How Pingora keeps count

2023-05-12

Yuchen Wu

6 min read

A while ago we shared how we replaced NGINX with our in-house proxy, Pingora. We promised to share more technical details as well as our open sourcing plan. This blog post will be the first of a series that shares both the code libraries that power Pingora and the ideas behind them.

Today, we take a look at one of Pingora's libraries: pingora-limits.

pingora-limits provides the functionality to count inflight events and estimate the rate of events over time. These functions are commonly used to protect infrastructure and services from being overwhelmed by certain types of malicious or misbehaving requests.

For example, when an origin server becomes slow or unresponsive, requests will accumulate on our servers, which adds pressure on both our servers and our customers' servers. With this library, we are able to identify which origins have issues, so that action can be taken without affecting other traffic.

The problem can be abstracted in a very simple way. The input is a (never ending) stream of different types of events. At any point, the system should be able to tell the number of appearances (or the rate) of a certain type of event.

In a simple example, colors are used as the type of event. The following is one possible example of a sequence of events:

```
red, blue, red, orange, green, brown, red, blue,...
```

In this example, the system should report that "red" appears three times.

The corresponding algorithms are straightforward to design. One obvious answer is to use a hash table, where the keys are the colors and the values are their corresponding appearances. Whenever a new event appears, the algorithm looks up the hash table and increases the appearance counter. It is not hard to tell that this algorithm's time complexity is O(1) (per event) and the space complexity O(n) where n is the number of the types of events.

How Pingora does it

The hash table solution is fine in common scenarios, but we believe there are a few things that can be improved.

- We observe traffic to millions of different servers when the misbehaving ones are only a few at a given time. It seems a bit wasteful to require space (memory) that holds the counter for all the keys.

- Concurrently updating the hash table (especially when adding new keys) requires a lock. This behavior potentially forces all concurrent event processing to go through our system serialized. In other words, when lock contention is severe, the lock slows down the system.

The motivation to improve the above algorithm is even stronger considering such algorithms need to be deployed at scale. This algorithm operates on tens of thousands of machines. It handles more than twenty million requests per second. The benefits of efficiency improvement can be significant.

pingora-limits adopts a different approach: count–min sketch (CM sketch) estimation. CM sketch estimates the counts of events in O(1) (per event) but only using O(log(n)) of space (polylogarithmic, to be precise, more details here). Because of the simplicity of this algorithm, which we will discuss in a bit, it can be implemented without locks. Therefore, pingora-limits runs much faster and more efficiently compared to the hash table approach discussed earlier.

CM sketch

The idea of a CM sketch is similar to a Bloom filter. The mathematical details of the CM sketch can be found in this paper. In this section, we will just illustrate how it works.

A CM sketch data structure takes two parameters, H: number of hashes (rows) and N number of counters (columns) per hash (row). The rows and columns form a matrix. The space they take is H*N. Each row has its own independent hash function (hash_i()).

For this example, we use H=3 and N=4:

0	0
0	0
0	0

When an event, "red", arrives, it is counted by every row independently. Each row will use its own hashing function (hash_i(“red”)) to choose a column. The counter of the column is increased without **worrying about collisions** (see the end of this section).

The table below illustrates a possible state of the matrix after a single “red” event:

0	1
0	0
1	0

Then, let’s assume the event "blue" arrives, and we assume it collides with "red" at row 2: both hash to the third slot:

1	1
0	0
1	0

Let’s say after another series of events, “blue, red, red, red, blue, red”, So far the algorithm observed 5 “red”s and 3 “blue”s in total. Following the algorithm, the estimator eventually becomes:

3	5
0	0
5	0

Now, let’s see how the matrix reports the occurrence of each event. In order to retrieve the count of keys, the estimator just returns the minimal value of all the columns to which that key belongs. So the count of red is min(5, 8, 5) = 5 and blue is min(3, 8, 3) = 3.

This algorithm chooses the cells with the least collisions (via the min() operations). Therefore, collisions between events in single cells are acceptable because as long as there are collision free cells for a given type of event, the counting for that event is accurate.

The estimator can overestimate when two (or more) keys collide on all slots. Assuming there are only two keys, the probability of their total collision is 1/N^H (1/64 in this example). On the other hand, it never underestimates because it never loses count of any events.

Practical implementation

Because the algorithm only requires hashing, array index and counter increment, it can be implemented in a few lines of code and lock-free.

The following is a code snippet of how it is implemented in Rust.

```rust
pub struct Estimator {
    estimator: Box<[(Box<[AtomicIsize]>, RandomState)]>,
}

impl Estimator {
    /// Increment `key` by the value given. Return the new estimated value as a
result.
    pub fn incr<T: Hash>(&self, key: T, value: isize) -> isize {
        let mut min = isize::MAX;
        for (slot, hasher) in self.estimator.iter() {
            let hash = hash(&key, hasher) as usize;
            let counter = &slot[hash % slot.len()];
            let current = counter.fetch_add(value, Ordering::Relaxed);
            min = std::cmp::min(min, current + value);
        }
        min
    }
}
```

Performance

We compare the design above with the two hash table based approaches.

1. naive: Mutex<HashMap<u32, usize>>. This approach references the simple hash table approach mentioned above. This design requires a lock on every operation.

2. optimized: DashMap<u32, AtomicUsize>. DashMap leverages multiple hash tables in order to shard the keys to reduce contentions across different keys. We also use atomic counters here so that counting existing keys won't need a write lock.

We have two test cases, one that is single threaded and another that is multi-threaded. In both cases, we have one million keys. We generate 100 million events from the keys. The keys are uniformly distributed among the events.

The results below are performed on Debian VM running on M1 MacBook Pro.

SpeedPer event (the incr() function above) timing, lower is better:

	pingora-limits	naive	optimized
Single thread	10ns	51ns	43ns
Eight threads	212ns	1505ns	212ns

In the single thread case, where there is no lock contention, our approach is 5x faster than the naive one and 4x faster than the optimized one. With multiple threads, there is a high amount of contention. Our approach is similar to the optimized version. Both are 7x faster than the naive one. The reason the performance of pingora-limits and the optimized hash table are similar is because in both approaches the hot path is just updating the atomic counter.

Memory consumption Lower is better. The numbers are collected only from the single threaded test cases for simplicity.

	peak memory bytes	total allocations	total allocated bytes
pingora-limits	26,184	9	26,184

	peak memory bytes	total allocations	total allocated bytes
naive	53,477,392	20	71,303,260
optimized	36,211,208	491	71,307,722

Pingora-limits at peak requires 1/2000 of the memory compared to the naive one and 1/1300 of the memory of the optimized one.

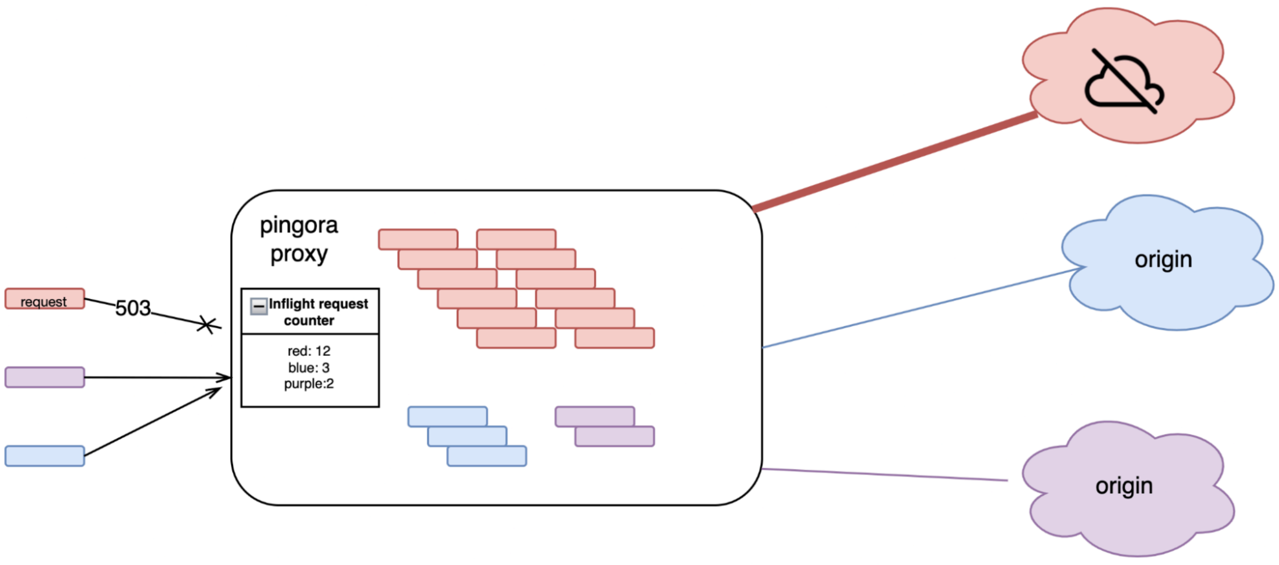
From the data above, pingora-limits is both CPU and memory efficient.

The estimator provided by Pingora-limits is a biased estimator because it is possible for it to overestimate the appearance of events.

In the case of accurate counting, where false positives are absolutely unacceptable, pingora-limits can still be very useful. It can work as a first stage filter where only the events beyond a certain threshold are fed to a hash table to perform accurate counting. In this case, the majority of low frequency event types are filtered out by the filter so that the hash table also consumes little memory without losing any accuracy.

How it is used in production

In production, pingora uses this library in a few places. The most common one is the connection limit feature. When our servers try to establish too many connections to a single origin server, in order to protect the server and our infrastructure from becoming overloaded, this feature will start rejecting new requests with 503 errors.



In this feature every incoming request increases a counter, shared by all other requests with the same customer ID, server IP and the server hostname. When the request finishes, the counter decreases accordingly. If the value of the counter is beyond a certain threshold, the request is rejected with a 503 error response. In our production environment we choose the parameters of the library so that a theoretical collision chance between two unrelated customers is about 1 / 2 ^ 52. Additionally, the rejection threshold is significantly higher than what a healthy customer's traffic would reach. Therefore, even if multiple customers' counters collide, it is not likely that the overestimated value would reach the threshold. So a false positive on the connection limit is not likely to happen.

Conclusion

Pingora-limits crate is available now on GitHub. Both the core functionality and the performance benchmark performed above can be found there.

In this blog post, we introduced pingora-limits, a library that counts events efficiently. We explained the core idea, which is based on a probabilistic data structure. We also showed through a performance benchmark that the pingora-limits implementation is fast and very efficient for memory consumption.

Not only that, but we will continue introducing and open sourcing Pingora components and libraries because we believe that sharing the idea behind the code is equally important as sharing the code itself.

Interested in joining us to help build a better Internet? Our engineering teams are hiring.

Discuss on X

Discuss on Hacker News

Discuss on Reddit

Performance Open Source Rust Pingora

Follow on X

Cloudflare | @cloudflare

RELATED POSTS

July 24, 2025 9:00 PM

Serverless Statusphere: a walk through building serverless ATProto applications on Cloudflare’s Developer Platform

Build and deploy real-time, decentralized Authenticated Transfer Protocol (ATProto) apps on Cloudflare Workers....

By Inanna Malick

Durable Objects, Wrangler, Rust

July 23, 2025 10:00 PM

Building Jetflow: a framework for flexible, performant data pipelines at Cloudflare

Faced with a data-ingestion challenge at a massive scale, Cloudflare's Business Intelligence team built a new framework called Jetflow....

By Harry Hough, Rebecca Walton-Jones , Andy Fan, Ricardo Margalhau, Uday Sharma

Data, Go, Performance, Design, Engineering

May 22, 2025 9:00 PM

Resolving a request smuggling vulnerability in Pingora

Cloudflare patched a vulnerability (CVE-2025-4366) in the Pingora OSS framework, which exposed users of the framework and Cloudflare CDN’s free tier to potential request smuggling attacks....

By Edward Wang, Andrew Hauck, Aki Shugaeva

Pingora, CDN, Security, CVE, Bug Bounty

May 20, 2025 9:00 PM

Performance measurements… and the people who love them

Developers have a gut-felt understanding for performance, but that intuition breaks down when systems reach Cloudflare's scale....

By Kevin Guthrie

Internet Performance, Latency, Open Source, Observability, TTFB