**Rate Limiting**

Some of the things that we'll cover in this module are:

1. What is Rate Limiting
2. Why do we need it
3. What are some common ways to implement rate limiting

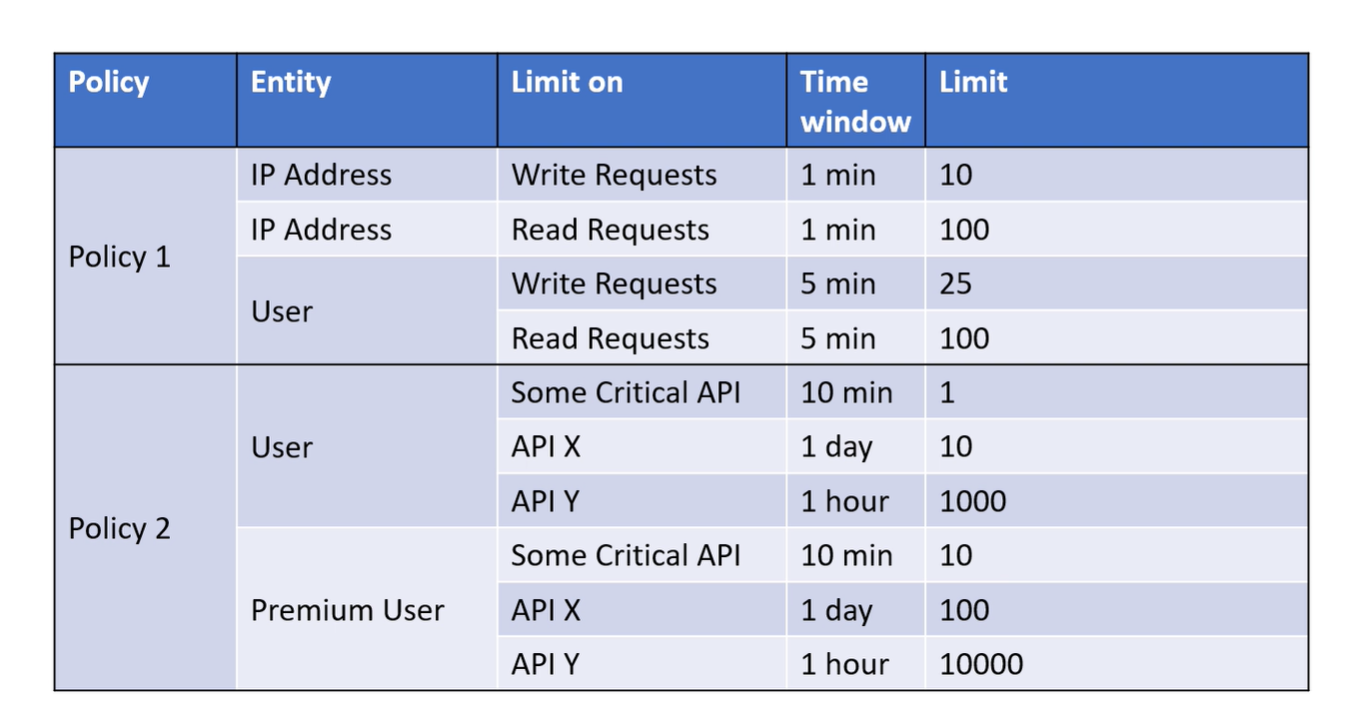
What is Rate Limiting?

**Rate limiting is a method of limiting the amount of network traffic that can be sent or received for a particular client in a specific time window.**

When we say network traffic, it could be the number of requests or number of reads and writes, or simply the payload size. And the entity you want to apply the limit on could be a user, an IP address, a data center, basically anything that can send a request to your system.

Now we say we are going to limit the requests initiated by an entity. What happens when they exceed the number of allowed requests? Well, we could ask them to fill a captcha to ensure it is not a bot programmed to send requests to our system, or we could configure our rate limiter to simply drop the request.

This rate-limiting logic can be simple, based on the number of hits, or it could have multiple limiting policies. It could also have complex policies like if a so and so the user is hitting our system for X API endpoint, they are only allowed to send 10 requests a day, but if they send requests for Y API endpoint, they will be allowed 1,000 requests per hour.



If the same requests were coming from a premium user, these limits would substantially increase.

Why Do We Need Rate Limiting?

Let us look at a few use cases where Rate Limiters might come in handy.

Bots

Say a competitor of Amazon wants to get the list of products they are selling, ratings for those products, and prices and discounts. Once they have this data, they can simply sell the same products at a slightly lower price than Amazon. How does Amazon protect itself from something like this? Amazon can simply check if all these requests are coming from the same IP address. If yes, it can apply a limit to the number of requests that can be made in a certain time window, not allowing their competitor to get the data.

DoS/DDoS Attacks

Let’s say our servers are able to handle the load in a normal scenario. But suddenly, we start getting thousands of requests per minute from a set of IP addresses. These IPs could belong to a set of attackers trying to overload our system and possibly break it. If they succeed, our actual users won't be able to access the system. Applying a rate limit to only allow a maximum number of requests from each IP address in a set time would be a good way to prevent such attempts.

SPAM

Let’s say a product with decent ratings suddenly starts getting really bad reviews. Now, this could, of course, mean that the product quality has deteriorated. But what if the users who recently bought this product were all using the same credit card or if all these requests were coming from the same geographic location? Chances are this is a competitor trying to SPAM our system with manipulated data. Rate limiting might be a good way to put a stop to this.

How to implement this?

So now we know what kind of problems rate limiting can solve. But how do we actually go about it? Well before we get to that let us look at a few things we need to keep in mind.

* The entity on which we are applying the limit
* A counter that keeps track of actions by the entity
* A configuration of how many actions of what type are allowed under what conditions
* Based on all this we need to decide if the request should be allowed or dropped

Algorithms

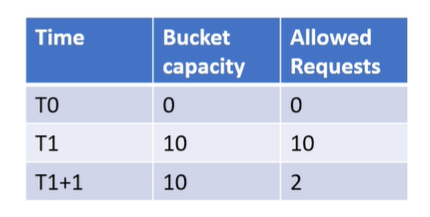
Now let us have a look at some algorithms that can be used to implement rate limiters.

Leaky Bucket

The idea is to maintain an upper bound of how many actions we want to allow in a time frame. We use a bucket analogy to keep track of this upper bound. Each incoming request increments the quantity in the bucket by a unit. When the bucket is full, we stop allowing the requests. The bucket will also leak at a certain rate. For example, a bucket leaking at 2 units per second means that even in high traffic conditions, we will allow 2 requests per minute. Thus, the size of the bucket and the flow rate are the only two parameters needed to define a leaky bucket rate limiter.



Now, there is a problem with this approach. If we have an empty bucket at T0, a user makes 10 calls at T1 = T0+1, we will allow it as it is within the limit. Our bucket is also leaking at 2 units/second. So if we get 10 requests at T1+1, we will allow 2 of them.



So we have allowed 12 requests to be serviced in 2 seconds, which is not the expected behavior.

However, this is a really good algorithm where we don't need to have super-strict time bound rate limits but we do want to maintain a rate limit over a longer period of time. Most of the non-monetary use cases usually fall in this bucket.

Fixed Window

Say we want to allow 4 requests in an hour. Each hour will have individual windows such that all the requests from 9-10 will end up in that bucket. We keep track of the count of requests for each hourly window and the moment it reaches the limit we start dropping requests.

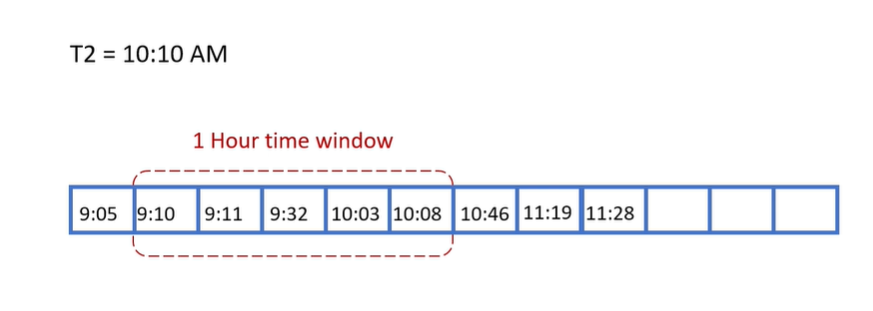


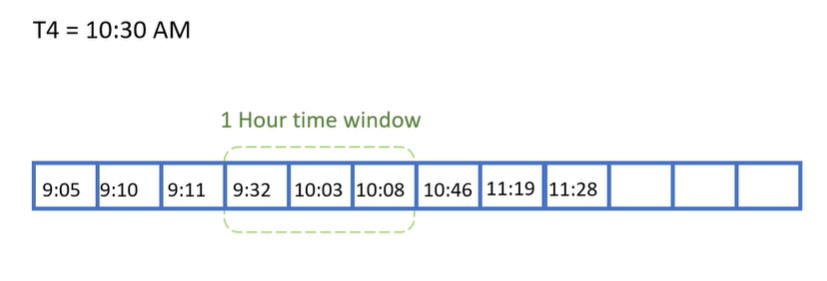
The problem is this approach is again of exceeding the rate limit. Lets say a user sends in 4 requests at 8:59 and another 4 at 9:00. We were supposed to allow 4 requests in an hour, and we have already served 8 in a span of 2 minutes, which for some use-cases, could be a problem.

Good use-cases for this algorithm are scenarios where we are okay with slight bursts of requests but do want rate limits over longer period of time, specially scenarios where we get requests once in a while, say something like a reconciliation system.

Sliding Window

In this approach, we will store the requests in a queue. Let's say our time frame is for 1 hour and we allow 4 requests each hour. We will map the requests in the window such that it spans an hour. We will keep moving the window with respect to the time of the incoming request, keeping the time frame covered by the window under an hour. If the number of serviced requests is less than 4, we will allow it otherwise it will be dropped. This means at any given time, in a one hour window, we will process 4 requests at most which is highly accurate.





This comes at a cost though, as instead of just keeping track of a counter, we are keeping track of requests along with the time they were allowed in a queue data structure. This needs more memory and CPU resources.

This algorithm is what you would want to use for most of your rate limiting requirements and specially in situations where you want accurate rate limits, say if you are providing SMS/Email sending capabilities to your clients and are limiting them based on which subscription they are on.

What Happens in the Real World

Now, even though we have seen the examples for limits as small as 4 requests an hour, in a real scenario the limits would be much larger. Irrespective of the limit though, the algorithm would still be applicable as discussed.

So now you know how to implement a rate limiter. But do we really need to build it from scratch? Well that is of course a possibility, but there are a lot of frameworks and libraries that provide predefined functionalities to perform rate limiting. Similarly, a lot of API gateways and cloud providers also offer this capability, so it might be worth checking out, before you decide to build your own system.