**Rate Limiting in Distributed Systems**

We can use a centralized data store such as **Redis or**[**Cassandra**](https://konghq.com/blog/using-instaclustr-and-cassandra-with-kong/) to allow more flexible load-balancing. A centralized data store will collect the counts for each window and consumer.

The two main problems with this approach are increased latency making requests to the data store and race conditions,

Graphical user interface, diagram

Description automatically generated with medium confidence

### **Race Conditions**

One of the most extensive 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_(computer_science)) request patterns. This issue happens when we use a naïve “get-then-set” approach, wherein we retrieve the current rate limit counter, increment it, and then push it back to the datastore. This model’s problem is that additional requests can come through in the time it takes to perform a full cycle of read-increment-store, each attempting to store the increment counter with an invalid (lower) counter value. This allows a consumer to send an extremely high rate of requests to bypass rate limiting controls.

Diagram

Description automatically generated

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. A lock would quickly become a significant performance bottleneck and does not scale well, mainly 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 us to quickly increment and check counter values without letting the atomic operations get in the way.

### **Optimizing for Performance**

The increased [latency](https://konghq.com/blog/observability-kubernetes-kong/) is another 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.

Make 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.