**Rate Limiting in Distributed Systems**

**Rate limiting** provides a mechanism to limit the number of requests to an API or service in a given time period, it protects APIs from malicious overuse by limiting how often each user can call the API.

Chart, bar chart

Description automatically generated

Chart, bar chart, histogram

Description automatically generated

**System Design Approach**

We can use a centralized data store **Redis** to allow more flexible load-balancing. A centralized data store will collect the counts for each window and consumer.

**Redis for Data Storage:** Building a rate limiter with Redis is easy because of two commands [INCR](https://redis.io/commands/incr) and [EXPIRE](https://redis.io/commands/expire). The basic concept is that we 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 of using Redis:**

More flexible for load-balancing rule

**Cons of using Redis:**

* increased latency making requests to the data store,
* race conditionsGraphical user interface, diagram

  Description automatically generated with medium confidence

### **Solution for Cons:**

One of the largest problems with a centralized data store is the potential for [race conditions](https://en.wikipedia.org/wiki/Race_condition) in [high concurrency](https://en.wikipedia.org/wiki/Concurrency_%28computer_science%29) 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.

Diagram

Description automatically generated

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.

### **Optimizing the Performance**

Increased [latency](https://konghq.com/blog/observability-kubernetes-kong/) is a disadvantage of using a centralized data store when checking the rate limit counters. Unfortunately, even checking a fast data store like Redis would result in milliseconds of additional latency for every request.

We can checks locally in memory to make these rate limit determinations with minimal latency. To make local checks, relax the rate check conditions and use an eventually consistent model. For example, each node can create a data sync cycle that will synchronize with the centralized data store. Each node periodically pushes a counter increment for each consumer and window to the datastore. These pushes atomically update the values. The node can then retrieve the updated values to update its in-memory version.

This cycle of converge → diverge → reconverge among nodes in the cluster is eventually consistent.

Diagram

Description automatically generated

The periodic rate at which nodes converge should be configurable. Shorter sync intervals will result in less divergence of data points when spreading traffic across multiple nodes in the cluster (e.g., when sitting behind a round robin balancer). Whereas longer sync intervals put less read/write pressure on the datastore and less overhead on each node to fetch new synced values.