the morning paper

a random walk through Computer Science research, by Adrian Colyer

The Tail at Scale

JANUARY 15, 2015JULY 26, 2017

The Tail at Scale (http://www.cs.duke.edu/courses/cps296.4/fall13/838-CloudPapers/dean_longtail.pdf) – Dean and Barroso 2013

We've all become familiar with the importance of fault-tolerance and the techniques that can be used to achieve it. Less well-known is the idea of *tail-tolerance*. A system that doesn't respond quickly enough feels clunky to its users and can have serious negative consequences for site/service usability with consequent knock-on effects being felt on the bottom line. Google's target for a responsive system is 100ms, as relayed by Dean and Barroso.

> Just as fault-tolerant computing aims to create a reliable whole out of less-reliable parts, large online services need to create a predictably responsive whole out of less-predictable parts; we refer to such systems as "latency tail-tolerant," or simply "tail-tolerant."

There are lots of reasons why every now and then a request might take longer than expected (we are given eight examples in the paper). How good do we need to be in order to meet response targets? If the 99% percentile of response times falls within our target, is that good enough?

> consider a system where each server typically responds in 10ms but with a 99th percentile latency of one second. If a user request is handled on just one such server, one user request in 100 will be slow (one second).

Maybe that's ok. But watch what happens when there is not just a single server involved in servicing a request, but 100. For example, a fan-out query, or a large number of microservices contributing to the result:

> If a user request must collect responses from 100 such servers in parallel, then 63% of user requests will take more than one second. Even for services with only one in 10,000 requests experiencing more than one-second latencies at the single-server level, a service with 2,000 such servers will see almost one in five user requests taking more than one second.

Under this scenario, even if your 99.99% percentile response time is ok, 20% of your users see a poor response time!! It's easy to work out the sums for your own scenario: just take your acceptable latency percentile (e.g. 0.999) and raise it to the power of the number of servers involved in responding.

For a real Google system with 10ms latency at the 99% percentile for any single request, the 99% percentile for all requests to finish is 140ms, and the 95% percentile is 70ms.

> … meaning that waiting for the slowest 5% of the requests to complete is responsible for half of the total 99%-percentile latency. Techniques that concentrate on these slow outliers can yield dramatic reductions in overall service performance.

Hopefully now you're convinced that the tail really can wag the dog when it comes to response times. How can we design systems to tolerate tail latency and still be responsive overall?

Firstly, you can try to reduce response-time variability by prioritizing interactive requests, breaking down large units of work into smaller pieces to allow interleaving (aka reducing _head-of-line blocking), and carefully managing background activities. Somewhat counter-intuitively, it can be better to synchronize a background task to run on all machines at the same time, rather than spread it out over time. A few moments thinking about the implications of the tail will reveal why…

> This synchronization enforces a brief burst of activity on each machine simultaneously, slowing only those interactive requests being handled during the brief period of back-ground activity. In contrast, without synchronization, a few machines are always doing some background activity, pushing out the latency tail on all requests.

At the end of the day though, you're never going to be able to fully limit latency variability, so you need to consider design patterns that can *tolerate* it. Here are 7 patterns to enhance tail tolerance:

- 1) **Hedged requests**: send the same requests to multiple servers, and use whatever response comes back first. To avoid doubling or tripling your computation load though, don't send the hedging requests straight away:

  defer sending a secondary request until the first request has been outstanding for more than the 95th-percentile expected latency for this class of requests. This approach limits the additional load to approximately 5% while substantially shortening the tail latency.

- 2) **Tied requests**: instead of delaying before sending out hedged requests, enqueue requests simulateously on multiple servers, but tie them together but telling each server who else also has this in their queue. When the first server processes the request, it tells the others to cancel it from their queues. Since the cancellation messages traverse the network…

  … it is useful for the client to introduce a small delay of two times the average nework message delay (1ms or less in modern data-center networks) between sending the first request and sending the second request.

In a real Google system, this tied-request mechanism reduced median latency by 16%, and a 40% reduction at the 99.9% percentile.

- 3) **Micro-partition**: have many more partitions than servers to help combat imbalances.

  With an average of, say, 20 partitions per machine, the system can shed load in rougly 5% increments and in 1/20th the time it would take if the system simply had a one-to-one mapping of partitions to machines.

- 4) **Selectively increase replication factors**: increase the replication factor for partitions you detect or predict will be hot. Load-balancers can then help to spread the load.

  Google's main Web search system uses this approach, making additional copies of popular and important documents in multiple micro-partitions.

- 5) **Put slow machines on probation**. When a slow machine is detected temporarily exclude it from operations (circuit breaker). Since the source of slowness is often temporary, monitor when to bring the effected system back online again:

  continue to issue shadow requests to these excluded servers, collecting statistics on their latency so they can be re-incorporated into the service when the problem abates.

- 6) **Consider 'good enough' responses**. Once a sufficient fraction of all servers have responded, the user may be best served by being given slighly incomplete results in exchange for better end-to-end latency. See also the related concept of dimmer switches (https://blog.acolyer.org/2014/10/27/improving-cloud-service-resilience-using-brownout-aware-load-balancing/).

- 7) Use **canary requests**.

  Another problem that can occur in systems with very high fan-out is that a particular request exercises an untested code path, causing crashes or extremely long delays on thousands of servers simultaneously. To prevent such correlated crash scenarios, some of Google's IR systems employ a technique called "canary requests"; rather than initially send a request to thousands of leaf servers, a root server sends it first to one or two leaf servers. The remaining servers are only queried if the root gets a successful response from the canary in a reasonable period of time.

DISTRIBUTED SYSTEMS GOOGLE

14 thoughts on "The Tail at Scale"

1. Pingback: Dremel: interactive analysis of web-scale datasets | the morning paper
2. Pingback: Raft Refloated: Do we have consensus? | the morning paper
3. Pingback: Scaling Concurrent Log-Structured Data Stores | the morning paper
4. Pingback: Queues don't matter when you can JUMP them | the morning paper
5. Pingback: StreamScope: Continuous reliable distributed processing of big data streams | the morning paper
6. Pingback: IX: A protected dataplane operating system for high throughput and low latency | the morning paper
7. Pingback: Serverless computing: economic and architectural impact | the morning paper
8. Pingback: RobinHood: tail latency aware caching – dynamic reallocation from cache-rich to cache-poor | the morning paper
9. Pingback: RobinHood: tail latency aware caching – dynamic reallocation from cache-rich to cache-poor | Theresa Welchy
10. Pingback: RobinHood: tail latency aware caching – dynamic reallocation from cache-rich to cache-poor | Daily AI
11. Pingback: Reading Week #2 | | bill duncan's blog
12. Pingback: Memcached – Caching Beyond RAM: Riding the Cliff - The Peaceful Revolutionist
13. Pingback: Memcached – Caching Beyond RAM: Riding the Cliff - Entertaining WE
14. Pingback: An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems | the morning paper