rate of traffic sent by a client or a service. In the HTTP world, a rate limiter limits the number of client requests allowed to be sent over a specified period.

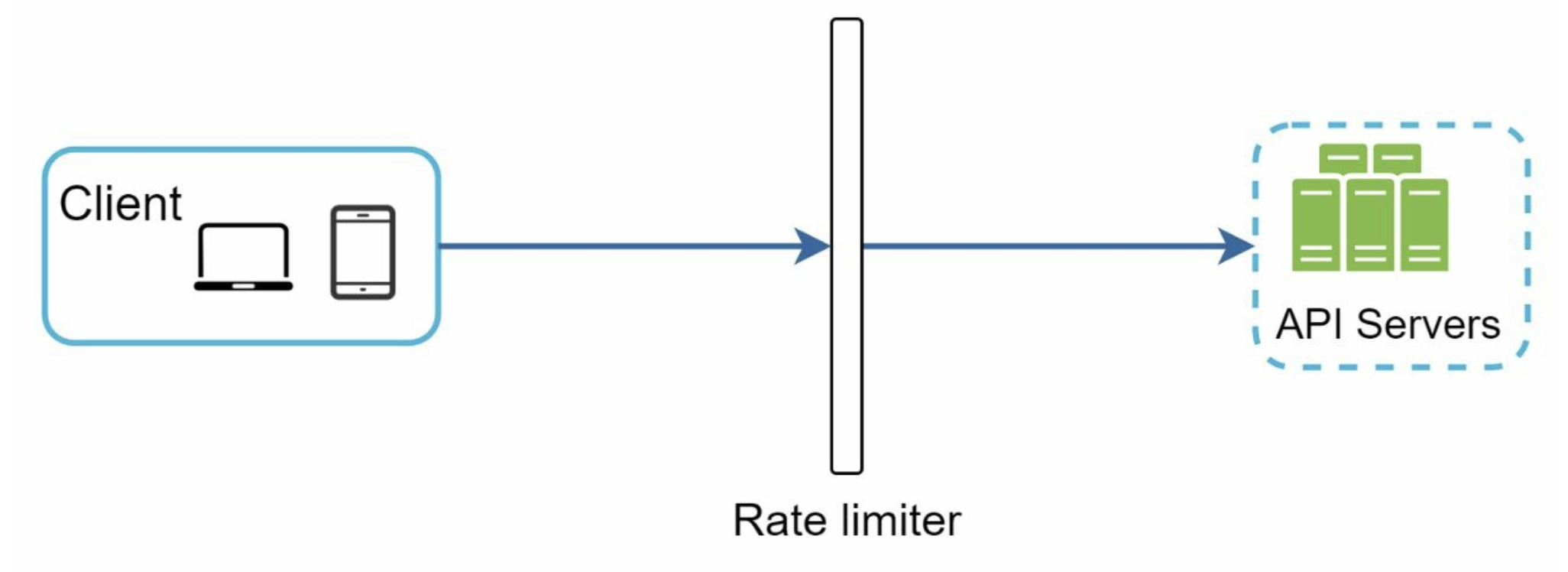
**Where to put the rate limiter?**

Intuitively, you can implement a rate limiter at either the client or server-side.

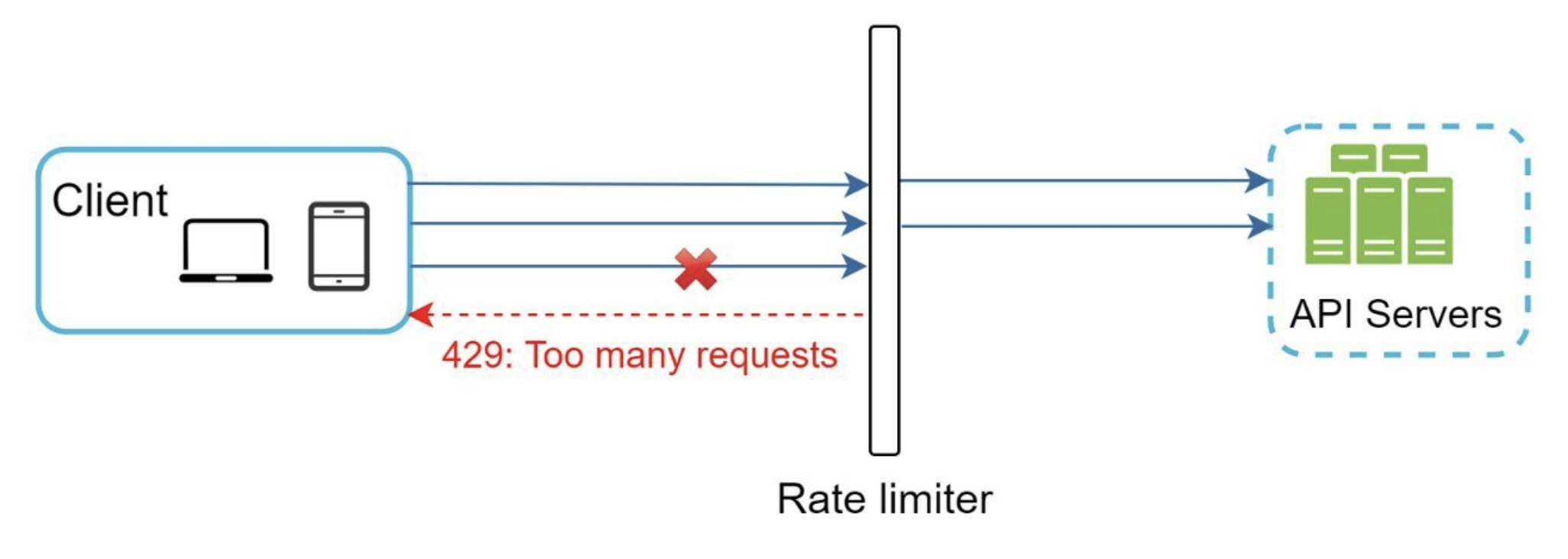
* **Client-side implementation**. Generally speaking, client is an unreliable place to enforce rate limiting because client requests can easily be forged by malicious actors. Moreover, we might not have control over the client implementation.
* **Server-side implementation**. Figure below shows a rate limiter that is placed on the server- side.

****

Besides the client and server-side implementations, there is an alternative way. Instead of putting a rate limiter at the API servers, we create a rate limiter middleware, which throttles requests to your APIs.



Assume our API allows 2 requests/sec, and a client sends 3 requests to the server within a second. The first two requests are routed to API servers. However, the rate limiter middleware throttles the third request and returns a HTTP status code **429**. The HTTP 429 response status code indicates a user has **sent too many requests**.



Cloud microservices have become widely popular and rate limiting is usually implemented within a component called **API gateway**. API gateway is a fully managed service that supports rate limiting, SSL termination, authentication, IP whitelisting, servicing static content, etc. For now, we only need to know that the API gateway is a middleware that supports rate limiting.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## **Algorithms for rate limiting**

• Token bucket

• Leaking bucket

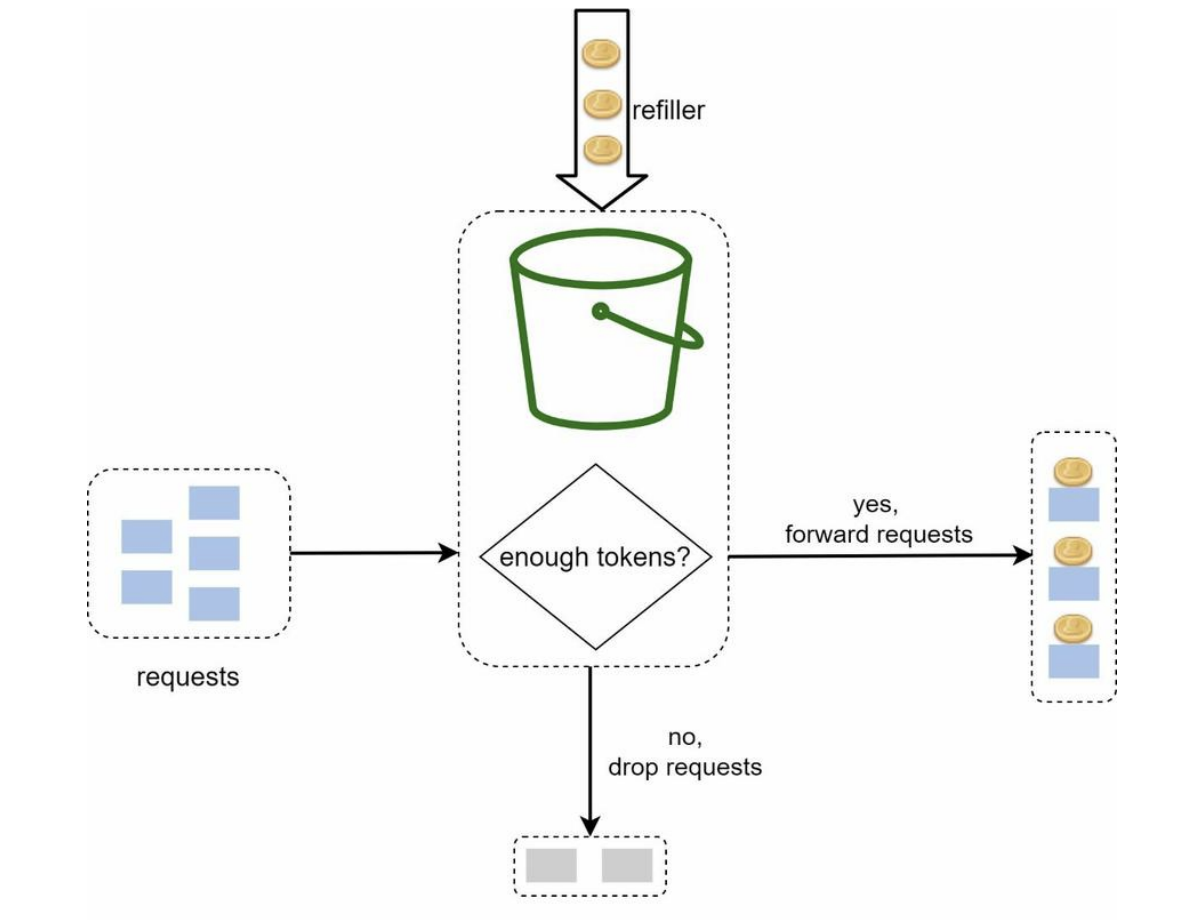
• Fixed window counter

• Sliding window log

• Sliding window counter

#### Token bucket algorithm

* A token bucket is a container that has pre-defined capacity.
* Tokens are put in the bucket at preset rates periodically.
* Once the bucket is full, no more tokens are added.
* The refiller puts 2 tokens into the bucket every second. Once the bucket is full, extra tokens will overflow
* **Used by** – Amazon, Stripe



The token bucket algorithm takes two parameters:

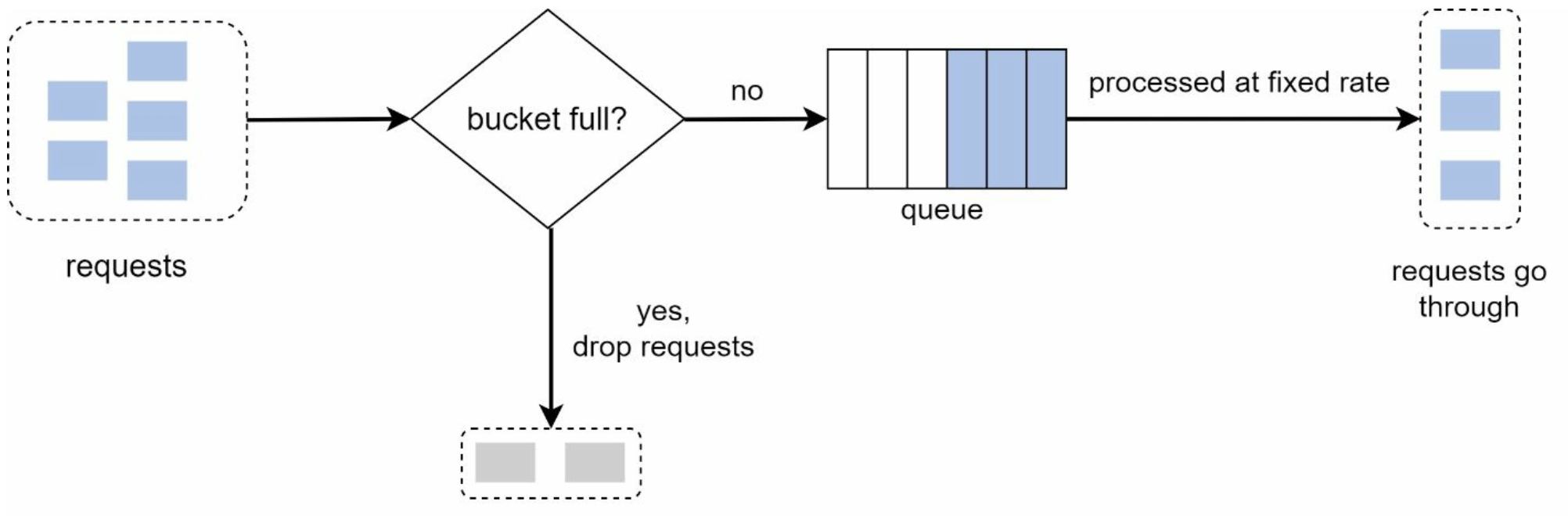
• **Bucket size**: the maximum number of tokens allowed in the bucket

• **Refill rate**: number of tokens put into the bucket every second

**Leaking bucket algorithm**

The leaking bucket algorithm is similar to the token bucket except that requests are processed at a fixed rate. It is usually implemented with a first-in-first-out (FIFO) queue. The algorithm works as follows:

* When a request arrives, the system checks if the queue is full. If it is not full, the request is added to the queue.
* Otherwise, the request is dropped.
* Requests are pulled from the queue and processed at regular intervals.
* **Used by** - Shopify



**Parameters** –

* **Bucket size**: it is equal to the queue size. The queue holds the requests to be processed at a fixed rate.
* **Outflow rate**: it defines how many requests can be processed at a fixed rate, usually in seconds.

Pros:

* Memory efficient given the limited queue size.
* Requests are processed at a fixed rate therefore it is suitable for use cases where a stable outflow rate is needed.

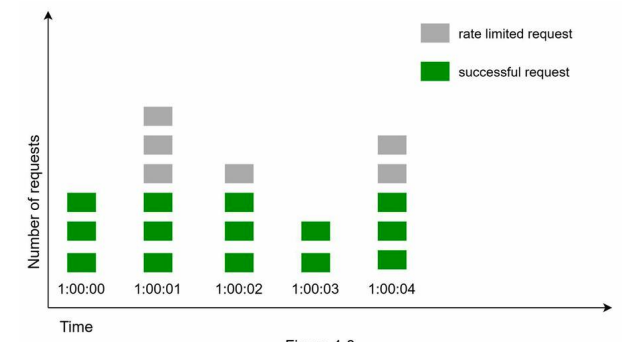
Cons:

* A burst of traffic fills up the queue with old requests, and if they are not processed in time, recent requests will be rate limited.
* There are two parameters in the algorithm. It might not be easy to tune them properly.

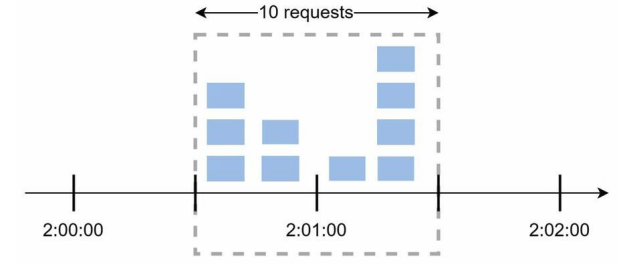
#### Fixed window counter algorithm

Fixed window counter algorithm works as follows:

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



A major problem with this algorithm is that a burst of traffic at the edges of time windows could cause more requests than allowed quota to go through.



Pros:

* Memory efficient.
* Easy to understand.
* Resetting available quota at the end of a unit time window fits certain use cases.

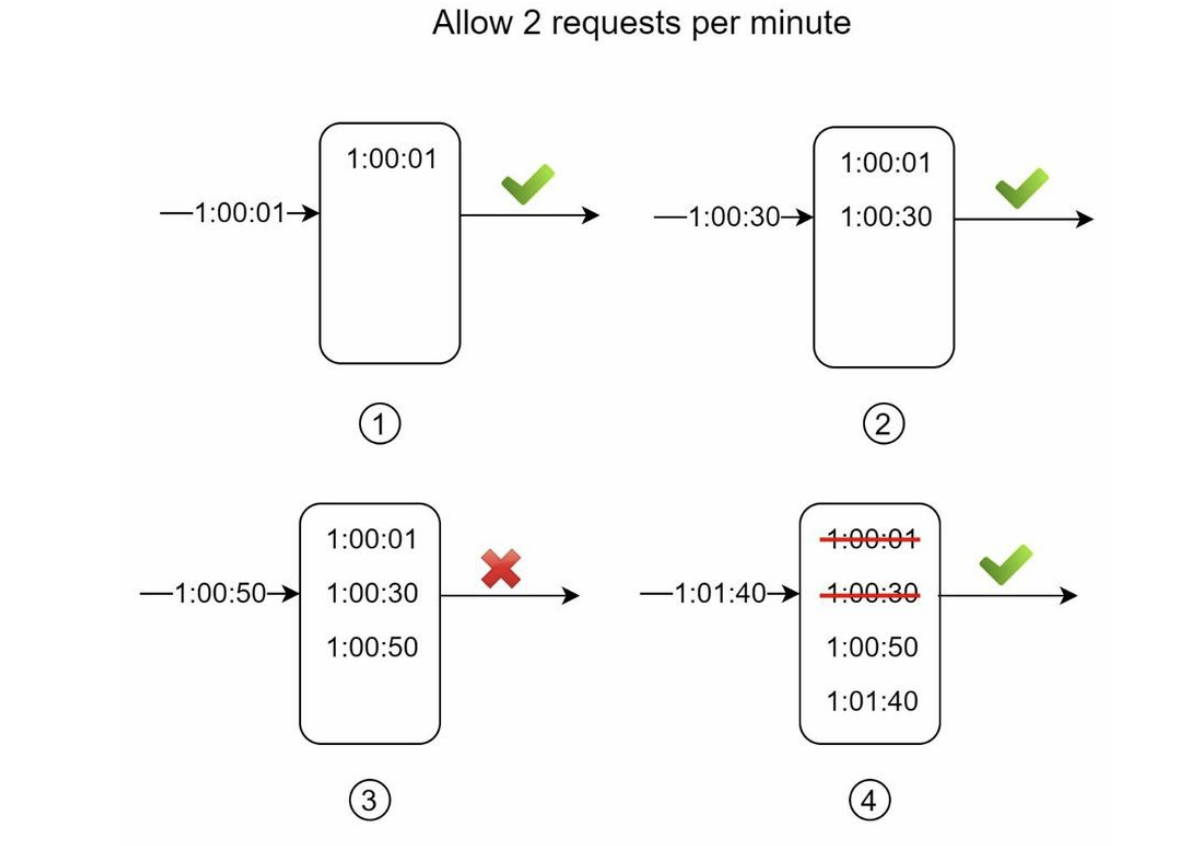
Cons:

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

#### Sliding window log algorithm

The fixed window counter algorithm has a major issue: it allows more requests to go through at the edges of a window. The sliding window log algorithm fixes the issue. It works as follows:

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



Pros:

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

Cons:

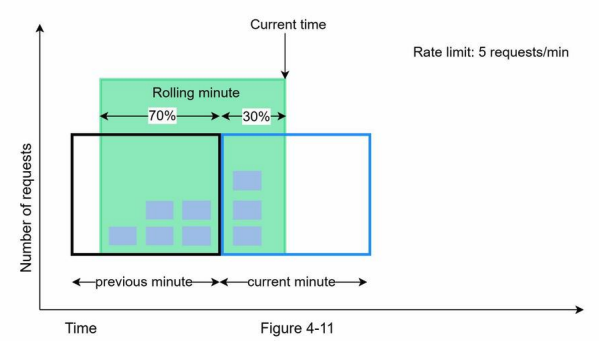
* The algorithm consumes a lot of memory because even if a request is rejected, its timestamp might still be stored in memory.

#### Sliding window counter algorithm

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

* Requests in current window + requests in the previous window \* overlap percentage of the rolling window and previous window.
* Using this formula, we get 3 + 5 \* 0.7% = 6.5 request. Depending on the use case, the number can either be rounded up or down.
* In our example, it is rounded down to 6.

Since the rate limiter allows a maximum of 7 requests per minute, the current request can go through. However, the limit will be reached after receiving one more request.



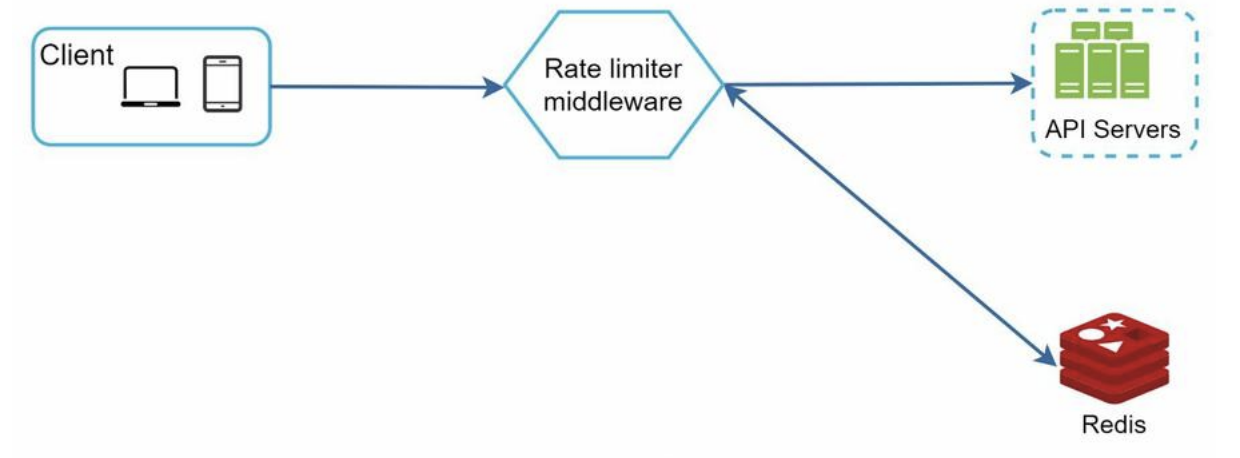
Pros

* It smooths out spikes in traffic because the rate is based on the average rate of the previous window.
* Memory efficient.

Cons

* It only works for not-so-strict look back window. It is an approximation of the actual rate because it assumes requests in the previous window are evenly distributed. However, this problem may not be as bad as it seems.
* According to experiments done by Cloudflare, only 0.003% of requests are wrongly allowed or rate limited among 400 million requests.

**High-level architecture**

****

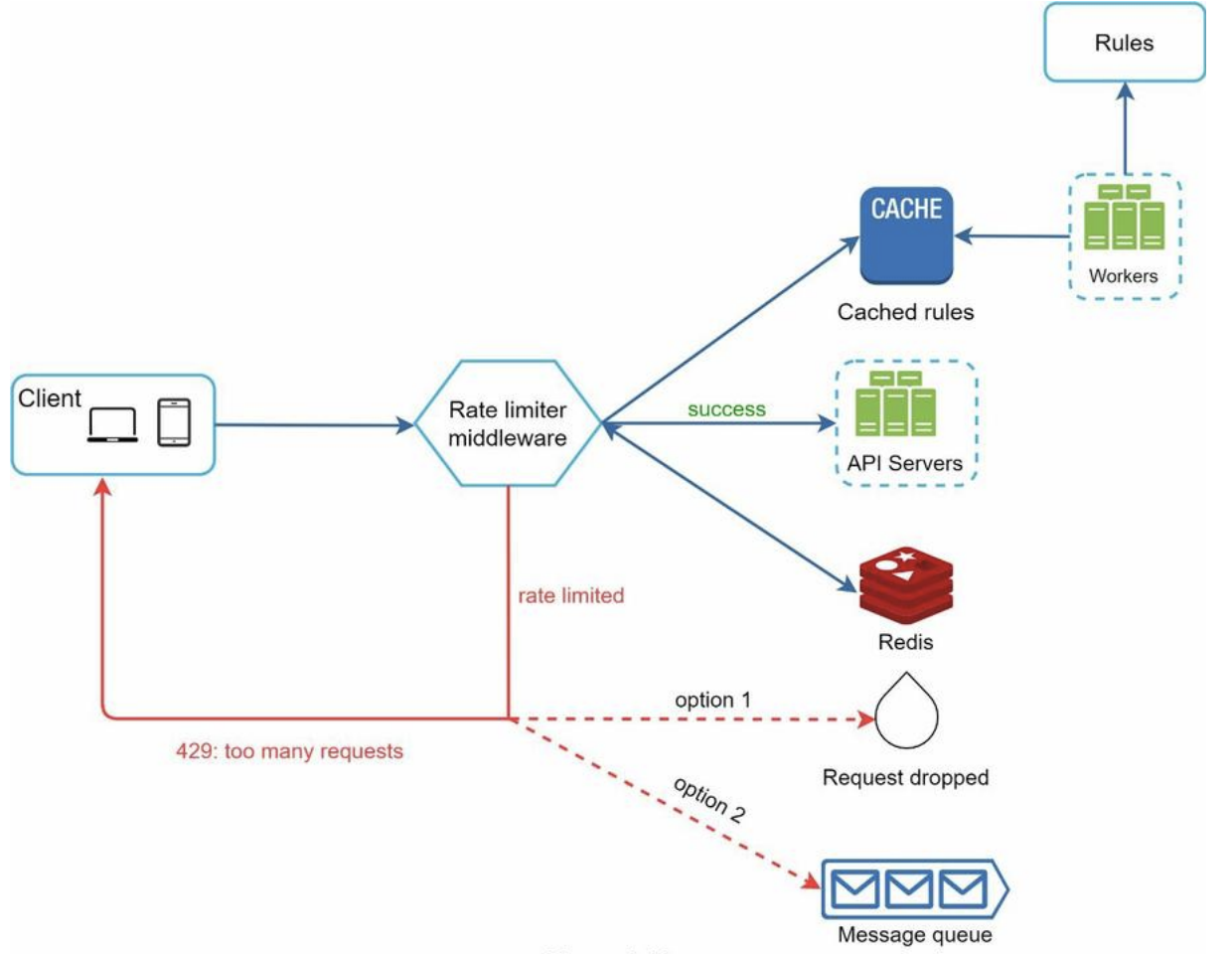
* The client sends a request to rate limiting middleware.
* Rate limiting middleware fetches the counter from the corresponding bucket in Redis and checks if the limit is reached or not.
  + If the limit is reached, the request is rejected.
  + If the limit is not reached, the request is sent to API servers. Meanwhile, the system increments the counter and saves it back to Redis.

**Detailed Design**

How does a client know whether it is being throttled? And how does a client know the number of allowed remaining requests before being throttled? The answer lies in HTTP response headers. The rate limiter returns the following HTTP headers to clients:

****

When a user has sent too many requests, a 429 too many requests error and X-Ratelimit-Retry-After header are returned to the client.

****

* Rules are stored on the disk. Workers frequently pull rules from the disk and store them in the cache.
* When a client sends a request to the server, the request is sent to the rate limiter middleware first.
* Rate limiter middleware loads rules from the cache. It fetches counters and last request timestamp from Redis cache.
* Based on the response, the rate limiter decides:
  + if the request is not rate limited, it is forwarded to API servers.
  + if the request is rate limited, the rate limiter returns 429 too many requests error to the client.
  + In the meantime, the request is either dropped or forwarded to the queue.