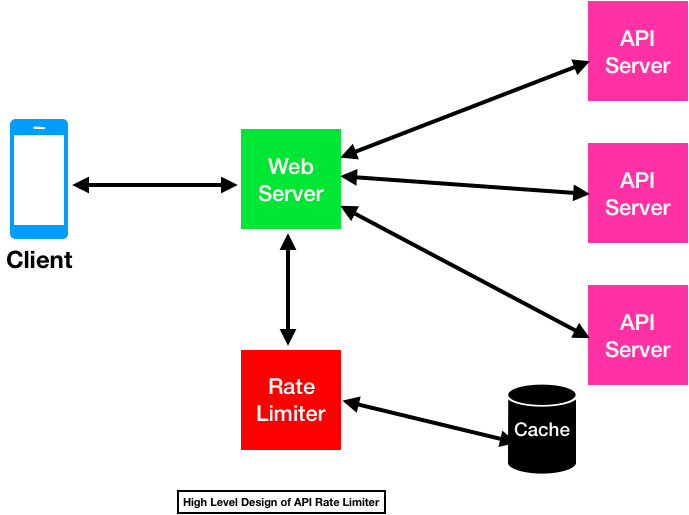
**API Rate Limiting System**

A rate limiter, at a high-level, limits the number of events an entity (user, device, IP, etc.) can perform in a particular time window.

In general, a rate limiter caps how many requests a sender can issue in a specific time window. It then blocks requests once the cap is reached.



**Why do we need Rate Limiting System?**

Protect services for malicious attack, overuse, occasional request spikes, DDoS attack, brute-force password attempts, security, reduce costs, revenue.

**Functional Requirements:**

* Limit the number of requests an entity can send to an API within a time window
* (Distributed Scenario) The APIs are accessible through a cluster, so the rate limit should be considered across different servers

**Non-Functional Requirements: (Availability, Latency, Scalability)**

* Highly available
* Not introducing substantial latencies

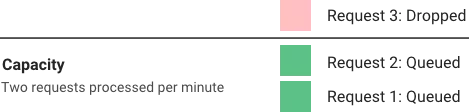
For these use cases, using persistent memory stores like MySQL is a bad idea because the time taken for disk seeks is high enough to hamper the rate limiter granularity. For instance, let’s say that we’re using a single MySQL instance to store the request counts and MySQL takes 1ms to process 1 request, which means we can achieve a throughput of 1000 req/sec. But a rate limiter using in-memory cache takes around 100nanosec (a main-memory access) to process 1 request, which implies a granularity of around 10Mreq/sec can be achieved.

**Algorithms for Rate Limiting**

**Leaky Bucket**

Leaky bucket (closely related to token bucket) is an algorithm that provides a simple, intuitive approach to rate limiting via a queue which you can think of as a bucket holding the requests.

Also known as a first in first out (FIFO) queue. If the queue is full, then additional requests are discarded (or leaked).



**Pros:**

* smooths out bursts of requests and processes them at an approximately average rate
* easy to implement on a single server or load balancer
* memory efficient for each user given the limited queue size

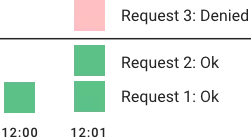
**Cons:**

* a burst of traffic can fill up the queue with old requests and starve more recent requests from being processed
* no guarantee that requests get processed in a fixed amount of time
* challenges of distributed environments: must use a policy to coordinate and enforce the limit between servers, if load balancing is used

**Fixed Window**

In a fixed window algorithm, a window size of n seconds (typically using human-friendly values, such as 60 or 3600 seconds) is used to track the rate. Each incoming request increments the counter for the window. If the counter exceeds a threshold, the request is discarded. The windows are typically defined by the floor of the current timestamp. *e.g. 12:00:03 with a 60 second window length, would be in the 12:00:00 window.*

In this algorithm, the time window is considered from the start of the time-unit to the end of the time-unit. For example, a period would be considered as 0-60 seconds for a minute irrespective of the time frame at which the API request has been made.



**Pros:**

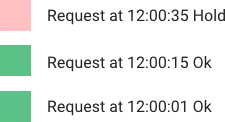
* ensures more recent requests gets processed without being starved by old requests

**Cons:**

* (boundary conditions) a single burst of traffic that occurs near the boundary of a window can result in twice the rate of requests being processed, because it will allow requests for both the current and next windows within a short time
* if many consumers wait for a reset window, for example at the top of the hour, then they may stampede your API at the same time.

**Sliding (Rolling) Log**

Sliding Log rate limiting involves tracking a time stamped log for each consumer’s request. These logs are usually stored in a hash set or table that is sorted by time. Logs with timestamps beyond a threshold are discarded. When a new request comes in, we calculate the sum of logs to determine the request rate. If the request would exceed the threshold rate, then it is held.



**Pros:**

* does not suffer from the boundary conditions of fixed windows. The rate limit will be enforced precisely
* sliding log is tracked for each consumer, you don’t have the stampede effect that challenges fixed windows

**Cons:**

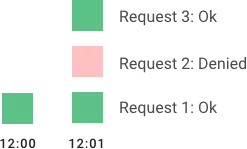
* very expensive to store an unlimited number of logs for every request
* also, expensive to compute because each request requires calculating a summation over the consumer’s prior requests, potentially across a cluster of servers

**Sliding (Rolling) Window\***

Hybrid approach that combines the low processing cost of the fixed window algorithm, and the improved boundary conditions of the sliding log.

Like the fixed window algorithm, we track a counter for each fixed window. Next, we account for a weighted value of the previous window’s request rate based on the current timestamp to smooth out bursts of traffic.

For example, if the current window is 25% through, then we weight the previous window’s count by 75%. The relatively small number of data points needed to track per key allows us to scale and distribute across large clusters.



**Pros:**

* flexibility to scale rate limiting with good performance
* avoids the starvation problem of leaky bucket
* avoid the boundary bursting problems of fixed window implementations

**Cons:**

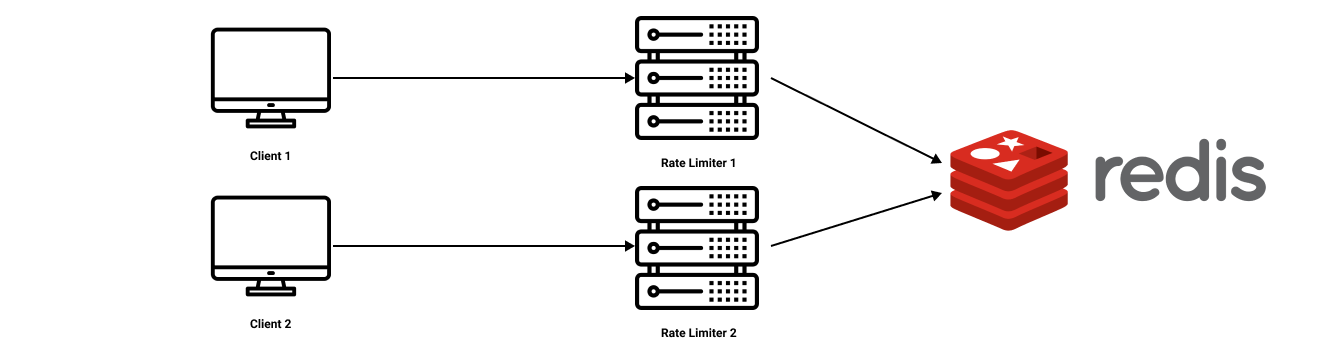
* cannot enforce hard throttling (only soft throttling)

**Why Redis as Data Storage?**

As we need some kind of database to store a list of timestamps grouped per access token. Maybe we could use MySQL or PostgreSQL? Yes, we could but then we would need a system that periodically removes outdated timestamps since neither MySQL or PostgreSQL allow us to set a time to life on a row. What about Memcached? Yes, that's also an option. Sadly, Memcached doesn't have the concept of an array or list (we could serialize an array using our favourite programming language). What about Redis? Redis is a key/value store that supports lists, sets, sorted sets and more. Thus, Redis seems to be better choice over the others.

Building a rate limiter with Redis is easy because of two commands INCR and EXPIRE. The basic concept is that you want to limit requests to a particular service in a given time period. Let’s say we have a service that has users identified by an API key. This service states that it is limited to 20 requests in any given minute.

To achieve this, we want to create a Redis key for every minute per API key. To make sure we don’t fill up our entire database with junk, expire that key after one minute as well.



**Pros:**

* More flexible for load-balancing rule

**Cons:**

* increased latency making requests to the data store,
* race conditions

**Race Condition:**

One of the largest problems with a centralized data store is the potential for race conditions in high concurrency request patterns.

This happens when you use a naïve “get-then-set” approach, wherein you retrieve the current rate limit counter, increment it, and then push it back to the datastore.

One way to avoid this problem is to put a “lock” around the key in question, preventing any other processes from accessing or writing to the counter. This would quickly become a major performance bottleneck, and does not scale well, particularly when using remote servers like Redis as the backing datastore.

A better approach is to use a “**set-then-get**” mindset, relying on atomic operators that implement locks in a very performant fashion, allowing you to quickly increment and check counter values without letting the atomic operations get in the way.