API Rate Limiter System Design

What is a Rate Limiter?

At most basic level, rate limiter restricts the number of events that a certain user or device can perform within a given time range. It helps us to limit the number of requests a sender can send in a particular period of time and is implemented as a protective mechanism to prevent excessive use of services. Once the upper limit is reached, the rate limiter blocks further requests.

In this blog, we will discuss components and algorithms related to design rate limiter.

Why is rate limiting used?

- Avoid resource starvation due to a Denial of Service (DoS) attack.
- Ensure that servers are not overburdened. Using rate restriction per user ensures fair and reasonable use without harming other users.
- Control the flow of information, for example, prevent a single worker from accumulating a backlog of unprocessed items while other workers are idle.

Requirements

It’s critical to ask questions and clarify needs and restrictions with the interviewer during a system design interview. Listed below are a few examples:

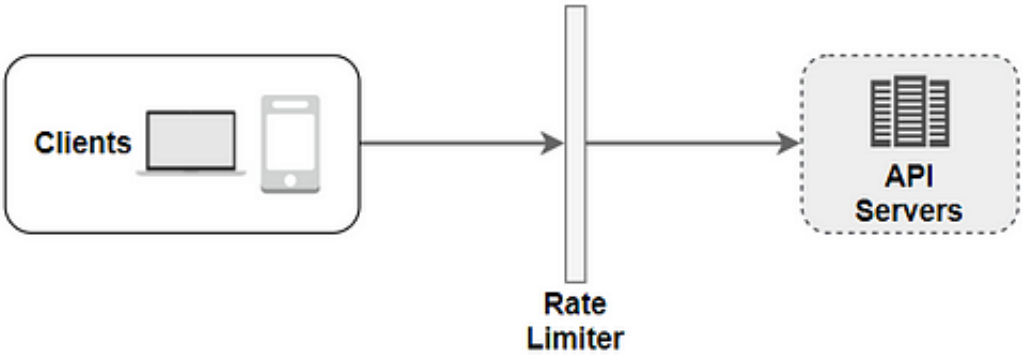
- Is the rate limiter configured to throttle API requests based on IP, user ID, or other parameters?
- What is the overall size of the system?
- Is the rate limiter on the client or the server-side API?

High-Level Design of Rate Limiter

Let’s keep things simple and communicate using a basic client-server approach. Where should the rate limiter be placed? A rate limiter can be implemented intuitively on either the client or server side.

- Client-side implementation: Hostile actors can quickly falsify client requests, a client is an unstable venue to impose rate restrictions. Furthermore, we may not have complete control over client implementation.
- Server-side implementation: On the server-side, a rate restriction is shown in the diagram below.

There is an alternative to the client and server-side implementations. We construct a rate limiter middleware, which throttles requests to your APIs, rather than establishing a rate limiter on the API servers, as indicated in the diagram below.

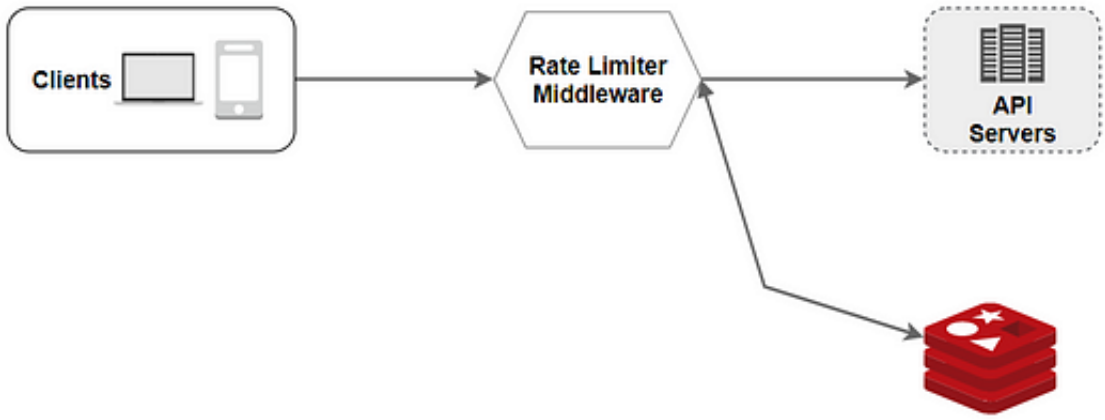


The underlying concept of rate-limiting algorithms is straightforward. We need a counter at the highest level to track how many requests are submitted from the same user, IP address, etc. The request is rejected if the counter exceeds the limit.

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” and “EXPIRE” are two commands that can be used to access in-memory storage.

- INCR: Used to increase the stored counter by 1.
- EXPIRE: Used to set a timeout for the counter. The counter is automatically deleted when the timeout expires.



- 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.

Rate Limiting Algorithms

There are numerous rate-limiting algorithms available. The use case, we suppose, is to limit the number of queries sent to a server.

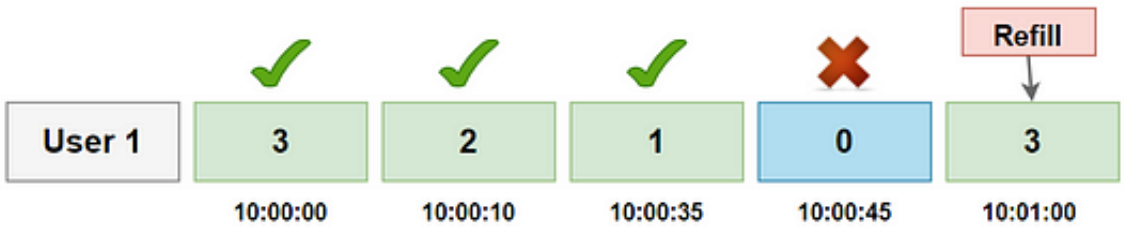
Consider the case where we need to create an API Rate Limiter that limits API calls based on the user (We can change implementation according to our criteria). A fixed number of requests can be sent per user in a fixed

time range. Let's look at the algorithms that are utilised for API rate limiting:

Token Bucket

Assume a bucket has a few tokens. When a request comes in, a token from the bucket must be taken and processed. The request will be refused if no token is available in the bucket, and the requester will have to try again later. As a result, the token bucket gets refreshed every time unit.

By allocating a bucket with a predetermined number of tokens to each user, we may limit the number of requests per user per time unit. When a user uses up all of his tokens in a certain amount of time, we know he or she has reached their limit, and we deny his/her requests until bucket is refilled.



Token Bucket API Rate Limiter

- There is a limit of 3 requests per minute per user.
- User 1 makes the first request at **10:00:00**; the available token is 3, therefore the request is approved, and the available token count is reduced to 2.
- At **10:00:10**, the user's second request, the available token is 2, the request is permitted, and the token count is decremented to 1.
- The third request arrives at **10:00:35**, with token count 1 available, therefore the request is granted and decremented.
- The 4th request will arrive at **10:00:45**, and the available count is already zero, hence API is denied.
- Now, at **10:01:00**, the count will be refreshed with the available token 3 for the third time.

Pros:

- It’s simple and straightforward to use.
- Memory-friendly

Cons:

- A race condition counter may cause an issue in a distributed system due to concurrent requests from the same user.

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. This 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).

Leaky Bucket API Rate Limiter

As you can see, if 3/Request/User is allowed and the Queue size is 3, the API Rate Limiter will block the 4th Request.

Pros:

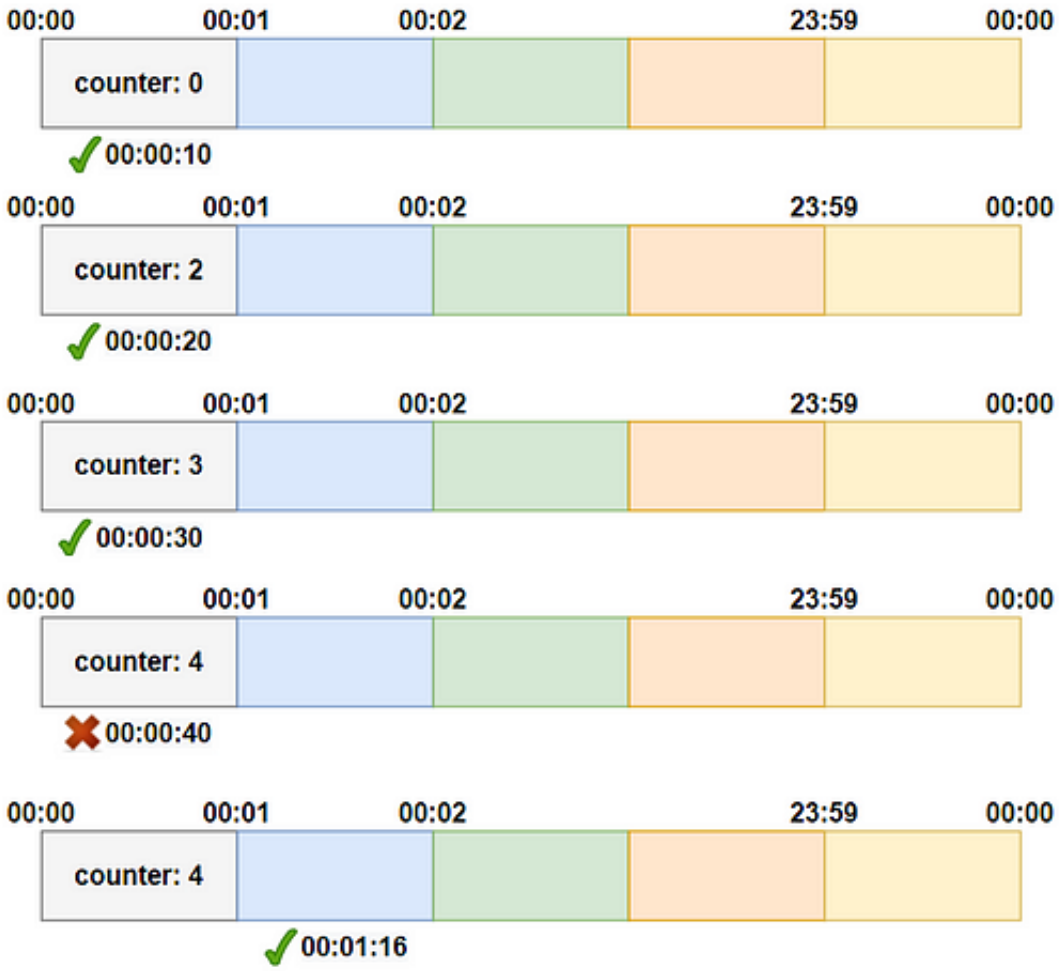
- It’s simple and straightforward to use.
- Memory-friendly

Cons:

- A race condition counter may cause an issue in a distributed system due to race conditions.

Fixed Window Counter

We will keep a bucket per unit time window in this algorithm and update the count requests made in that time window. We will cancel the request if it exceeds the limit. A rate limiter for a service that only allows 3 requests per minute, for example, will have the data model below. The buckets are one-minute windows, with values holding the counts of requests seen during that minute.



If a new request is seen at 00:00:10, we get the count from the current bucket (00:00-00:01), which is 0, and check if processing one more request exceeds the permissible limit of 3, in which case an exception is raised; if not (which is the case here), the bucket's count is incremented by unit 1 and the request is allowed to be processed. When a request arrives at 00:00:40 and the window count is three, no request is

authorised and an exception is triggered. When a new window is created, just the counts of the current window are saved, and older windows are erased (i.e in the above case, if the min changes, the older bucket is deleted)

Space complexity: O(1) for storing the count for the current window Time complexity: O(1) for get and simple atomic increment operation

Pros:

- Easy to put into action.
- Smaller memory footprint, because all that's done is store the counts.
- Can leverage Redis-like technology with built-in concurrency.

Sliding Window Logs

Sliding Log rate limitation keeps note of each consumer's request in a time-stamped log. These logs are normally stored in a time-sorted hash set or table. Logs with timestamps that exceed a certain limit are discarded.

- A queue/sorted list of timestamps indicating the moments when all historical calls happened within the time range of the most recent window is maintained for each user.
- When a new request is received, a check is made for any timestamps older than the window time, and these are deleted as they are no longer relevant. (Instead of doing this at every request, this step can also be done on a regular basis after every 'n' minutes or when a certain length of the queue is reached.)
- The user's queue/set is updated with the new timestamp. The request is allowed to proceed if the number of elements in the queue does not exceed the authorised count; otherwise, an exception is triggered.
- Remove all timestamps older than "CurrentTime—1 minute" from the Sorted Set.
- Count how many elements there are in the sorted set. If this number

exceeds our “3” throttling limit, reject the request.
- Accept the request and add the current time to the sorted set.

Space complexity: O(Max requests seen in a window time)—In a time window, all of the requests’ timestamps are saved.

Time complexity: O(Max requests seen in a window time)—A portion of timestamps is deleted.

Pros:

- It works flawlessly

Cons:

- Memory usage is high. To manage numerous users or huge window times, all of the request timestamps must be kept for a window time, which necessitates a lot of memory.
- Removing earlier timestamps has high time complexity.

Sliding Window Counters

The low processing cost of the fixed window algorithm is combined with the increased boundary constraints of the sliding log in this hybrid approach. To begin, we track a counter for each fixed window, similar to the fixed window technique. To reduce surges of traffic, we account for a weighted value of the previous window’s request rate based on the current date.

We keep track of each user’s request counts by using various fixed time windows, such as 1/60th the size of our rate limit’s time frame. If we have a minute rate limit, for example, we can record a count for each second and calculate the sum of all counters in the previous minute when we get a new request to determine the throttling limit.

1. Remove any counters that are more than one minute old.
2. If a request is received in the current bucket, the counter will be increased.

3. If the current bucket has reached its throat limit, the request will be blocked.

Space Complexity: O(number of buckets)

Time Complexity: O(1)

Pros:

- Because only the counts are saved, there is no big memory footprint.

Cons:

- Only works for non-strict look-back window times, such as smaller unit times.

References

1. Rate-limiting strategies and techniques—Google Cloud
2. How we built rate-limiting capable of scaling to millions of domains

Thanks to **Navtosh Kumar** for his contribution to creating the first draft of content. Please write in the message below if you find anything incorrect, or if you want to share more insight. Enjoy learning, Enjoy algorithms!