Design of a Rate Limiter

We'll cover the following

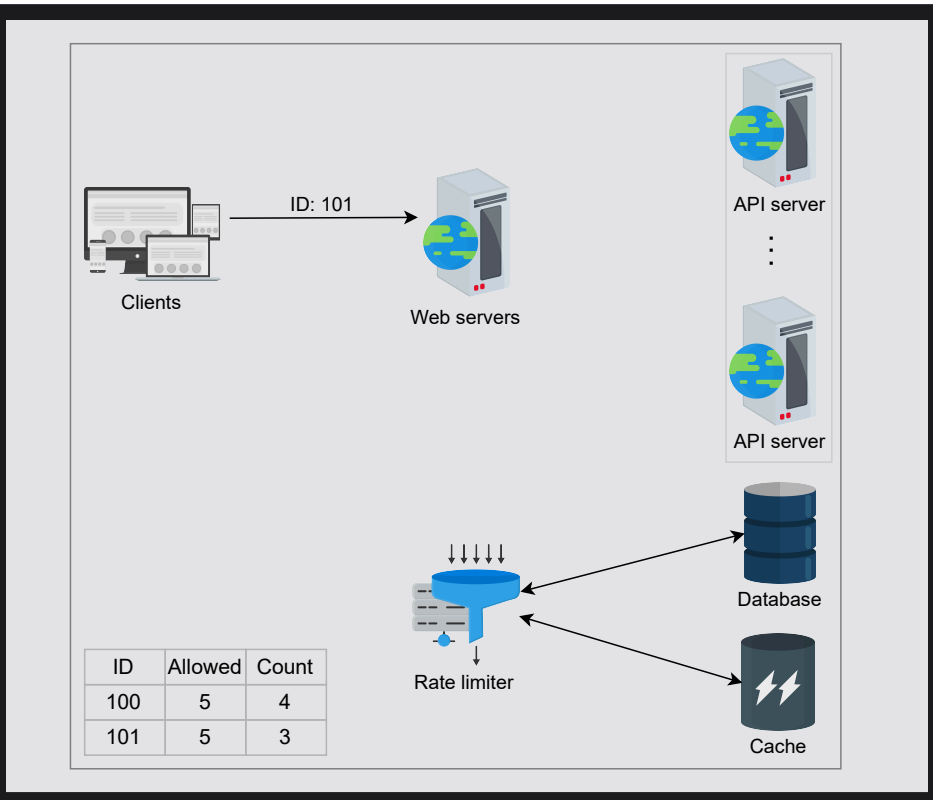
High-level design

A rate limiter can be deployed as a separate service that will interact with a web server, as shown in the figure below. When a request is received, the rate limiter suggests whether the request should be forwarded to the server or not. The rate limiter consists of rules that should be followed by each incoming request. These rules define the throttling limit for each operation. Let's go through a rate limiter rule from Lyft, which has open-sourced its rate limiting component.

```
domain: messaging
descriptors:
   -key: message_type
    value: marketing
    rate_limit:
               unit: day
               request_per_unit: 5
```

Rate-limiting rules from Lyft

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.



A request with ID 101, received by one of the web servers

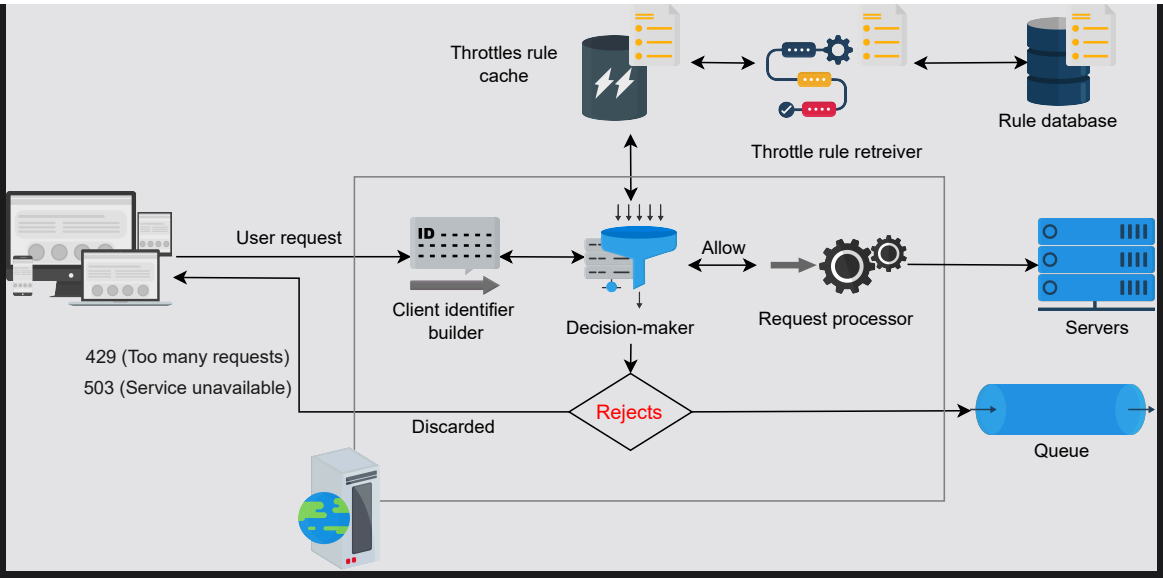
1 of 6

Detailed design

The high-level design given above does not answer the following questions:

- Where are the rules stored?
- How do we handle requests that are rate limited?

In this section, we'll first expand the high-level architecture into several other essential components. We'll also explain each component in detail, as shown in the following figure.



The rate limiter accepts or rejects requests based on throttle rules

Let's discuss each component that is present in the detailed design of a rate limiter.

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.

In case the predefined limit is crossed, APIs return an HTTP response code `429 Too Many Requests`, and one of the following strategies is applied to the request:

- Drop the request and return a specific response to the client, such as "too many requests" or "service unavailable."
- If some requests are rate limited due to a system overload, we can keep those requests in a queue to be processed later.

Request processing

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.

In the event of a failure, a rate limiter is unable to perform the task of throttling. In these scenarios, should the request be **accepted or rejected**?

Give your answer along with proper reasoning in the AI assessment widget below.

Want to know the correct answer?

Should requests be rejected or accepted in face of failures?

H1 H2 H3 | B I | | | |

Provide your answer here!

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.

> **Note:** We can use sharded counters for rate-limiting under the highly concurrent nature of traffic. By increasing the number of shards, we reduce write contention. Since we have to collect counters from all shards, our reading may slow down.

A rate limiter should not be on the client's critical path

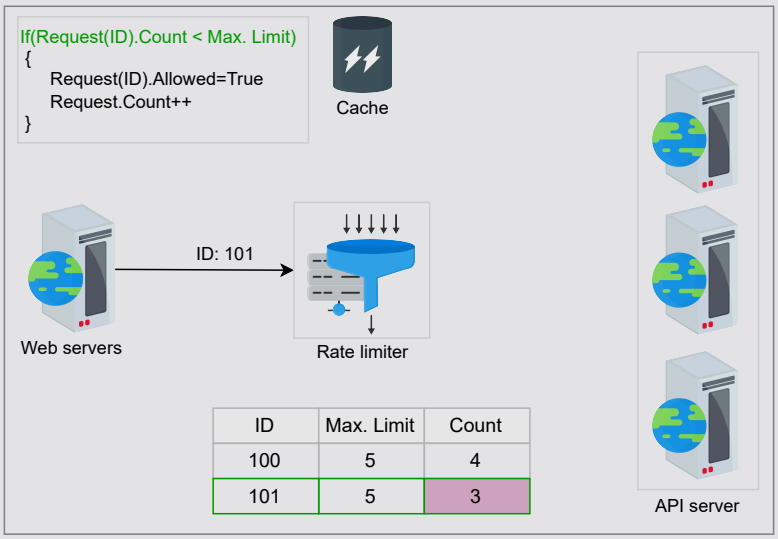
Let's assume a real-time scenario where millions of requests hit the front-end servers. Each request will retrieve, update, and push back the count to the respective cache. After all these operations, the request is sent forward to be served. This approach could cause latency if there is a high number of requests. To avoid numerous computations in the client's critical path, we should divide the work into offline and online parts.

Initially, when a client's request is received, the system will just check the respective count. If it is less than the maximum limit, the system will allow the client's request. In the second phase, the system updates the respective count and cache offline. For a few requests, this won't have any effect on the performance, but for millions of requests, this approach increases performance significantly.

Let's understand the online and offline updates approach with an example. In the following set of illustrations, when a request is received, its ID is forwarded to the rate limiter that will check the condition `if(request(ID).Count <= Max. Limit` by retrieving data from the cache. For simplicity, assume that one of the requests ID is 101, that is, `request(ID) = 101`. The following table shows the number of requests made by each client and the maximum number of requests per unit time that a client can make.

Request ID	Maximum Limit	Count
100	5	4
101	5	3

If the condition is true, the rate limiter will first respond back to the front-end server with an `Allowed` signal. The corresponding `count` and other relevant information are updated offline in the next steps. The rate limiter writes back the updated data in the cache. Following this approach reduces latency and avoids the contention that



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

Note: We've seen a form of rate limiting in TCP network protocol, where the recipient can throttle the