Infrastructure

Deterministic Aperture: A distributed, load balancing algorithm

By
Bryce Anderson
and
Ruben Oanta
Thursday, 19 December 2019

In this blog post, we will walk through a new client-side load balancing technique we've developed and deployed widely at Twitter which has allowed our microservice architecture to efficiently scale clusters to thousands of instances. We call this new technique deterministic aperture and it's available within Finagle (https://github.com/twitter/finagle), Twitter's protocol-agnostic remote procedure call (RPC) framework.

Our microservice architecture comprises many different services with varying capacity requirements. As parts of the Twitter application grow, we can scale demands on capacity by adding more instances or replicas to a respective service cluster (i.e., horizontal scaling). In order to properly utilize all the replicas of a service cluster, Finagle embeds a client-side load balancer in every Finagle client. This allows us to operate with fewer layers of physical infrastructure. In modern serving systems, these load balancers are often referred to as application load balancers and they serve two primary functions:

- They allow callers of a backend service to safely utilize the aggregate capacity by dividing the work among backend replicas.
- Because all requests to a respective service now flow through a load balancer, it is well positioned to route around replicas when they inevitably fail.

So what do these load balancers balance over exactly? They balance over both sessions (OSI L5) and requests (OSI L7). We primarily measure the effectiveness of a load balancer implementation based on the resulting request distribution, since requests are often a more accurate proxy for resource utilization. However, since sessions are the vehicle for requests, the session distribution is an important consideration. Let's dive into both.

Request distribution using power of two choices

To balance requests, we use a distribution strategy called power of two choices (P2C). This is a two-step process:

1. For every request the balancer needs to route, it picks two unique instances at random.
2. Of those two, it selects the one with the least load.

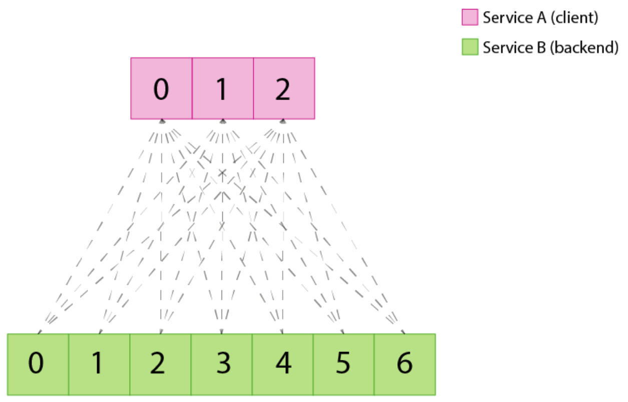
If you prefer seeing the algorithm in code, you can find Finagle's Scala implementation here (https://github.com/twitter/finagle/blob/finagle-19.6.0/finagle-core/src/main/scala/com/twitter/finagle/loadbalancer/p2c/P2CPick.scala#L46-L75). This strategy is based on the ideas behind Michael Mitzenmacher's paper, "The power of two choices in randomized load balancing." (https://ieeexplore.ieee.org/document/963420) The key result from Mitzenmacher's work is that comparing the load on two randomly selected instances converges on a load distribution that is exponentially better than random, all while reducing the amount of state (e.g., contentious data structures) a load balancer needs to manage.

What's more, because the P2C algorithm lets us break up the process of selection and evaluation into distinct phases, we can have sophisticated definitions of load. However, for the common use case, preferring the fewest number of outstanding requests in the local load balancer is both simple and effective. In practice, our implementation of P2C results in a uniform request distribution, so long as sessions are equally distributed.

So how do we balance sessions? The simplest approach is to use a mesh topology.

Session distribution using a mesh topology

A typical way to visualize a mesh service-to-service topology is like this:



Service A is a client of service B, and each instance of A has an embedded load balancer. In turn, each load balancer opens up a session to all replicas of service B, resulting in an even session distribution.

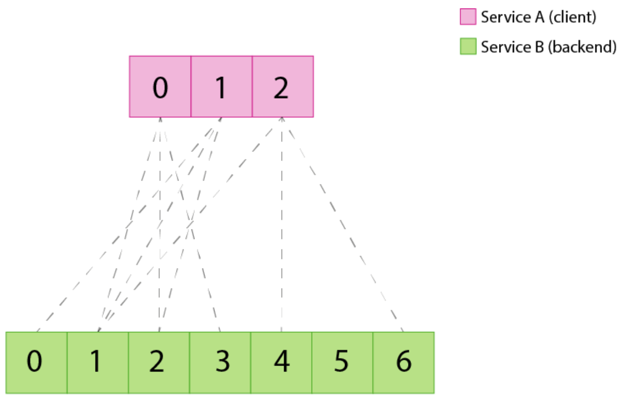
However, the full mesh topology has all sorts of negative consequences as we've scaled our services to thousands of instances. For starters, there's the inherent cost of a session such as allocating a socket, buffers, etc. On top of that, we need to ensure that balancers don't compromise success rate by using a stale or unhealthy session. To do this, we must implement relatively expensive health checks and circuit breakers per session, but these can be ineffective because each connection collects very little data. What makes matters worse is that because load balancers are distributed, each balancer has to independently discover the same failures — there's no isolation.

Given these issues, we set out to reduce the number of sessions that a load balancer manages without drastically changing our architecture or introducing expensive coordination between load balancers.

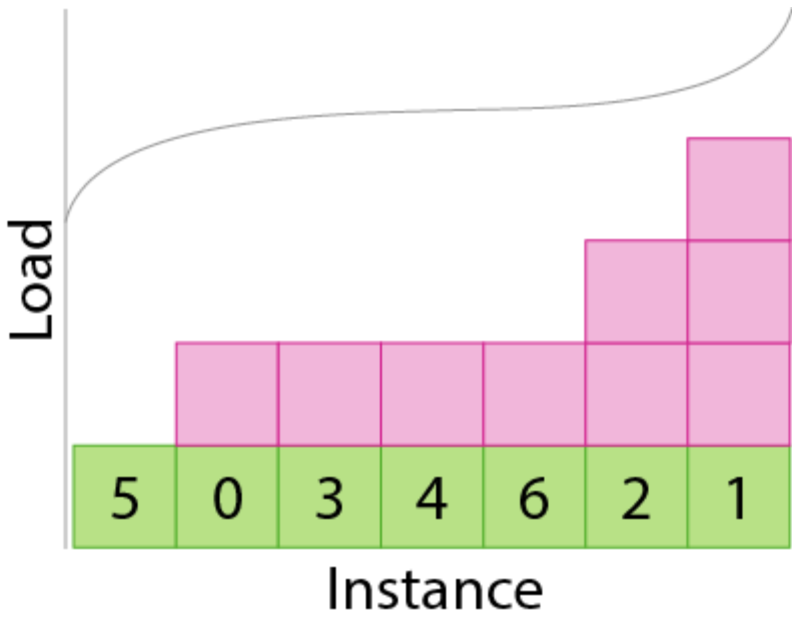
Random aperture: Session distribution that scales, but isn't fair

Our first attempt at a solution in this space was something we called random aperture. This is based on a pretty obvious idea: instead of selecting from all backends, pick a random subset. Good idea, but it raises a new question: how many instances should we pick? As it turns out, there isn't an easy answer and it varies based on the request concurrency of a respective client. Thus, we installed a feedback controller (https://github.com/twitter/finagle/blob/finagle-19.6.0/finagle-core/src/main/scala/com/twitter/finagle/loadbalancer/aperture/LoadBand.scala#L10-L34) within random aperture that sizes the subset based on a client's offered load.

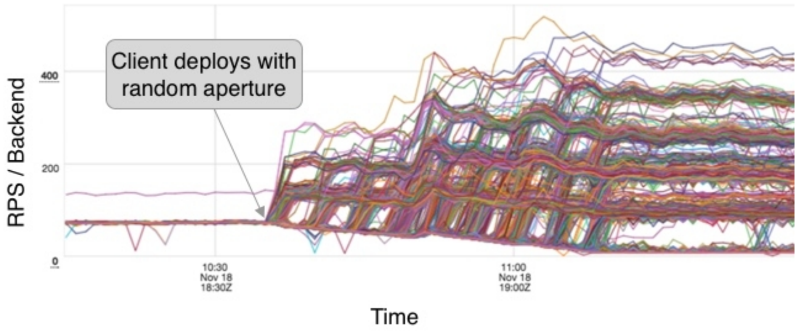
Let's take a look at what a hypothetical random aperture topology might look like with an aperture size of 3:



Already we can see a couple interesting things. First, we've achieved our goal of reducing the overall number of sessions while still maintaining redundancy of 3 in our load balancers. This will reduce the overhead associated with a rapidly growing number of sessions and let us continue to scale our two clusters horizontally. However, we have generated an imbalance on the backends: instance 1 has three sessions and 5 has none! This is better illustrated in histogram form:



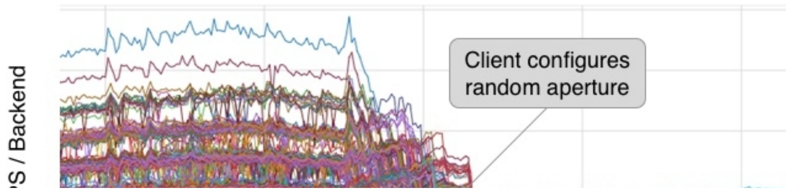
We can see that the load is distributed in approximately a binomial distribution: most backend instances will have 1 or 2 connections but some can have many less, or worse, many more. This is definitely not ideal but this is a small, hypothetical example — is it a real problem in practice? This data shows the effect on requests per second (RPS) per backend when we deploy a production cluster configured to change from P2C to random aperture:

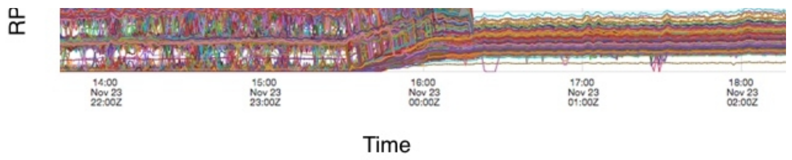


Before the deployment we see a nice collection of traces neatly stacked atop one another. After the deployment: pandemonium! While we succeeded in dramatically reducing our aggregate connection count by over 99%, we've made a mess of our request distribution. Interestingly, we can see the formation of some distinct bands of load: some getting almost no requests and others reaching nearly 400 rps! This is the binomial distribution in action!

It's fun to generate interesting charts, but this situation is clearly not acceptable for service owners. This disparate request distribution makes it nearly impossible for service owners to do capacity planning. Is this a dead end or can we fix it? Fortunately, we have mathematical tools with which to reason about statistics. One approach we took was to suggest that service owners compute their number of connections by modeling their system as a binomial distribution. That is, service owners had to plug in client and server counts along with candidate connection counts until the model converged on an acceptable variance.

The process was a bit crazy since it required a considerable amount of upfront work for service owners, but it worked. This is the same dashboard after optimizing the random aperture parameters via the strategy described above:





Much better, but still not perfect. We see that the discrete steps are gone but we still have a distribution of request rates that is wider than before using random aperture. However, this was an acceptable spread and the only way to reduce it is with more connections, exactly the situation we are trying to avoid. In other words, we found the sweet spot in the connections-vs-load variance curve for this service in this cluster. The problem was that this value would not work for other clusters of differing request rates and sizes, and wouldn't even necessarily stay current for this cluster if it or its backend were to resize. This would work for now and put out a few fires, but this wasn't something that we could turn on across all services at Twitter.

Deterministic Aperture: a highly scalable and fair load balancer

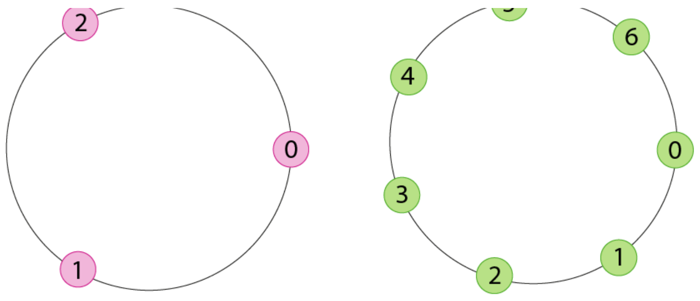
Random aperture was a "two steps forward, one step back" achievement. We could continue to scale our clusters but now forced complex configuration on service owners in addition to wasting capacity on instances that fall into the lower end of the request distribution. How could we improve on this? The gains came from subsetting, the problems came from doing it randomly. We next set out to make subsetting fair by doing it deterministically.

Ring coordinates

In order to deterministically subset, we needed a new way to think about or model our topology. This new representation needed to have a few core properties:

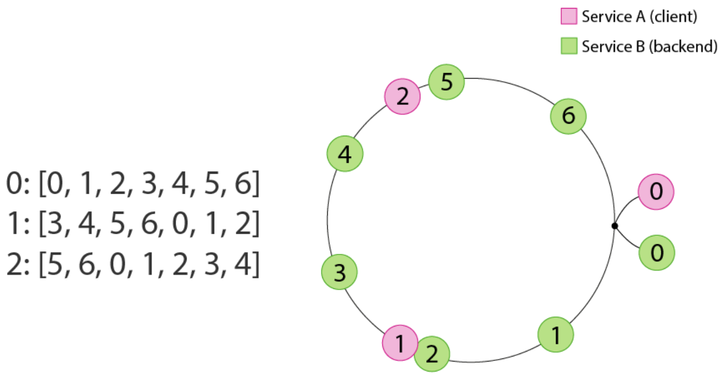
- Light coordination — the representation should not require any heavy weight distributed consensus for the clients to select a subset of backends.
- Minimal disruption — changes to the number of clients or backends should not cause a major reshuffle such that we incur undue connection churn.

One representation that fits the bill is what we call the *ring coordinate system*.

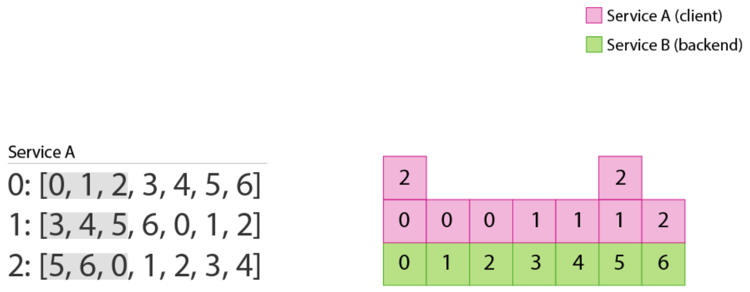


In this representation we place clients and backends on rings in equidistant intervals, named the peer ring and destination ring, respectively. This is similar to consistent hashing except that we can guarantee a perfect distribution of nodes across the ring domain, for a respective service, because we know the membership of every ring ahead of time — it is simply the set of clients or servers!

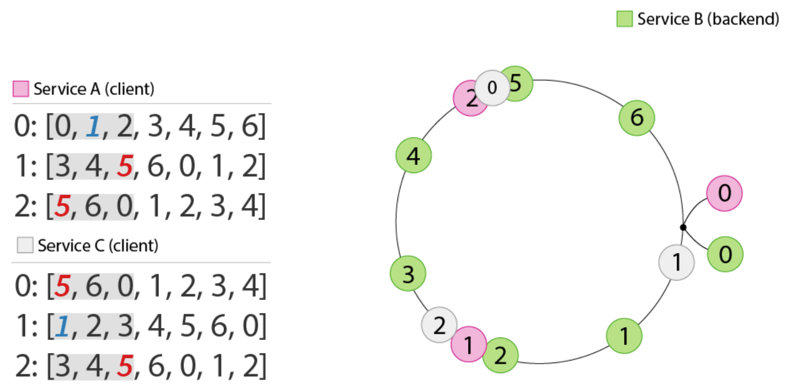
With our peer and destination rings constructed, we can now compose or overlay them to derive a relationship between the respective services. Clients will select their backends in an order defined by walking the ring clockwise:



This results in a far better session distribution than we had with random aperture.



In this model we no longer have instances with zero or three sessions! However, we're not perfect: we still have two backends with two sessions. Unfortunately, just as with random aperture, we can expect this problem to compound if we have more than one service talking to service B:



Instantly you can see crowding on the visual ring representation. When we count up the number of sessions assigned to each instance of service B, we see that instance 1 only has two sessions, while instance 5 is up to four! We have again reintroduced the banding effect of random aperture, albeit at a dramatically reduced scale.

The ring model is a marked improvement over random aperture, but can we do better? In the case of the three clients and seven backends scenario, if we try to alleviate this problem by again manipulating session counts we see that the only way to make it even is by raising every client's session count to seven, effectively going back to the full mesh topology. We could try and enforce some relationship between the number of clients and backends, but clearly that solution is inflexible and won't scale well.

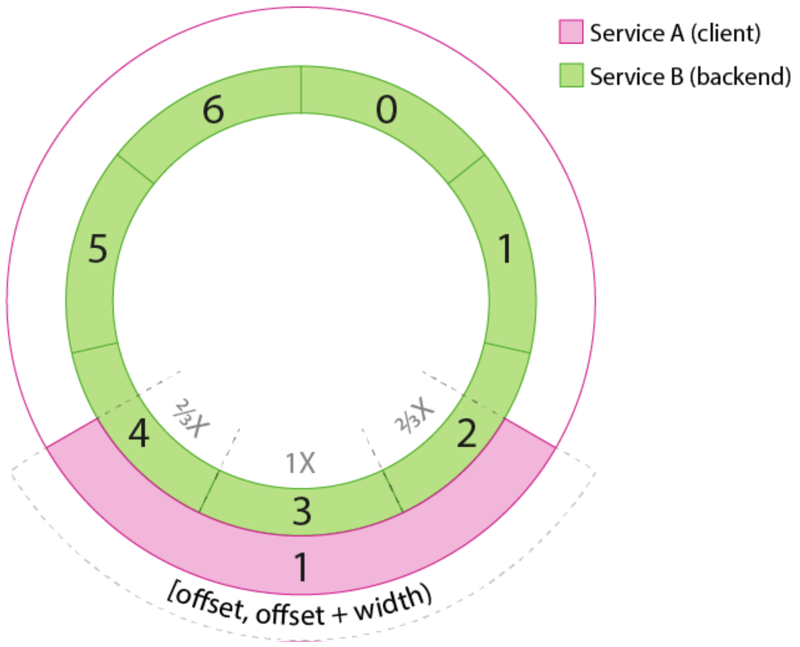
Continuous Ring Coordinates

We tweaked our representation one more time to represent clients and backends as continuous slices of a ring instead of discrete points on a ring. We define the relationship between them as the overlap of their respective slices and, more importantly, that overlap could be fractional. We call this the continuous coordinate model. In this model, we can see that backend instance 2 is asymmetrically shared between instances 0 and 1 of service A. That translates into one extra session, but if we weigh those sessions as to respect the ring representation, we can get an even request load to all of service B.

P2C with Continuous Ring Coordinates

With the continuous ring coordinates model, we have a way to map clients to a subset of backends evenly. How can we respect the representation when balancing requests? One way to do this is to leverage the random selection process in P2C! Each balancer picks two coordinates in its range and maps them to discrete instances on the destination ring. Because we pick randomly and uniformly across a load balancer's range, we inherently respect the fractional boundaries. In other words, we only need to make sure the “pick two” process respects the intersection of ranges between the peer and destination rings, and the rest falls into place.

Let’s take a look at an example of how this works by highlighting the selection process for instance 1 of service A:



Its range overlaps with all of instance 3 and 2/3rds of instance 4 and 2 of service B. This means we would like instance 4 and 2 to receive 2/3rds as much load as instance 3. Receiving less load within the context of P2C means that it has to be picked less often. So our selection process becomes the following:

1. For every request, pick two points within [offset, offset + width) and map those points to discrete instances on the destination ring.
2. Of the two discrete instances, select the one with the least load weighted by its degree of intersection in the ring model.

Step 2 includes a subtle but important modification. Some instances will be picked less often than others because they have a smaller intersecting slice, thus their load will be lower than instances with a larger intersecting slice. In order to avoid biasing towards the smaller intersecting slices when comparing load, we need to normalize it inversely proportional to the size of the intersection.

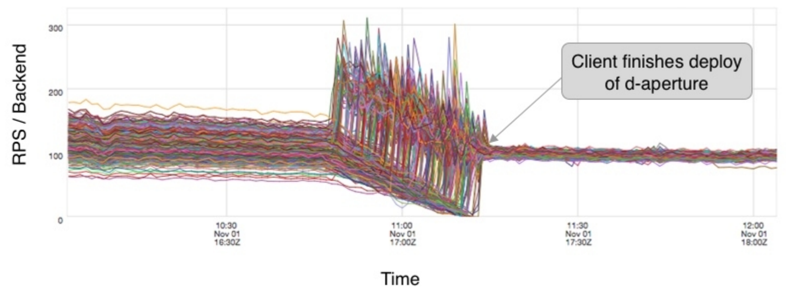
The modified version of P2C in Finagle can be found here (https://github.com/twitter/finagle/blob/finagle-19.6.0/finagle-core/src/main/scala/com/twitter/finagle/loadbalancer/aperture/Aperture.scala#L522-L557). Aside from the math to compute the intersections between rings, the code for this follows the original P2C algorithm closely.

Aperture sizing

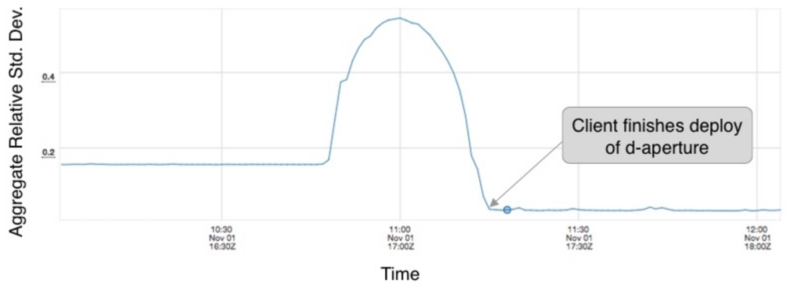
We haven't explicitly mentioned aperture size in this model, but if you've noticed it naturally falls out of the representation. At the minimum, it's 1/N, where N is the number of instances in the peer ring. But it doesn't have to be fixed to the minimum. We can grow or shrink it so long as we do whole rotations (https://github.com/twitter/finagle/blob/develop/finagle-core/src/main/scala/com/twitter/finagle/loadbalancer/aperture/Aperture.scala#L449-L474) around the destination ring. For example, it's still possible to size it dynamically based on request load (so long as all the members of the peer ring agree on the same size which will require a more active form of coordination). By default at Twitter, we've found empirically that a size of at least 10 strikes a good balance between resilience and connection count.

Results

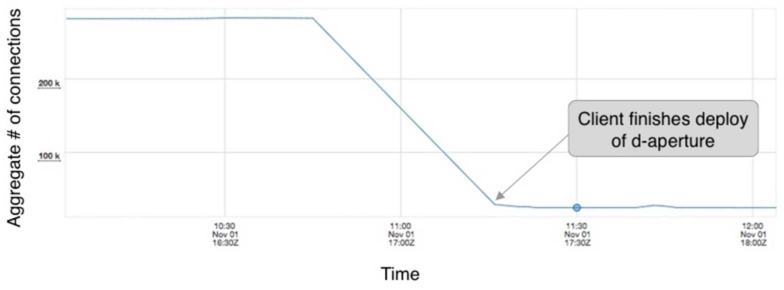
Deterministic aperture has been a quantum leap forward for load balancing at Twitter. We've been able to successfully operate it across most Twitter services for the past several years and have seen great results. Here are some highlights from a representative case where a service topology has upgraded from a well-configured random aperture to deterministic aperture:



We see a marked improvement in request distribution with a 78% reduction in relative standard deviation of load:



All while reducing the connection count drastically. This same service topology saw a 91% drop (from ~280K to ~25K) in connection count during the upgrade:



Keep in mind that this random aperture configuration already minimized session count, and we were still able to reduce the connections by 91%. Most services that were using P2C (and this was the majority case!) saw even better results.

Limitations

The flexibility and performance of deterministic aperture has been successful and general enough to cement its place as the default load balancer algorithm at Twitter. Most services have been migrated to the new load balancer with only a handful of exceptions. With that comes a few small operational changes and limitations to consider.

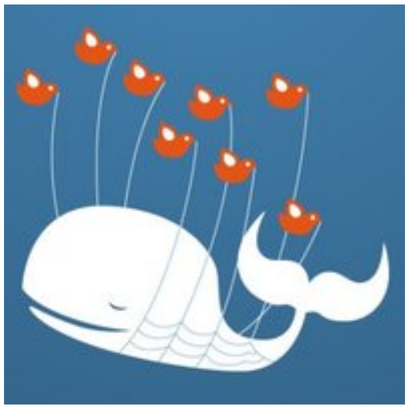
- When setting up a stand-alone instance or small collection of instances as canaries, the number of connections will often be much higher than what you'd see with a production size cluster. This is due to the need to fully cover the destination rings: with only a small number of peers amongst which to divide the load you can expect more connections. In the limiting case of 1 peer we see P2C behavior, for 2 peers we get roughly half the connections of P2C, etc.
- Even with loose coordination requirements we still require that naming updates are propagated relatively rapidly to peers or we lose the even distribution. In practice, we've seen positive results when naming updates exhibit a coherence time on the order of 10s of seconds.
- Like all subsetting algorithms, the deterministic aperture model is predicated on the load generated by individual clients being even. This implies that each service layer using this algorithm is receiving an even amount of load. Outside of this constraint, the unevenness will be propagated to the destination cluster, leading once again to imbalance. The easiest solution to this is to make sure your load is evenly dispersed (i.e., eliminate hot spots) as it enters your system, using a perfectly fair balancer with a full mesh topology at the top of the RPC stack.
- Bursty traffic can also be a problem for deterministic aperture. A common situation where this happens is for services that flush a local cache on an interval. Immediately after a cache flush, the client may see a dramatic increase in load which can be detrimental to the members of its aperture while the rest of the destination ring may be idle. In situations like this a full mesh topology is often the better choice so that after a cache flush all backends can absorb the impulsive load.

Fin

If you've made it this far, there is one last piece of good news: with a couple of pieces of metadata (https://github.com/twitter/finagle/blob/finagle-19.6.0/finagle-core/src/main/scala/com/twitter/finagle/loadbalancer/aperture/ProcessCoordinate.scala#L65-L76) you can use deterministic aperture in your Finagle-based services.

Many people have contributed to the evolution of load balancing at Twitter. We'd be remiss to not acknowledge: Billy Becker, Marius Eriksen, Daniel Furse, Steve Gury, Eugene Ma, Nick Matheson, Moses Nakamura, Kevin Oliver, Brian Rutkin, and Daniel Schobel for their contributions!

We'd also like to thank Rebecca Isaacs, Steve Hetland, Ryan O'Neill, and Elif Dede for their feedback and contributions to shaping this blog post!



(https://www.twitter.com/brycelanderson)
Bryce Anderson

(https://www.twitter.com/rubenoanta)
Ruben Oanta