Are You Load Balancing Wrong?

THOMAS A. LIMONCELLI

**ANYONE CAN USE
A LOAD BALANCER.
USING THEM
PROPERLY IS MUCH
MORE DIFFICULT.**

A reader contacted me recently to ask if it is better to use a load balancer to add capacity or to make a service more resilient to failure. The answer is: both are appropriate uses of a load balancer. The problem, however, is that most people who use load balancers are doing it wrong.

In today's web-centric, service-centric environments the use of load balancers is widespread. I assert, however, that most of the time they are used incorrectly. To understand the problem, we first need to discuss a little about load balancers in general. Then we can look at the problem and solutions.

A load balancer receives requests and distributes them to two or more machines. These machines are called replicas, as they provide the same service. For the sake of simplicity, assume these are HTTP requests from web browsers, but load balancers can also be used with HTTPS requests, DNS queries, SMTP (email) connections, and many other protocols. Most modern applications are engineered to work behind a load balancer.

LOAD BALANCING FOR CAPACITY VERSUS RESILIENCE

There are two primary ways to use load balancers: to increase capacity and to improve resiliency.

Using a load balancer to increase capacity is very simple. If one replica is not powerful enough to handle the entire incoming workload, a load balancer can be used to

distribute the workload among multiple replicas.

Suppose a single replica can handle 100 QPS (queries per second). As long as fewer than 100 QPS are arriving, it should run fine. If more than 100 QPS arrive, then the replica becomes overloaded, rejects requests, or crashes. None of these is a happy situation.

If there are two machines behind a load balancer configured as replicas, then the capacity is 200 QPS; three replicas would provide 300 QPS of capacity, and so on. As more capacity is needed, more replicas can be added. This is *horizontal scaling*.

Load balancers can also be used to improve resiliency. Resiliency means the ability to survive a failure. Individual machines fail, but the system should continue to provide service. All machines eventually fail—that's physics. Even if a replica had perfect uptime, you would still need resiliency mechanisms because of other externalities such as software upgrades or the need to physically move a machine.

A load balancer can be used to achieve resiliency by leaving enough spare capacity that a single replica can fail and the remaining replicas can handle the incoming requests.

Continuing the example, suppose four replicas have been deployed to achieve 400 QPS of capacity. If you are currently receiving 300 QPS, each replica will receive approximately 75 QPS (one-quarter of the workload). What will happen if a single replica fails? The load balancer will quickly see the outage and shift traffic such that each replica receives about 100 QPS. That means each replica is running at maximum capacity. That's cutting it close, but it is acceptable.

The difference between using a load balancer to increase capacity or improve resiliency is an operational difference, not a configuration difference.

What if the system had been receiving 400 QPS? Under normal operation, each of the four replicas would receive approximately 100 QPS. If a single replica died, however, the remaining replicas would receive approximately 133 QPS each. Since each replica can process about 100 QPS, this means each one of them is overloaded by a third. The system might slow to a crawl and become unusable. It might crash.

The determining factor in how the load balancer was used is whether the arriving workload was above or below 300 QPS. If 300 or fewer QPS were arriving, this would be a load balancer used for resiliency. If 301 or more QPS were arriving, this would be a load balancer for increased capacity.

The difference between using a load balancer to increase capacity or improve resiliency is an operational difference, not a configuration difference. Both use cases configure the hardware and network (or virtual hardware and virtual network) the same, and configure the load balancer with the same settings.

The term *N+1 redundancy* refers to a system that is configured such that if a single replica dies, enough capacity is left over in the remaining *N* replicas for the system to work properly. A system is *N+0* if there is no spare capacity. A system can also be designed to be *N+2* redundant, which would permit the system to survive two dead replicas, and so on.

THREE WAYS TO DO IT WRONG

Now that we understand the two different ways a load balancer can be used, let's examine how most teams fail.

Level 1: The team disagrees

Ask members of the team whether the load balancer is being used to add capacity or improve resiliency. If different people on the team give different answers, you're load balancing wrong.

If the team disagrees, then different members of the team will be making different engineering decisions. At best, this leads to confusion. At worst, it leads to suffering.

You would be surprised at how many teams are at this level.

Level 2: Capacity undefined

Another likely mistake is not agreeing how to measure the capacity of the system. Without this definition, you do not know if this system is $N+O$ or $N+1$. In other words, you might have agreement that the load balancing is for capacity or for resilience, but you do not know whether you are using it that way.

To know for sure, you have to know the actual capacity of each replica. In an ideal world, you would know how many QPS each replica can handle. The math to calculate the $N+1$ threshold (or high-water mark) would be simple arithmetic. Sadly, the world is not so simple.

You can't simply look at the source code, know how much time and resources each request will require, and determine the capacity of a replica. Even if you did know the theoretical capacity of a replica, you would need to verify it experimentally. We're scientists, not barbarians!

Capacity is best determined by benchmarks. Queries are generated and sent to the system at different rates, with the response times measured. Suppose you consider

a 200-ms response time to be sufficient. You can start by generating queries at 50 per second and slowly increase the rate until the system is overloaded and responds slower than 200 ms. The last QPS rate that resulted in sufficiently fast response times determines the capacity of the replica.

How do you quantify response time when measuring thousands or millions of queries? Not all queries run in the same amount of time. You can't take the average, as a single long-running request could result in a misleading statistic. Averages also obscure bimodal distributions. (For more on this, see chapter 17, Monitoring Architecture and Practice, of *The Practice of Cloud System Administration, Volume 2*, by Thomas Limoncelli, Strata R. Chalup, and Christina J. Hogan; Addison-Wesley, 2015.)

Since a simple average is insufficient, most sites use a percentile. For example, the requirement might be that the 90th percentile response time must be 200 ms or better. This is a very easy way to toss out the most extreme outliers. Many sites are starting to use MAD (median absolute deviation), which is explained in a 2015 paper by David Goldberg and Yinan Shan, The Importance of Features for Statistical Anomaly Detection (<https://www.usenix.org/system/files/conference/hotcloud15/hotcloud15-goldberg.pdf>).

Generating synthetic queries to use in such benchmarks is another challenge. Not all queries take the same amount of time. There are short and long requests. A replica that can handle 100 QPS might actually handle 80 long queries or 120 short queries. The benchmark must use a mix that reflects the real world.

If all queries are read-only or do not mutate the system, you can simply record an hour's worth of actual queries and replay them during the benchmark. At a previous employer, we had a data set of 11 billion search queries used for benchmarking our service. We would send the first 1 billion queries to the system to warm up the cache. We recorded measurements during the remaining queries to gauge performance.

Not all workloads are read-only. If a mixture of read and write queries is required, the benchmark data set and process are much more complex. It is important that the mixture of read and write queries reflects real-world scenarios.

Sadly, the mix of query types can change over time as a result of the introduction of new features or unanticipated changes in user-access patterns. A system that was capable of 200 QPS today may be rated at 50 QPS tomorrow when an old feature gains new popularity.

Software performance can change with every release. Each release should be benchmarked to verify that capacity assumptions haven't changed.

If this benchmarking is done manually, there's a good chance it will be done only on major releases or rarely. If the benchmarking is automated, then it can be integrated into your CI (continuous integration) system. It should fail any release that is significantly slower than the release running in production. Such automation improves engineering productivity not only because it eliminates the manual task, but also because you immediately know the exact change that caused the regression. If the benchmarks are only done occasionally, then finding

a performance regression involves hours or days of searching for which change caused the problem.

Ideally, the benchmarks are validated by also measuring live performance in production. The two statistics should match up. If they don't, you must true-up the benchmarks.

Another reason why benchmarks are so complicated is caches. Caches have unexpected side effects. For example, intuitively you would expect that a system should get faster as replicas are added. Many hands make light work. Some applications get slower with more replicas, however, because cache utilization goes down. If a replica has a local cache, it is more likely to have a cache hit if the replica is highly utilized.

Level 3: Definition but no monitoring

Another mistake a team is likely to make is to have all these definitions agreed upon, but no monitoring to detect whether or not you are in compliance.

Suppose the team has determined that the load balancer is for improving both capacity and resilience, they have defined an algorithm for measuring the capacity of a replica, and they have done the benchmarks to ascertain the capacity of each replica.

The next step is to monitor the system to determine whether the system is $N+1$ or whatever the desired state is.

The system should not only monitor the utilization and alert the operations team when the system is out of compliance, but should also alert the team when the system is nearing that state. Ideally, if it takes T minutes to add capacity, the system needs to send the alert at least T minutes before that capacity is needed.

Cloud-computing systems such as AWS (Amazon Web Services) have systems that can add more capacity on demand. If you run your own hardware, provisioning new capacity may take weeks or months. If adding capacity always requires a visit to the CFO to sign a purchase order, you are not living in the dynamic, fast-paced, high-tech world you think you are.

SUMMARY

Anyone can use a load balancer. Using it properly is much more difficult. Some questions to ask:

1. Is this load balancer used to increase capacity ($N+O$) or to improve resiliency ($N+1$)?
2. How do you measure the capacity of each replica? How do you create benchmark input? How do you process the benchmark results to arrive at the threshold between good and bad?
3. Are you monitoring whether you are compliant with your $N+M$ configuration? Are you alerting in a way that provides enough time to add capacity so that you stay compliant?

If the answer to any of these questions is “I don’t know” or “No,” then you’re doing it wrong.

Thomas A. Limoncelli is a site reliability engineer at Stack Overflow Inc. in New York City. His books include *The Practice of Cloud Administration* (<http://the-cloud-book.com>), *The Practice of System and Network Administration* (<http://the-sysadmin-book.com>), and *Time Management for System Administrators*. He blogs at EverythingSysadmin.com and

tweets at @YesThatTom. He holds a B.A. in computer science from Drew University.

Related Papers

The Tail at Scale

Jeffrey Dean, Luiz André Barroso

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

Communications of the ACM 56(2): 74-80

<http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/abstract>

Resilience Engineering: Learning to Embrace Failure

A discussion with Jesse Robbins, Kripa Krishnan, John Allspaw, and Tom Limoncelli

<http://queue.acm.org/detail.cfm?id=2371297>

The Pathologies of Big Data

Adam Jacobs

Scale up your datasets enough and all your apps will come undone. What are the typical problems and where do the bottlenecks generally surface?

<http://queue.acm.org/detail.cfm?id=1563874>

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.