Dan Slimmon

SRE at Hashicorp

The Latency/Throughput Tradeoff: Why Fast Services Are Slow And Vice Versa

ON 2019/02/262019/03/25 / BY DAN SLIMMON / IN UNCATEGORIZED

Special thanks to the graceful and cunning Ben Ng for consulting on this post.

I'm finally getting around to reading that DevOps* book everybody's been raving about, *Site Reliability Engineering: How Google Runs Production Systems (https://landing.google.com/sre/sre-book/toc/index.html).* (https://landing.google.com/sre/sre-book/toc/index.html) My verdict so far: it's pretty good.

Here's one of the first passages to jump out to me, from *Chapter 3: Embracing Risk*:

The low-latency user wants Bigtable's request queues to be (almost always) empty so that the system can process each outstanding request immediately upon arrival. (Indeed, inefficient queuing is often a cause of high tail latency.) The user concerned with offline analysis is more interested in system throughput, so that user wants request queues to never be empty. To optimize for throughput, the Bigtable system should never need to idle while waiting for its next request.

This is a profound and general insight. When I read this passage, my last decade of abject suffering suddenly came into focus for me.

When I say "abject suffering," I'm of course talking about ElasticSearch administration. When a storage system like ElasticSearch has to serve both high-latency and high-throughput workloads, it is *guaranteed* to get ugly. This fact is super important, which is why I'm devoting this blog post to exploring the relationships among latency, throughput, and capacity from a queueing perspective. I hope I can make these relationships stick in your mind like they've stuck in mine.

* Go ahead. Tell me DevOps and SRE aren't the same thing. I dare you.

The tradeoff between throughput and latency

Consider a service that responds to requests. As an example, let's say it's a service that takes as input a picture of a dog and returns a picture of that dog wearing a silly hat.

Like almost any service (exception: Tourbillon (https://github.com/danslimmon/tourbillon)), our service can only handle a certan number of requests per second [to put hats on dogs (RPSTPHOD)]. We'll call this number its capacity. If we have 200 processes devoted to dog-hatting, and dogs take on average 400 milliseconds to haberdash, then the theoretical capacity of the system is

(200) / (0.4s) = 500 hats per second

Artist's rendering

Now let's consider the two types of users that depend on our service:

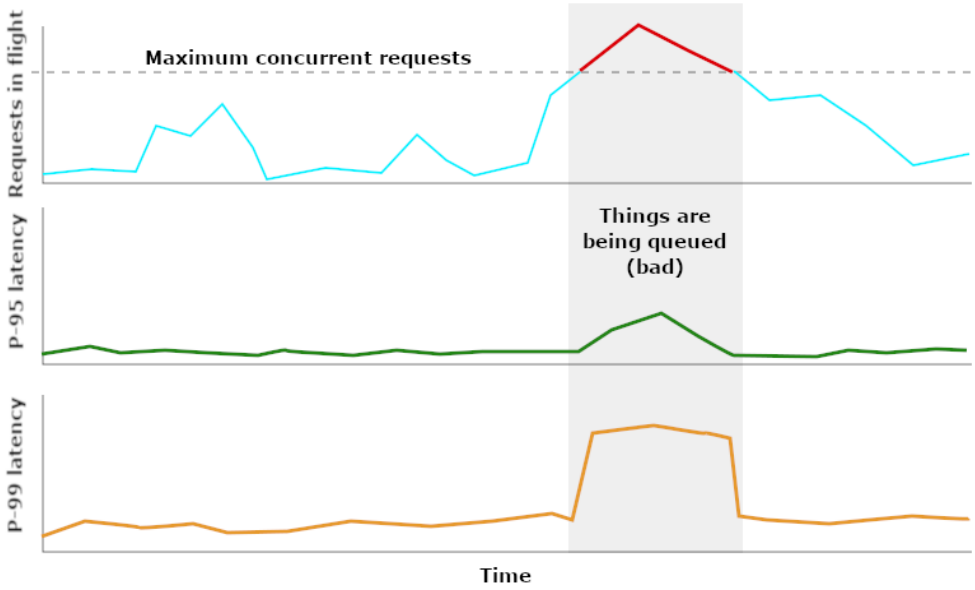
- **On-the-spot dog hatters.** At any given time, these users have a single dog picture that requires a hat as soon as possible. Perhaps they're using our service to support a website that generates a single dog-hat picture per page load, and they want their page to load quickly. These users are interested primarily in how quickly they can get a hat on a dog. In a word: latency.
- **Bulk dog-hatters.** These users tend to have massive data sets that they want processed as quickly as possible. The most obvious example would be a law enforcement agency wanting to compare their large database of pet photos to surveillance footage of a particular dog robbing a bank while wearing a hat. Bulk dog-hatters care not about the latency of any individual dog-hatting, but about the *throughput* they can achieve. In other words: how close they can get to our service's theoretical capacity of 500 hats per second.

But here's the problem: *no single cluster of dog-hatting servers can be*

optimal for both types of users. And the better we make the service for one kind of user, the worse we make it for the other.

The needs of on-the-spot users

In order to minimize latency for our on-the-spot users (without dropping any of their requests), we need to make sure that there's always a processor idle when their request comes in. If we fail to make sure of this, then new requests will have to be queued while we wait for a spot to open up, thus inflating latency. The system needs some "slack."

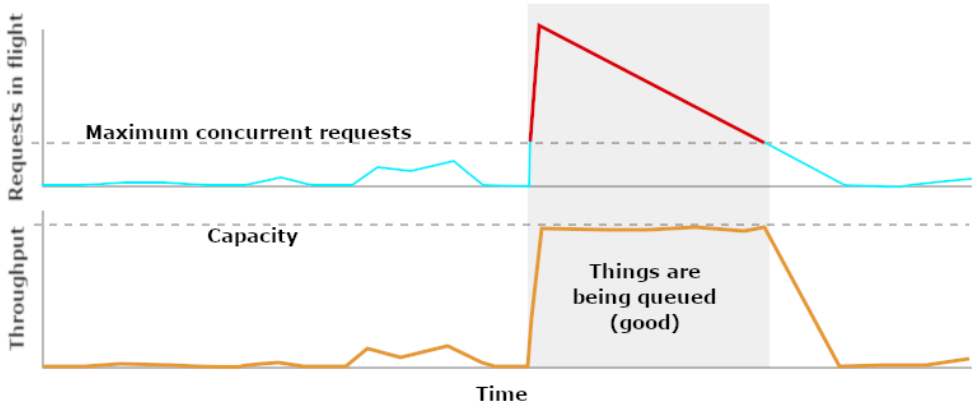


Since we need slack, we don't ever want throughput to approach capacity. The closer we get to our system's capacity, the more drastically latencies will balloon, like I talked about in this post (https://blog.danslimmon.com/2016/08/26/the-most-important-thing-to-understand-about-queues/).

The needs of bulk users

Our bulk dog-hatters, on the other hand, don't care so much about request latency. Some of their individual requests might take seconds, or minutes, or even *hours* to complete. What they care about is how quickly our service can process their entire data set. In other words, they care about getting throughput as close as possible to capacity.

This means that, whenever a job is running, bulk dog-hatters wantthere to be (virtually) zero slack. Every processor should be active at all times. Consequently, our queue sizes will explode as soon as the job starts, and our queues will stay occupied until the job is almost done.



In this case, we want our queues to be full whenever there's a bulk job running. Anything else would give sub-optimal throughput.

Splitting the cluster up

The needs of on-the-spot and bulk users are incompatible. One group needs minimal latency, while the other group needs maximal throughput.

If both of these groups are using the same cluster, we're going to have serious problems. On-the-spot users' latencies will vary widely depending on whether there's currently a bulk job in progress, and bulk users' job times will vary depending on the number of on-the-spot users currently using the system. No matter how much we scale or tweak tuning parameters, neither group will get what they need. And what's worse, we'll be stuck in a perpetual tug-of-war between the priorities of these two groups.

So let's split our cluster in two: a "low latency" cluster and a "high throughput" cluster. And let's let our users pick the right one for their use case. This way, we'll have much clearer expectations about the performance and scaling characteristics of our service, and we'll avoid the frustrating priority tug-of-war that characterized our mixed-use cluster.

The split doesn't have to be complete. Instead of having two wholly separate clusters, we could have some kind of load balancer that reserves a certain portion of our fleet for low-latency traffic and slots bulk jobs onto dedicated segments of the cluster. The details of every solution will vary. What matters is that on-the-spot and bulk dog-

hatters aren't drawing on the same pool of resources.

Once we do split up our cluster, then, what should we expect the performance characteristics of the new clusters to be? What will their graph dashboards look like when they're healthy, or near capacity, or over capacity? In an upcoming post, I'll use some more queueing reasoning to answer these questions. So get hype for that!

[UPDATE: It's here (https://blog.danslimmon.com/2019/03/25/latency-and-throughput-optimized-clusters-under-load/)!]

2 thoughts on "The Latency/Throughput Tradeoff: Why Fast Services Are Slow And Vice Versa"

1. Pingback: Latency- and Throughput-Optimized Clusters Under Load – Dan Slimmon

2. **Steven**
   Thank you for this blog post. As someone who doesn't have a lot of experience dealing with queues, it's a bit hard to understand what's mentioned in the book. Your explaination really helped. Thank you!

   2022/05/05 AT 14:21 ↪ **REPLY**