Liam Follow
Aug 23, 2022 · 6 min read · ✦ · Listen

System design: How to design a rate limiter

Introduction

In computer network, rate limiting is used to limit the rate of the requests. Rate limiter is widely used in production to protect underlying services and resources by limiting the number of client requests allowed to be sent over a specified period.

Why rate limiting is used

Benefits:

- Prevent resource st DoS) attack.

- Prevent servers fro lying rate limiting per user to provide fair and reasonable use, without affecting other users.
- Controlling flow. For example, preventing a single worker from accumulating a queue of unprocessed items while other workers are idle. [1]

Requirements

In a system design interview, it's very important to ask questions and clarify the requirements and constraints with the interviewer. Here are a few examples:

- Is it a client-side rate limiter or a server-side API rate limiter?
- What does the rate limiter throttle API requests based on? (Could be user IP, location, user ID, etc.)
- Will the system work for the distributed system?

In this article, we are going to assume the requirements of a rate limiter are:

- Not a client-side rate limiter
- Work in a distributed environment
- High fault tolerance
- High accuracy
- Low latency

High-level design

Client-side or server-side

The reason not to implement it on the client-side is that it's easy to bypass or be forged. Moreover, we might not be able to force using a uniform client.

Besides the client and server-side implementations, there is an alternative way., we create a rate limiter at the middleware layer, which is more intuitive since API gateway become widely popular in a microservice architecture.

Algorithms

Although maintaining a To make Medium work, we log user data. occurrence, there are several different algorit By using Medium, you agree to our Privacy Policy, including cookie policy.

1. **Token bucket**: The bucket has a capacity of tokens and the tokens will be refilled at some rate. Each request will attempt to withdraw a token from the bucket, if there are no tokens in the bucket, the service has reached its limit, otherwise, the request goes through.

2. **Leaky bucket**: A leaky bucket is similar to a token bucket, but the rate is limited by the amount that can drip or leak out of the bucket. The bucket is like a queue or buffer, requests are processed at a fixed rate. Requests will be added to the bucket as long as the bucket is not full, any extra request spills over the bucket edge are discarded.

3. **Fixed window**: Windows are split upfront and each window has a counter. Each request increases the counter by one. Once the counter reaches the threshold, new requests are dropped until a new time window starts. This algorithm is easy to implement but they are subject to spikes at the edges of the window.

4. **Sliding window**: The sliding window algorithm can mitigate the problem mentioned in the fixed window algorithm. The idea of sliding window is to keep track of all request timestamps and calculate whether the counter exceeds the past fixed period when a request arrives, this algorithm is called a sliding window log. Based on this, an optimized algorithm called "sliding window counter" requires fewer operations on the timestamps. I will talk about the difference and implementation detail.

- 4.1 **Sliding window log**: As discussed above, this algorithm keeps track of all request timestamps. When a new request comes in, remove all the outdated timestamps and then count the remaining timestamps. If the counter exceeds the threshold, reject the request. To implement this sliding window log algorithm, we can take advantage of the sorted sets of Redis. We can use \`ZREMRANGEBYSCORE\` to count the timestamps. We can even set a TTL equal to the rate-limiting window and let Redis take care of the timestamp removal.
- 4.2 **Sliding window counter**: Instead of storing and managing all timestamps, this sliding window counter algorithm is the combination of a fixed window and sliding window, and it keeps the benefits of the sliding window while just

requiring just 2 variables to keep track of the rate and counter. The idea of this algorithm is to use t nter to extrapolate an accurate approxima window. It smoothes the traffic spike.

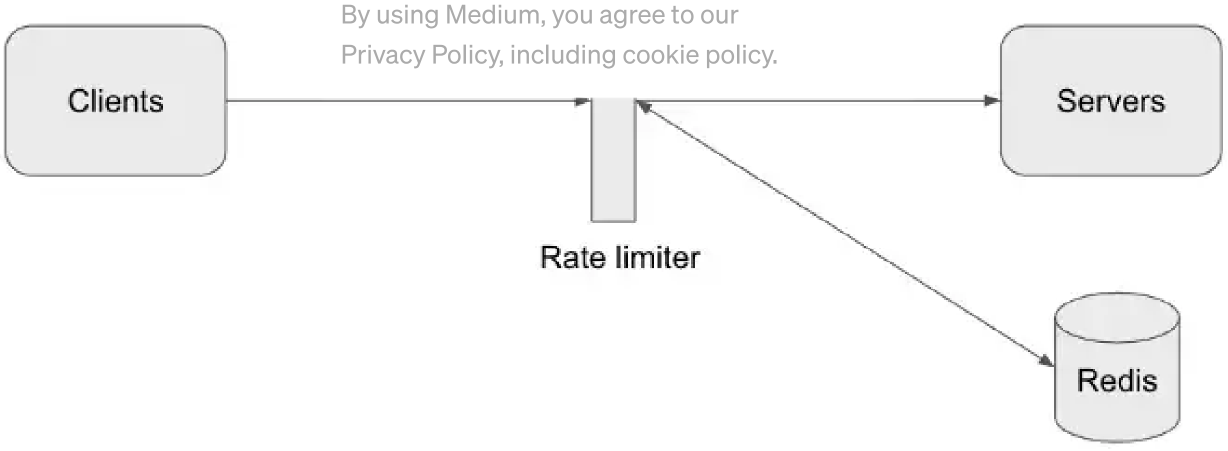
To give you an example, suppose the window period is 1 minute, the counter of the previous window is 42, and the current window starts 15 seconds ago. Based on this, the rate approximation of the current window can be calculated calculated like this: **42 * ((60–15) / 60) + 18 = 49.5**

In conclusion, both algorithms have pros and cons. The bucket algorithms are simple and easy to implement. However, the downside of the token bucket algorithm and leaky bucket algorithm is that both of them have 2 parameters and it's not easy to tune the parameters properly. As for the fixed window algorithm, it allows spikes at the edges of a window, which might cause more requests than the allowed quota to go through. However, sliding windows have the benefits of a fixed window, but the rolling window of time smooths outbursts. [2]

In the following section, we are going to design the architecture of the rate limiter using the sliding window log algorithm.

architecture

We have discussed the algorithms, and now moving to the high-level design. We are going to store the data in the database, in-memory storage is likely a good choice because the rate limiter should respond as quickly as possible. Redis is the best option here because it is fast, has a simple API to manipulate the data, and supports a time-based expiration strategy.

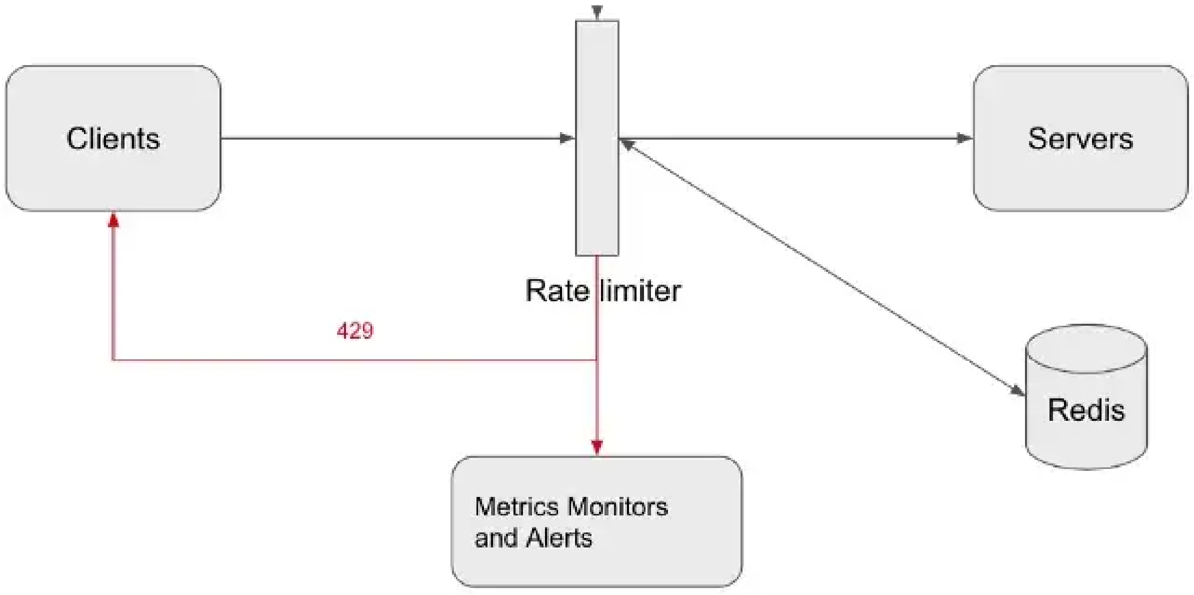


Detailed design

Rules setting and response after being throttled

To support a flexible setting, the rate limiter loads the rules configuration when it starts or has a mechanism to allow users to update the rules. Here we introduce a config database to persist the configuration.

Once a request is rate limited, the rate limiter should be able to notify its client. Suppose we use HTTP protocol, code 429 can be returned and in the response, headers like **X-Ratelimit-Remaining** and **X-Ratelimit-Retry-After** can be added to tell the clients more info about the rate limit.



Race condition

Race condition might occur in a highly concurrent environment with the sliding window log algorithm because it needs to remove, add and count the timestamps. The data becomes stale while multi-threads/instances are processed at the same time. However, we can take advantage of the *MULTI* command. In this mode, all Redis operations can be performed as an atomic action. This means that if two processes both try to perform an action for the same user, there's no way for them to not have the latest information, which prevents the race condition. [4]

Recap

In this post, we discussed the requirements of a rate limiter and different algorithms of rate-limiting and their pros/cons. We also proposed the architecture of our design, talked about the potential issues in practice, and provided solutions to them.

References

1. Rate-limiting strategies and techniques — Google Cloud

2. How we built rate limiting capable of scaling to millions of domains

3. Techniques for enforcing rate limits — Google Cloud

4. Better rate limiting with Redis sorted sets

5. System Design Inte

Originally published on https://liamchzh.com/tech/2020/11/18/system-design-4/

System Design Interview Programming Software Engineering Faang Backend