Problem and Design Scope:  
1. What kind of rate limiter are we going to design? Is it a client-side rate limiter or server-side API rate limiter?  
- We focus on the server-side API rate limiter

2. What is the scale of the system? Is it built for a startup or a big company with a large user base?  
- The system must be able to handle a large number of requests.

3. Will the system be distributed in nature?

- Yes  
4. Does the rate limiter throttle API requests based on IP, the user ID, or other properties? - The rate limiter should be flexible enough to support different sets of throttle rules.

5. Is the rate limiter a separate service or should it be implemented in application code?  
- it is a design decision up to you

6. Do we need to inform the users who are throttled?

- Yes

Summary:

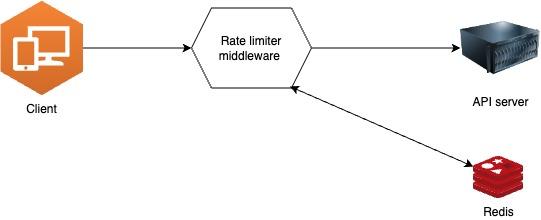
* Accurately limit excessive requests
* Low latency: The rate limiter should not slow down HTTP response time
* Use as little memory as possible
* Distributed rate limiting: The rate limiter can be shared across multiple servers or processes
* Exception Handling: Show clear exceptions to users whenever the user's request is throttled.
* High fault tolerance: If there are any problems with the rate limiter, it does not affect the entire system.

Different types of rate limiter are there:

* Token bucket
* Leaky bucket
* Fixed window counter
* Sliding window log
* Sliding window counter

High level System:

* We need a counter to keep track of requests that are sent from the same user, IP address, etc. If this counter is larger then threshold, the request will be disallowed.
* Where shall we store the counter? We should not store this data in DB as it will increase latency due to slowness of disk access. In memory cache(Redis) is chosen because it is fast and supports time based expiration.
* Redis offers an in memory store that often offers two commands: INCR(increment) and EXPIRE



Design Deep Dive:

* How are rate limiting rules created? Where are the rules stored?
* How to handle requests that are rate limited?

Rate limiting Rules:

Some examples of rate limiting rules:-

1. domain: messaging

descriptors:

- key: message\_type

value: marketing

rate\_limit:  
 unit: day

requests\_per\_unit: 5

1. domain: messaging

descriptors:

- key: auth\_type

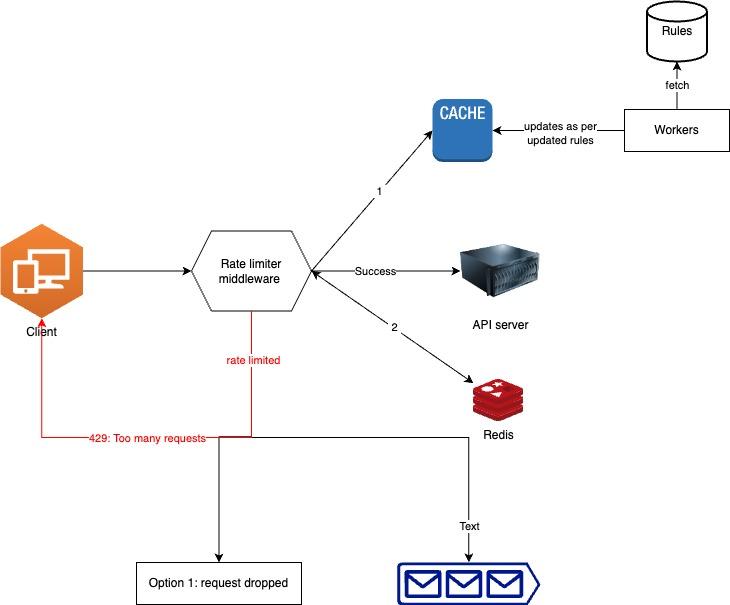
value: login

rate\_limit:  
 unit: minute

requests\_per\_unit: 5

Exceeding the rate limit:

How does a client know whether it is throttled or not? And how does the client know the number of allowed remaining requests before being throttle?  
The rate limiter return following in HTTP headers to clients:  
X-Ratelimit-Remaing  
X-Ratelimit-Limit  
X-RateLimit-retry\_After  
When a user gets a 429 too many request error then these details are given in the header.

Detailed design:  


Rate limiter in a distributed system:

There are two major issues that needs to be handled for rate limiter in distributed system:  
Race condition

Race condition:

Assume the counter value in Redis is 3. If two requests concurrently read the counter value before either of them writes the value back, each will increment the counter by one and write it back without checking the other thread. Both requests (threads) believe they have the correct counter value 4. However, the correct counter value should be 5.

Locks are the most obvious solution for solving race condition. However, locks will significantly slow down the system. Two strategies are commonly used to solve the problem: Lua script [13] and sorted sets data structure in Redis

**Performance optimization**

Multi-data center setup is crucial for a rate limiter because latency is high for users located far away from the data center. Most cloud service providers build many edge server locations around the world. For example, as of 5/20 2020, Cloudflare has 194 geographically distributed edge servers [14]. Traffic is automatically routed to the closest edge server to reduce latency.

Second, synchronize data with an eventual consistency model. If you are unclear about the eventual consistency model

**Monitoring**

• The rate limiting algorithm is effective.

• The rate limiting rules are effective.

For example, if rate limiting rules are too strict, many valid requests are dropped. In this case, we want to relax the rules a little bit. In another example, we notice our rate limiter becomes ineffective when there is a sudden increase in traffic like flash sales. In this scenario, we may replace the algorithm to support burst traffic. Token bucket is a good fit here.

Hard vs soft rate limiting.  
 • Hard: The number of requests cannot exceed the threshold.

• Soft: Requests can exceed the threshold for a short period.

• Rate limiting at different levels. In this chapter, we only talked about rate limiting at the application level (HTTP: layer 7). It is possible to apply rate limiting at other layers. For example, you can apply rate limiting by IP addresses using Iptables [15] (IP: layer 3). Note: The Open Systems Interconnection model (OSI model) has 7 layers [16]: Layer 1: Physical layer, Layer 2: Data link layer, Layer 3: Network layer, Layer 4: Transport layer, Layer 5: Session layer, Layer 6: Presentation layer, Layer 7: Application layer.

• Avoid being rate limited.

Design your client with best practices:

• Use client cache to avoid making frequent API calls.

• Understand the limit and do not send too many requests in a short time frame.

• Include code to catch exceptions or errors so your client can gracefully recover from exceptions.

• Add sufficient back off time to retry logic.