#Rate Limiter

A rate limiter, in general, restricts the number of requests a sender (person, device, IP etc) can send in a given period of time. Once the cap is reached, rate limiter blocks requests.

Why is Rate Limiting used?

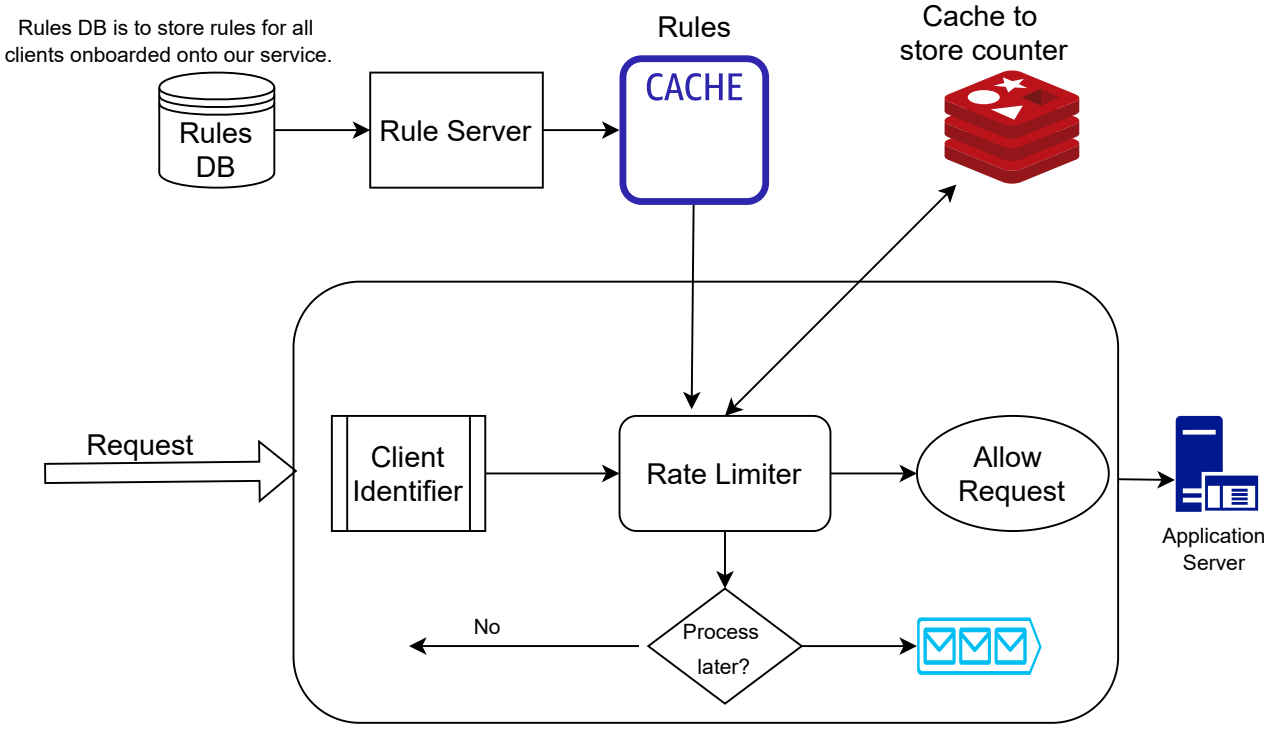
- Avoid resource startvation due to a Denial of Service (DoS) attack.
- Ensure that servers are not overburdened by some particular users.

Where to put the Rate Limiter?

- Client-side implementation: Cons: We may not have complete control over client
- Server-side implementation:
- Rate limiter middleware (or in API gateway):

High level overview

- The underlying concept of rate-limiting algorithms is straightforward. We need a counter at the highest level to track how many requests are submitted from the same user, IP address, etc.The request is rejected if the counter exceeds the limit.
- Where should we keep the counters?: Due to the slowness of disc access, using the database is not a smart option. Because it is quick and supports a time-based expiration technique, an in-memory cache can be chosen. Redis, for example is a popular choice for rate-limiting. "INCR" (increase counter by 1) and "EXPIRE" (timeout for the counter) are two commands that can be used to access in-memory storage.
- A request is sent to rate limiting middleware by the client.
- The rate-limiting middleware retrieves the counter from the associated Redis bucket and determines whether or not the limit has been reached.
- The request is refused if the limit is reached.
- The request is forwarded to API servers if the limit is not reached.
- In the meantime, the system adds to the counter and saves it to Redis.



Alogirithm for Rate Limiting:

Token bucket:

- 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.
- Each API request consumes one token. When a request arrives, we check if there is at least one token left in the bucket. If there is, we take one token out of the bucket, and the request goes through. If the the bucket is empty, the request is dropped.
- Despite the token bucket algorithm's elegance and tiny memory footprint, its Redis operations aren't atomic. In a distributed environment, the "read-and-then-write" behavior creates a race condition, which means the rate limiter can at times be too lenient.If only a single token remains and two servers' Redis operations interleave, both requests would be let through.
- Our token bucket implementation could achieve atomicity if each process were to fetch a Redis lock for the duration of its Redis operations. This, however, would come at the expense of slowing down concurrent requests from the same user and introducing another layer of complexity.

Leaky bucket:

- The leaky bucket algorithm is based on the idea that if the average rate at which water is poured exceeds the rate at which the bucket leaks, the bucket will overflow.
- The algorithm uses a queue, which can be thought of as a bucket that holds the requests, to provide a simple and obvious way to rate limiting.When a request is submitted, it is added to the queue's end. The first item in the queue is then processed at a regular frequency.Additional requests are discarded if the queue is full (or leaked)
- In the leaky bucket algorithm, the requests are processed at an approximately constant rate, which smooths out bursts of requests. Even though the incoming requests can be bursty, the outgoing responses are always at a same rate.
- 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.

Fixed Window:

- We keep a counter for a given duration of time, and keep incrementing it for every request we get. Once the limit is reached, we drop all further requests till the time duration is reset.
- The advantage here is that it ensures that most recent requests are served without being starved by old requests.
- Although the fixed window approach offers a straightforward mental model, it can sometimes let through twice the number of allowed requests per minute. For example, if our rate limit were 5 requests per minute and a user made 5 requests at 11:00:59, they could make 5 more requests at 11:01:00 because a new counter begins at the start of each minute. Despite a rate limit of 5 requests per minute, we've now allowed 10 requests in less than one minute!

Sliding Window:

Let's say We set a limit of 50 requests per minute on an API endpoint.
In this situation, I did 18 requests during the current minute, which started 15 seconds ago, and 42 requests during the entire previous minute. Based on this information, the rate approximation is calculated like this:
rate = 42 * ((60-15)/60) + 18
= 42 * 0.75 + 18
= 49.5 requests

```python
from time import time, sleep


class SlidingWindow:

    def __init__(self, capacity, time_unit, forward_callback, drop_callback):
        self.capacity = capacity
        self.time_unit = time_unit
        self.forward_callback = forward_callback
        self.drop_callback = drop_callback
        self.cur_time = time()
        self.pre_count = capacity
        self.cur_count = 0

    def handle(self, packet):
        if (time() - self.cur_time) > self.time_unit:
            self.cur_time = time()
            self.pre_count = self.cur_count
            self.cur_count = 0
        ec = (self.pre_count * (self.time_unit - (time() - self.cur_time)) / self.time_unit) + self.cur_count
        if ec > self.capacity:
            return self.drop_callback(packet)
        self.cur_count += 1
        return self.forward_callback(packet)
```

Distributed Rate-Limiting:

1) Sticky sessions:

- if we enable "stickiness" on the load balancer, a client should always reach the same instance. And this would allow us to use a simple "local" rate-limiting.
- session stickiness and scaling don't mix well (what's the use of creating new instances, if all existing clients are stuck to the old ones?).

2) Chatty servers:

- each instance call every other one to ask for their current count for a given client, and sum that up. Or we could do it the other way around, with each server broadcasting a "count update" to the others.
- The more instances we have, the more calls need to be made.
- Each instance needs to know the address of every other one, and this has to be updated each time the service is scaled up or down.

3) centralized and synchronous storage system

- in-memory cache (like Memcached or Redis) can be used
- Require strong transactional support (Two instances handling calls for the same client will want to update the same counter/list).