Cortex - Global rate limiter

Author: Marco Pracucci
Date: October 2019
Status: *Approved*

Problem

Cortex currently have few global limits which are actually implemented as per-process limits:

- **Distributor**
  - `ingestion_rate`
    Per-user ingestion rate limit in samples per second
- **Ingester**
  - `max_series_per_user`
    Maximum number of active series per user
  - `max_series_per_metric`
    Maximum number of active series per metric name

Being these limits implemented per-process and due to the lack of any centralized coordination, the actual limit depends on the number of running distributors / ingestors and how well traffic is balanced between them.

This doc outlines a proposal to introduce global limits.

Proposal for `max_series_per(user|metric)`

A single time series is hashed to determine to which ingesters it should be sent to. Cortex currently supports two hashing functions:

- tenant + metric name (default)
- tenant + metric name + labels (if `-distributor.shard-by-all-labels` is enabled)

Case: shard-by-all-labels is enabled

Assuming an even distribution, multiple series with the same metric name are evenly spread across all ingesters and each series is stored in a number of ingesters equal to the replication factor.

Thus, the unique count of active series per metric name across all ingesters is approximately:
`(active series on 1 ingester / replication factor) * total number of ingesters`

Given we don't need an accurate count (ie. when the ingesters topology changes), we can use a **per-ingester** "max series per metric" limit equal to:
`(max global limit / number of ingesters) * replication factor`

The number of active ingesters is picked from the ring.

The same concept can be applied to **`max_series_per_user`**.

Case: shard-by-all-labels is disabled (default)

max_series_per_metric
Given a metric is always pushed to the same set of ingesters (based on the replication factor), we can configure the per-ingester local limit equal to the global limit.

max_series_per_user
Metrics series may not be evenly distributed across all cluster ingesters, because an high cardinality metric can lead to an higher number of active series on the ingesters where it's replicated, than the other ingesters in the cluster.

Some feedback is welcome on this case. Some options:

- Option #1: do not allow to set the global limit if `shard-by-all-labels` is disabled (check done by config validation)
- ~~Option #2: set the per-ingester limit the same way we do when `shard-by-all-labels` is enabled (depends on use cases: the higher the number of metrics, the lower the impact of a single high cardinality metric)~~

Proposal for `ingestion_rate`

The proposed solution is to introduce a "global" rate limiter as an alternative to the current "local" one (default). The two rate limiters are mutually exclusive.

The global rate limiter is based on the **local rate limiter** (`golang.org/x/time/rate`), but configured on each distributor with the following settings:

- **Local limit**: `global limit / number of active distributors`
- **Local burst**: at least `max samples in a request`

Introduce a way to have the count of active distributors in the cluster

- Option 1 (general purpose):
  An approach could be keep a general purpose pool of active distributors in the Cortex KVStore (Consul / Etcd / Gossip). Each distributor keeps itself updated in the pool,

updating a per-distributor keepalive timestamp every X period (ie. 20s) and unregistering itself on graceful shutdown. At any given time, the number of active distributors is the count of distributors registered in the pool whose keepalive timestamp is recent than `X + delta` (ie. `X * 3`).
No watching on the KVStore required. The number of distributors will be counted each time a distributor updates its heartbeat timestamp.

- ~~Option 2 (Kubernetes only):~~
  ~~Support Cortex clusters running on K8S only and implement a service discovery based on K8S API. Keep the interface generic, so that in the future we can plug other service discoveries if required.~~

Local burst should be at least the max samples in a request, otherwise any request with more samples than the burst will be rejected, no matter what.

Other related improvements

- Remove `limiter_reload_period` config option re-initializing the local limiter only once limit change (right now it's re-initialized every reload period)

Evaluated alternatives for `ingestion_rate`

Alternative: memcached

Introduce a global scalable rate limiter.

Goals

- No latency impact
- Keep backward compatibility
- Preferably no new external dependency

Global rate limiter

The proposed solution is to introduce a "global" rate limiter as an alternative to the current "local" one (default). The two rate limiters are mutually exclusive.

The rate limiter is based on a sliding time window algorithm, implemented using two fixed time windows - and approximating a linear distribution of requests over time - storing counters in **memcached** (no new external dependency). The idea comes from this CloudFlare blog post.

The global rate limiter is configured by:

- **Limit**: X samples per sec
  This is the average max samples per sec allowed over the observed time span
- **Time span**: Y seconds
  This is the time window of the bucket where the per-tenant ingested samples are counted, thus the burst allowed is `limit * time span`

Being the rate limiter generic, in the following we'll talk about "tokens" instead of "samples" (for Cortex) or "bytes" (for Loki), as the unit of the limited resource.

At any given point in time, for each tenant, there are two counters stored in memcached:

- Tokens count for the previous time bucket
- Tokens count for the current time bucket

The actual rate is calculated as slide window over the two buckets, approximating a linear distribution of requests in the previous time span (bucket):

```
currBucketAgeSecs = now - currBucketStartTime
prevBucketAgeSecs = timeSpanSecs - currBucketAgeSecs
currRate = ((prevTokens * (prevBucketAgeSecs / timeSpanSecs)) +
currBucketTokens) / timeSpanSecs
```

The rate limiter triggers if the `currRate > limit`. Once triggered, we can calculate for how long will be limited without adding further tokens to the current bucket with the formula:

```
decayRate = prevBucketRate = prevTokens / timeSpanSecs
exceedingRate = currRate - limit

if currTokens >= limit * timeSpanSecs {
    limitForSecs = timeSpanSecs - currBucketAgeSecs
} else {
    limitForSecs = min(prevBucketAgeSecs, exceedingRate/decayRate)
}
```

Implementation proposal #1 - With shared blacklist

The core idea to have a rate limiter - without adding any extra latency to check if a tenant hit the limit - is based on:

- Asynchronously increase memcached counters
- Store a shared blacklist on memcached. The entire blacklist (all blacklisted tenants) is serialized and stored as 1 single key in memcached. Distributors polls every 200ms the blacklist to see if it has changed.

How it works:

1. A new Push request arrives to the distributor
2. If the tenant is in the blacklist, the request is rejected
3. Otherwise process the request and push an async job to update the current bucket tokens counter:
    a. Limited pool of executors
    b. If the stack (LIFO) grows due to memcached high latency / unavailable, discard entries related to buckets older than the previous bucket
4. The rate limiter polls the blacklist stored on memcached and keep a local copy. The blacklist contains the list of blocked tenants and - for each tenant - until which timestamp should be blocked:
    a. When the async job (used to update counters) hits the rate limit, it calculates how long the tenant should be blocked and adds it to the blacklist (so that gets propagated to other distributors within the polling period 200ms). The blacklist is updated with a CompareAndSwap operation, and retried in case of race conditions.
    b. Cleanup: whenever the blacklist is updated, old expired entries are removed

Implementation proposal #2 - Local blacklist

The core idea is to avoid having a shared blacklist stored on memcached, thus:

- Asynchronously increase memcached counters
- Keep a local blacklist on each distributor

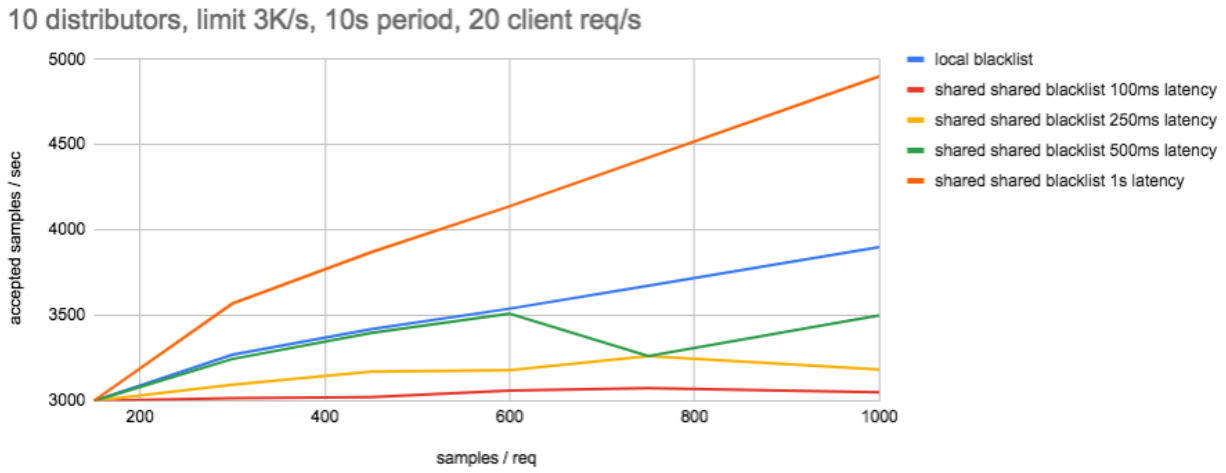
How it works:

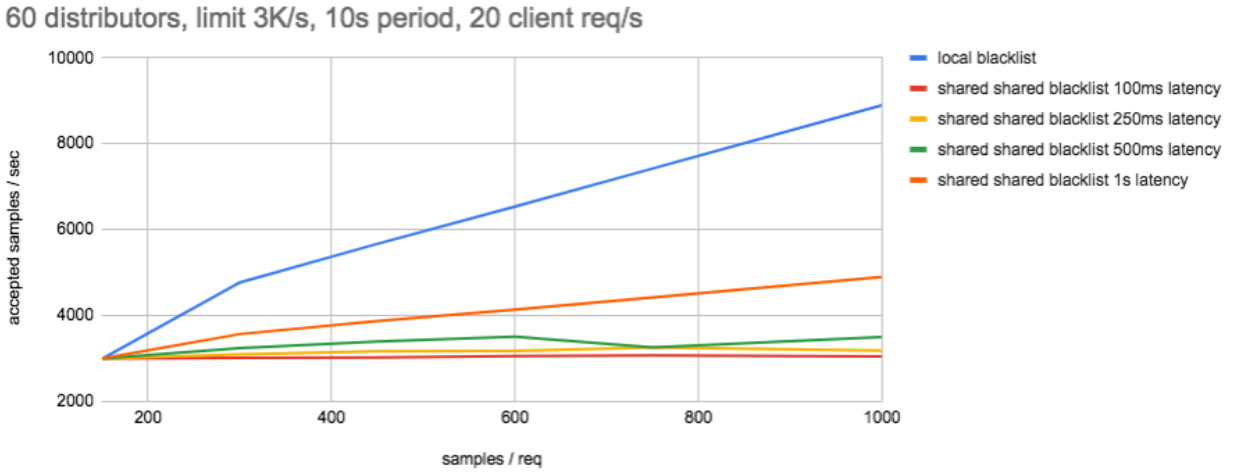
1. A new Push request arrives to the distributor
2. If the tenant is in the local blacklist, the request is rejected
3. Otherwise process the request and push an async job to update the current bucket tokens counter:
    a. Once counters on memcached are updated, it checks if the rate limit has been hit. If so, it calculates how long the tenant should be blocked and adds it to a local blacklist

Comparison between implementation proposal #1 and #2

I've run a simulation to compare both approaches:

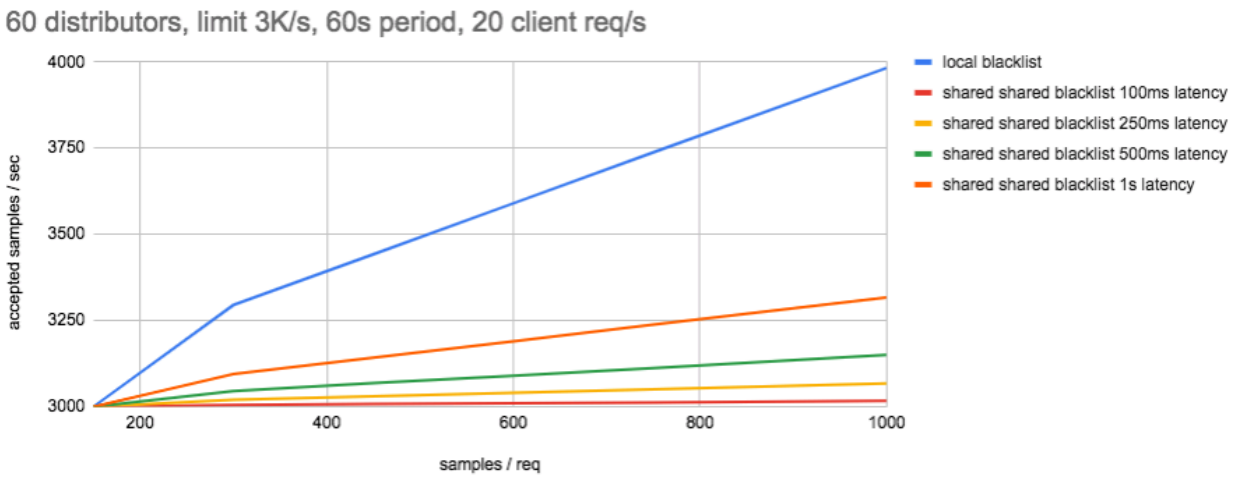
- Simulation source code
- Result data





With a short time span period (10s):
The local blacklist looks getting results close to the enforced limit when the number of number of distributors is low (*in a cluster with 1 distributor would be equal to the shared blacklist without latency*) or the samples per request is low.

The shared blacklist looks getting results close to the enforced limit when the latency is reasonably low.



With a longer time span period (60s):
A longer time span significantly mitigates the downside of the local blacklist and makes it appealing for our purpose. The downside is that the larger the time window is, the higher the burst we allow in a short time span (ie. 3K limit / sec over 60s means we accept 80K samples over 1 single second before blacklisting).

Failure modes / side effects

- If memcached is unavailable, bypass the rate limiter (all requests are considered legit).
- If memcached node is lost / replaced / restarted the rate limiter counters and blacklist reset, which should be tolerable given the buckets short time span and the expected frequency of a memcached node data loss
- In the proposed implementation there may be race conditions due to the async counters update (and the lack of any distributed locking) which may lead to allow more tokens than the limit. The over provisioning will be compensated with a lower actual limit in the next time span (being the current rate calculated on the previous + current bucket).

Exported metrics

Introduce metrics exported by the global rate limiter:

- **`cortex_distributor_ratelimiter_requests_total`**`{user="tenant-id",status="allowed|blocked"}`
  Counter - Total number of requests sent to the rate limiter, splitted by status
- **`cortex_distributor_ratelimiter_counters_update_duration_seconds`**
  Histogram - Time spent updating counters on memcached
- **`cortex_distributor_ratelimiter_blacklist_update_duration_seconds`**
  Histogram - Time spent updating blacklist on memcached

Implement the limiter so that the namespace is configurable to both support "cortex" and "loki".

> *Ed says: It becomes important when we write alerts, it's also confusing to Loki users who have no idea what Cortex is, we see this in slack, people who are using Loki but have never heard of Cortex. I think it does help to keep them configurable*

Other related improvements

- Remove `limiter_reload_period` config option and use a reasonable internal hard-coded value, re-initializing the local limiter only once limit values have changed (right now it's re-initialized every reload period)

Alternative: doorman (https://github.com/youtube/doorman)

Doorman is a distributed **client-side** rate limiter.

It offers an API to have a client requesting some capacity out of a total capacity available for a limited resource (ie. ingested samples per sec). The capacity for a resource is granted upon request and the client keeps holding the granted capacity until it's not explicitly released by the client itself. While the capacity is held by the client, the client periodically refresh the granted capacity: this is a mechanism used by doorman to rebalance the distributed capacity among requesters.

Let's consider a simple scenario:

- 3K samples / sec limit (free tier)
- 30 distributors in the cluster
- 1 push request / sec containing 1K samples

Workflow (T is time in seconds):

- T1: push request hits distributor #1, which requests capacity=1K and gets capacity=1K
- T2: push request hits distributor #2, which requests capacity=1K and gets capacity=1K
- T3: push request hits distributor #3, which requests capacity=1K and gets capacity=1K
- T4: push request hits distributor #4, which requests capacity=1K. Doorman has already given out all capacity, so returns capacity=0 and the request is rejected even if the actual rate is 1K / sec
    - The reason why it returns capacity=0 is because doorman doesn't push capacity changes back to all distributors to rebalance the capacity, but it will be rebalanced once the capacity grant of each distributor hits the refresh period (configurable, min 5s)
- T5: *nothing happen*

- T6: the refresh period of the distributor #1 expires, so it sends a new get capacity request to doorman (request 1K) and doorman grants only capacity=750 (4K total capacity asked so far divided by 4 distributors)
- T7: *capacity refreshed for distributor #2*
- T8: *capacity refreshed for distributor #3*
- T9: finally the capacity is refreshed for distributor #4 as well, which gets capacity=750 like all other ones. Yuppi! Unfortunately they all got less than 1000, so any subsequent request hitting these distributors will be actually rejected

The problem is that distributor #1, #2, #3 are still holding the granted capacity even if they're not receiving traffic anymore. An option may be releasing capacity very fast (ie. after 1s of idle).

The example above shows the workflow as being synchronous, but we want a zero-latency implementation so we would do something like:

1. Distributor has not asked any capacity yet (or the previously asked capacity has already been released)
2. Distributor asks some capacity, without synchronously waiting the response
3. Distributor allows the push request to come in, because doesn't known how much capacity has been granted to it
4. Distributor receives some capacity, which may be "0" or more: this capacity will effectively apply in a subsequent request to the same distributor, which in our example will never occur because we'll release the capacity earlier (ie. after 1s of idle), making this rate limiter implementation useless in this scenario

Other drawbacks of doorman:

- Last commit 3 years ago (*looks unmaintained, but has got 1.5K stars on GitHub*)
- Limits configuration is done through its own yaml file - to integrate with cortex, we may vendor it into cortex repo, create a new component which reads cortex config and run doorman server in Go. Alternatively, we would have to force Cortex users to configure defaults and per-tenant limits in doorman config file instead of Cortex config file.

Alternative: redis-based rate limiter

Pros:

- Server-side LUA scripting allows to implement a token bucket rate limiter without race conditions within 1 network round trip

Cons:

- New external dependency (a Redis cluster)