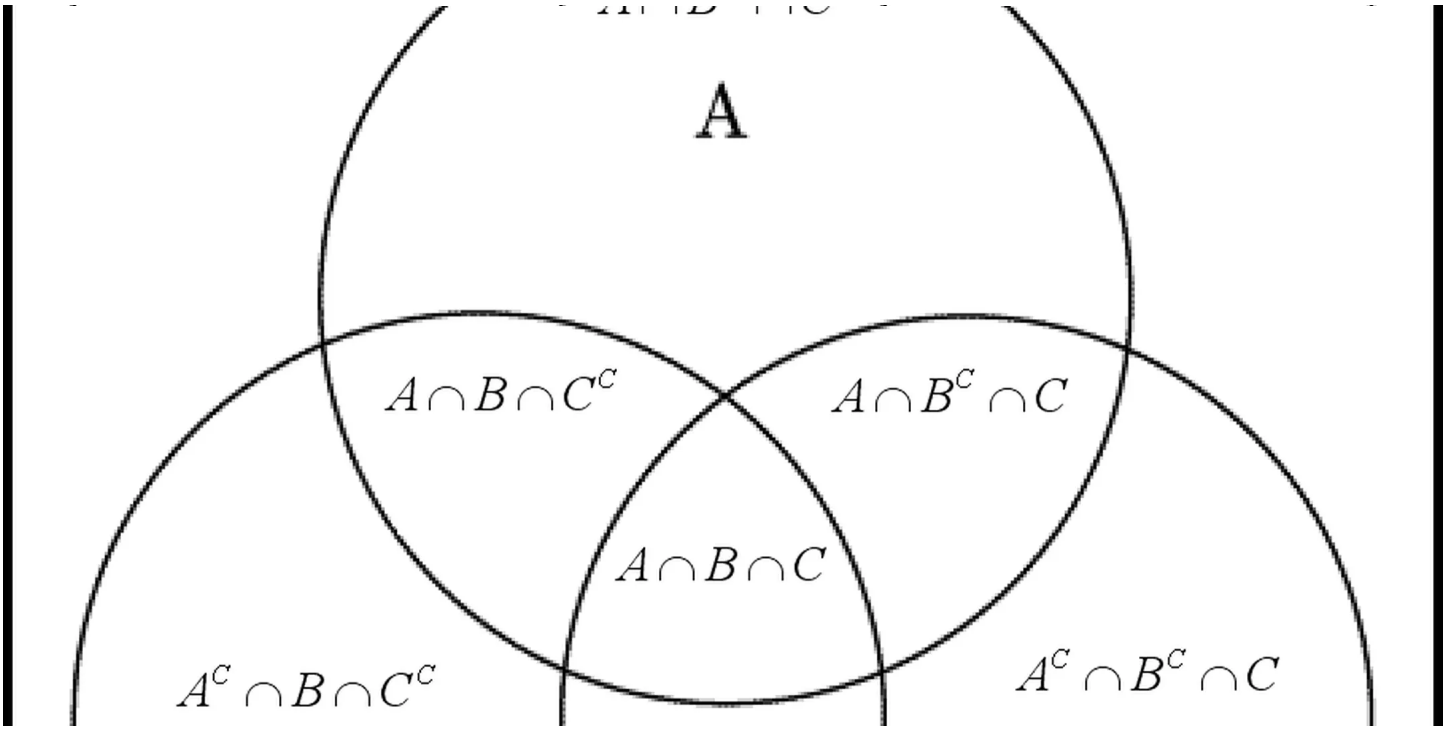
Engineering

Better Load Balancing: Real-Time Dynamic Subsetting

May 17 / Global



Overview

Subsetting is a common technique used in load balancing for large-scale distributed systems. In this blog post, we will briefly introduce Uber's current service mesh architecture that has been powering thousands of critical microservices in Uber since 2016. We will then discuss the challenges we faced when trying to scale the number of tasks in the mesh and issues with our initial subsetting approach. We'll finish with how we came up with the real-time dynamic subsetting solution and its results in production.

Dictionary

Here are a few generic terms that we'll be using throughout the article:

- *A (micro) service*: a routable deployment unit, consisting of multiple *tasks*, exposing *procedures,* and initiating *calls* (e.g., "User Profile Service")
- *Procedure*: a unit of work exposed by a microservice (e.g., "add user")
- *Call*: (i.e., *request*, *query*, *RPC* – Remote Procedure Call) a single procedure execution at a time
- *Load:* measure of business of a given *task*, which could refer to different metrics, like *traffic load* or *CPU load*
- *Traffic:* (i.e., *traffic load*) the amount of work sent to either a *service* or a *task*, typically expressed in Queries Per Second (QPS)
- *CPU load:* the amount of CPU time a given *task* requires to handle the *traffic load*
- *Task*: (i.e., *instance*) one of the identical copies of a *service*, deployed on multiple *hosts*, typically as a docker container
- *Host*: a physical machine, typically hosting collocated *tasks* of multiple independent *services*
- *Task pool:* a logical grouping of *tasks* of the same *service,* typically within a cluster
- *Client*: (i.e., *frontend, caller*) a microservice initiating a call
- *Server*: (i.e., *backend, callee*) a microservice "handling" a call

- *Cluster:* a logical grouping of hosts within a *zone*—each *cluster* contains hundreds of services and each service is typically deployed across multiple *clusters*
- *Zone/Region:* a physical grouping of hosts, providing separate failure domains
- *Control plane:* parts of the system deciding how to shape the traffic, orchestrating the *data plane*
- *Data plane:* (i.e., *forwarding plane*) components actually moving the data around

Uber Service Mesh

What's a Service Mesh?

There are many definitions of service mesh, but ours is: *the layer of infrastructure that allows microservices to communicate with each other via remote procedure calls (RPC) without worrying about infrastructure details*.

When a microservice needs to communicate with another microservice, all that is required are the *destination service name*, *procedure,* and the *request.* The service mesh takes care of the rest, such as:

- Discovery: find the the latest backend tasks of services
  - Needed because cluster management could move backend tasks between hosts
  - Exclude unhealthy backend tasks
- Load balancing: ensure load is properly distributed to available tasks
- Authentication: securely communicate between services
- Traffic shaping: avoid forwarding requests to certain zones/clusters
  - When an outage happens, requests will continue to be routed to available zones/clusters
- Observability: provide insights into latency, traffic patterns, etc.
- Reliability: features like throttling, rate-limit, automatic failover, etc.

Uber Service Mesh Architecture

Uber has been running with microservices architecture since 2014. Several iterations of service mesh have been implemented since, and the current version has been powering RPC between thousands of critical microservices (in millions of containers) since 2016.

General Overview

Uber's service mesh operates largely through self-management. Each of the services in the mesh is fully owned by a product team and the networking layer puts very few constraints on their setup.

Networking layer doesn't place any limits on service connectivity—assuming the right authentication is in place, any service can communicate with any service. The services tend to grow organically, and we don't pre-allocate any capacity for a given service in the networking layer. We see a few services being both decommissioned and created every week, but we don't enforce service lifetime—some services have been running since 2014.

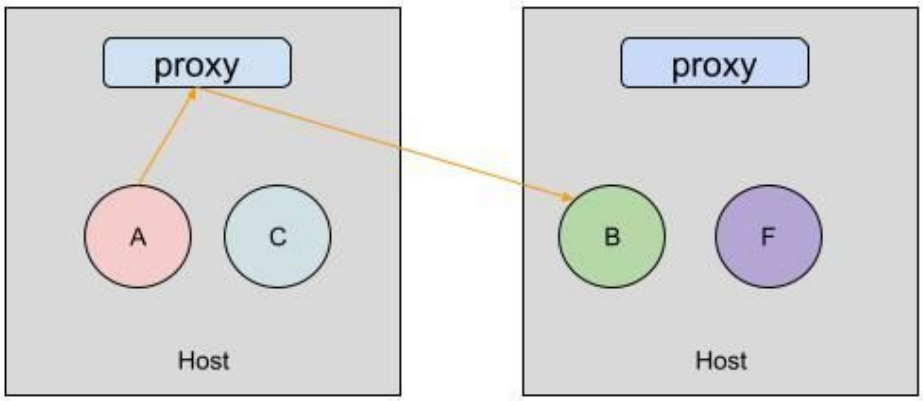
All this has a number of interesting implications:

- A service might move from 100 to 100,000 QPS minute over minute (some do this frequently as part of batch processing)
- A service might start connecting to a new service at any time
- A service might expose a single or hundreds of procedures
- Some services have tens and some tens of thousands of tasks
- It's not uncommon for services to have either hundreds of callers or hundreds of callees
- Some requests have latency measured in <10 ms, and some in minutes
- Tasks of different services can both handle and generate very different throughput—some tasks receive/generate 1-2 QPS, and some 1000+ QPS
- Any number of services might be deploying at any time (and some services run on continuous deployment) resulting in constant placement changes
- It's not uncommon for a particular service, due to either changing call pattern or a bad deployment, to suddenly start rejecting all requests or health-checks
    - As a result, it's also not uncommon for us to see 10-100x re-try storms due to a

misconfigured caller somewhere in the call-chain

Technical Overview

Uber infrastructure operates in multiple zones in multiple regions, across both on-prem data centers and public cloud providers. Most of the control plane components are zonal, and each typically consists of several sub-components.



Service A makes an RPC request to Service B via on-host proxy.

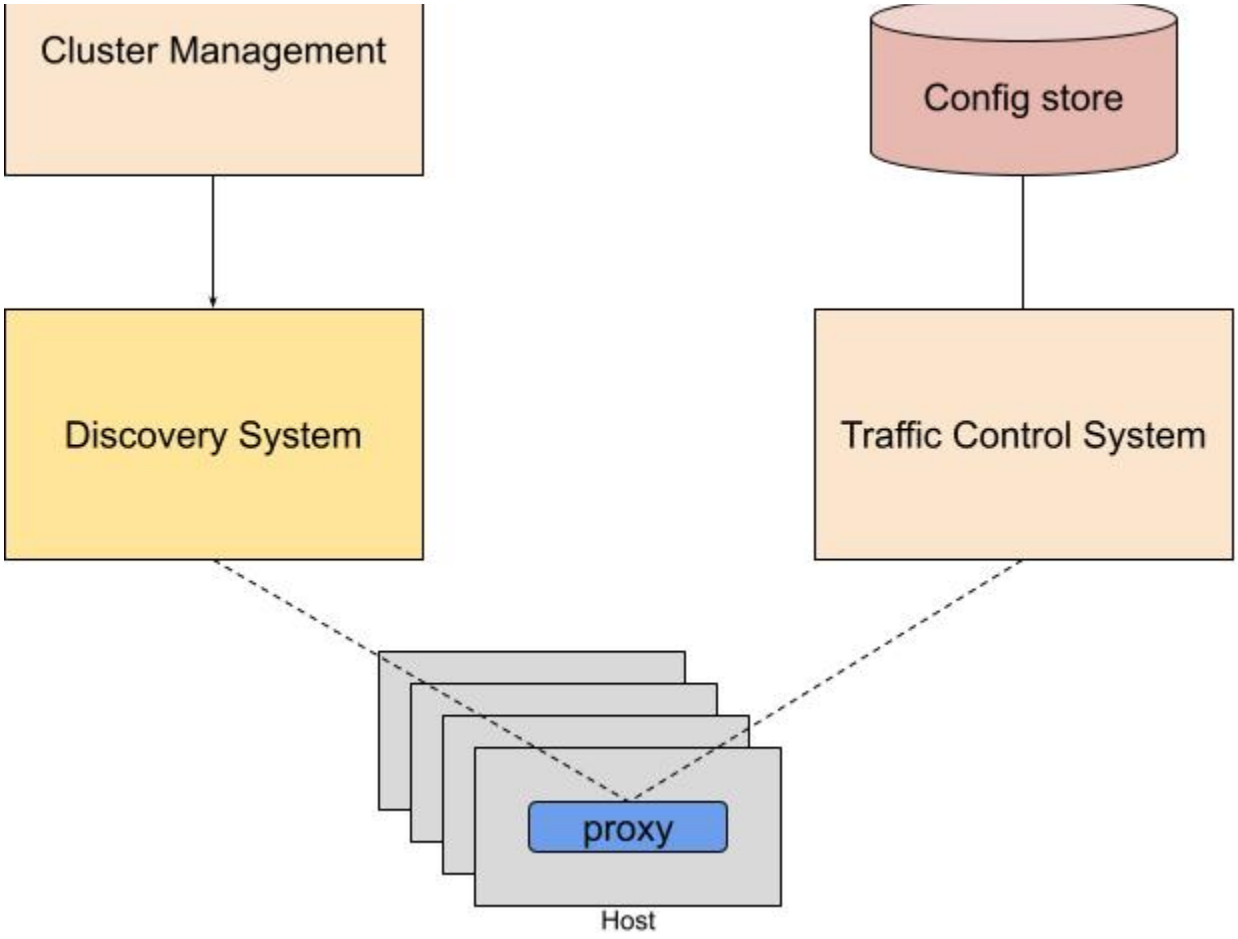
At a very high level, Uber's service mesh architecture is centered around an on-host L7 reverse proxy:

This diagram shows the fundamental flow when service makes RPC calls:

- The caller service sends the RPC to a local port that the host proxy is listening to
- The host proxy will forward the RPC directly to a backend task of callee service

Behind this simple model, there are many details (which are themselves worth another engineering blog).

On the other side of the diagram, the proxy connects to Control Plane components to decide *how* to send the traffic again, at a very high level:



On-Host Proxy

In every host, we have a proprietary on-host proxy listening to several fixed ports. Its responsibility is to receive incoming RPC requests and then forward them to a task of the destination service.

To do that, the proxy will connect to the control plane services to:

- Receive the traffic assignment from Traffic Control service to know how to split traffic between different zones/clusters for a service
- Use the URIs of the backend task pools from the traffic assignment to get the latest task placement from discovery service
- Perform load balancing between available backend tasks

The proxy is a *shared* resource by all the backend tasks running on the same host. In other words, we treat each host as an isolated domain.

Discovery System

This system interfaces with cluster management to keep the most up-to-date information about the service backend tasks. The information is then provided to the on-host proxy so that proxy can perform load balancing on up-to-date instances.

Traffic Control System

This service is the conductor of the whole system; it produces *traffic assignment*, which consists of the traffic split each backend task pool should receive. The assignment is consumed by the proxies.

It exports many config options for service owners to decide how much traffic any given backend task pool should receive. It also allows service owners to create a *drain* to avoid certain backend tasks' pools, or to failover traffic across datacenters.

Now that we have an understanding of the Uber service mesh, we will be focusing on Efficient Load Balancing Technique in this blog, so let's dive deep into that.

Load Balancing At Play

Load Balancing Goals

One of the responsibilities of the service mesh is to make sure that each backend task of a service consumes a similar percentage of available resources—in our context it's typically CPU time. The reason for this is twofold:

- Reliability: an overloaded task will perform worse than expected
- Efficiency: bad load balancing results in resource overallocation (i.e., in order to prevent overloading the "hot" tasks, we add more capacity to spread the load more)

We measure the effectiveness of load balancing via CPU *load imbalance.* We plan to write a separate post about load balancing, but in short, we define the *CPU load imbalance* as the ratio of p99/average CPU utilization of tasks for a given service.

Load Balancing Concepts

What is Subsetting?

Subsetting in the context of load balancing means dividing the set of potential backend tasks into overlapping "subsets," so that while all backends receive traffic in aggregate, each proxy performs the load balancing on a limited set of tasks.

It's common for proxies to maintain long-lived connections to the backend tasks to avoid the cost of setting up and tearing down TCP connections for every request. Each connection consumes some resource (CPU, memory, etc.) at both ends even if it is idle, as some minimum bookkeeping is required.

In theory the maintenance cost is small, but with the number of backend tasks increasing due to the addition of new microservices or scaling existing microservices horizontally to handle more requests, the maintenance costs become non-trivial very quickly. As an example, if a service with 10,000 tasks attempted to call another service with 10,000 tasks, we'd need 100,000,000 TCP connections.

Subsetting alleviates the situations where a backend task receives a large amount of connections from proxies, or a proxy has to connect to a large number of backend tasks.

This problem has been described on the web before:

- Google SRE book explains this in depth
- Twitter has encountered similar problems
- Netflix's Ribbon supports subsets
- Envoy supports "load balancer subsets"

Our solution solves the same problem, but in a different way.

An interesting observation is that subsetting does not necessarily mean worse load balancing. Depending on the load balancing algorithm chosen, a fully connected mesh might perform worse than a well selected subset (e.g., a least-pending-requests load balancing might degenerate into round robin for an excessively large subset). Unfortunately, selecting *the right* subset is tricky, so it's better to overshoot—a too-small subset size will likely lead to a larger degradation than a too-large one.

Legacy Subsetting

Due to the size of the service mesh, the on-host Proxy needs to use subsetting.

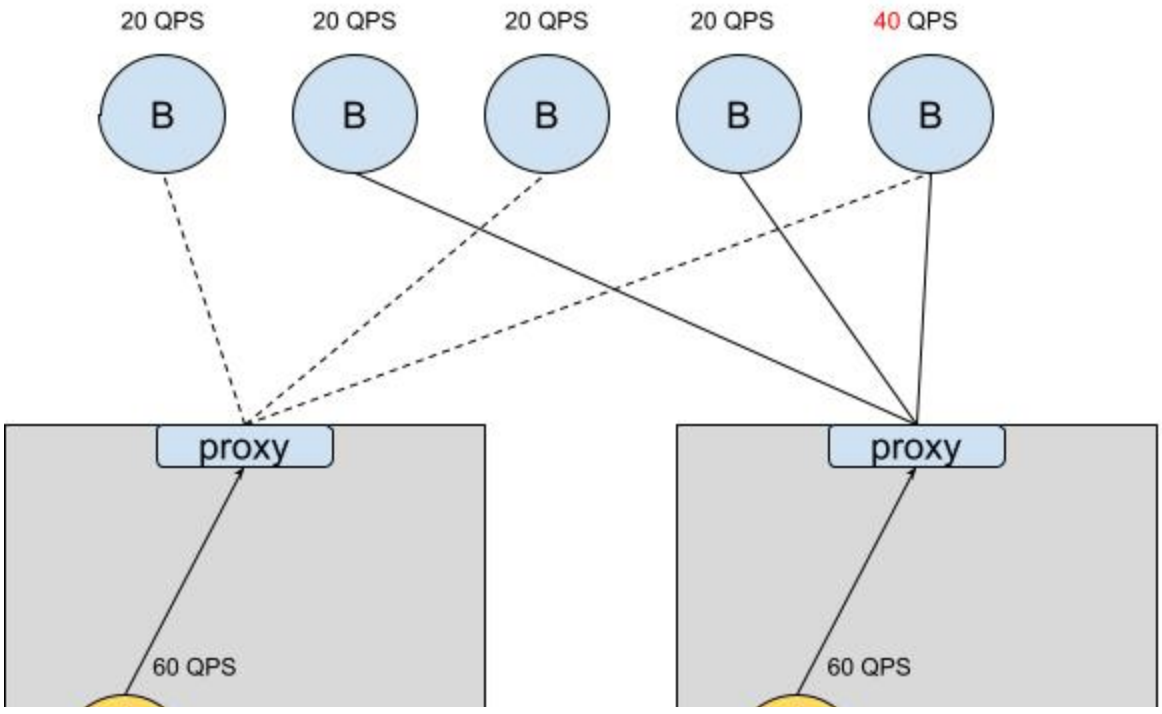
When we started, a static default subsetting size was used. Each microservice owner could decide to use an override if needed. This worked well initially, but presented a number of challenges over time.

Problems with Legacy Subsetting

Imbalance Due to Random Task Selection

As mentioned above, the traffic shaping starts with the *traffic assignment*, which defines the split of traffic that should be sent to each pool. The on-host Proxy then *randomly* picks backend tasks in available pools and performs load balancing upon them, independently from the control plane.

The diagram below shows an example of Service A calling Service B with combined 120 QPS. The static subsetting size is 3 while there are 5 backend tasks for B; as a result, an instance of B receives higher traffic.



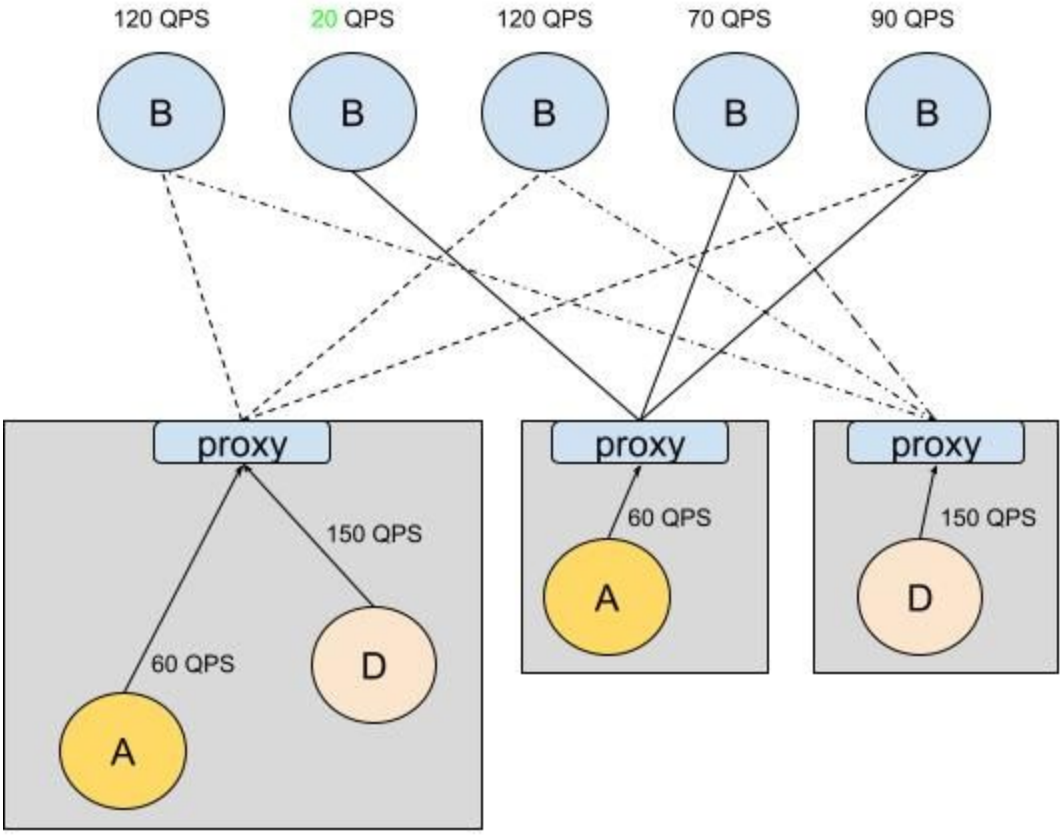
The example shows the potential issue of subsetting: if the number of caller tasks is low, it's possible to cause both overload and underutilization on some of the callee tasks. In some cases, it's possible that some backend tasks won't be selected at all. The hope is that both caller/callee will have the right task count ratios to balance the load with a little luck.

Theoretically, this situation should get better as the service mesh grows—if a backend service is called by hundreds of services or thousands of tasks, one could assume that at some point the "randomness" would even out. This is not what we observed in practice: *due to the imbalanced task count ratio and the heterogeneity of requests, the imbalance for especially the largest services remained high.*

Note that the diagrams above and below are a simplification. The proxies don't just send an even amount of traffic to all connected backends. At the time of this work the proxies were utilizing *"least-pending request"* load balancing, which would *slightly* even out the imbalance caused by subset selection. In practice, however, least-pending wasn't working well enough. This was because each of the proxies kept track of the requests it sent to each backend independently—the only information implicitly exchanged was the response latency of each backend. This helped in load balancing slightly, as the overloaded backends would respond more slowly to all proxies, but our container utilization was not high enough for this to affect the load balancing in a meaningful way. Also, we wanted the load balancing to be effective *before* the overload was visible, as it would have been affecting the end users by then too.

Imbalance Due to Host Co-location

On-host proxy is *shared* between all backend tasks in the same host. As a result, if multiple services in the same host try to make RPC calls to the same callee service, the randomly picked subset of backend tasks will receive more requests than the rest.



An example showing both A & D services calling B, with A & D co-located on the same host.

In some cases, this can also happen if only a single service is generating the traffic. Our Compute Platform might schedule multiple tasks of the same service on the same host, resulting in the same imbalance situation.

Operational Cost

As the service mesh grew, the imbalance issues happened more often.

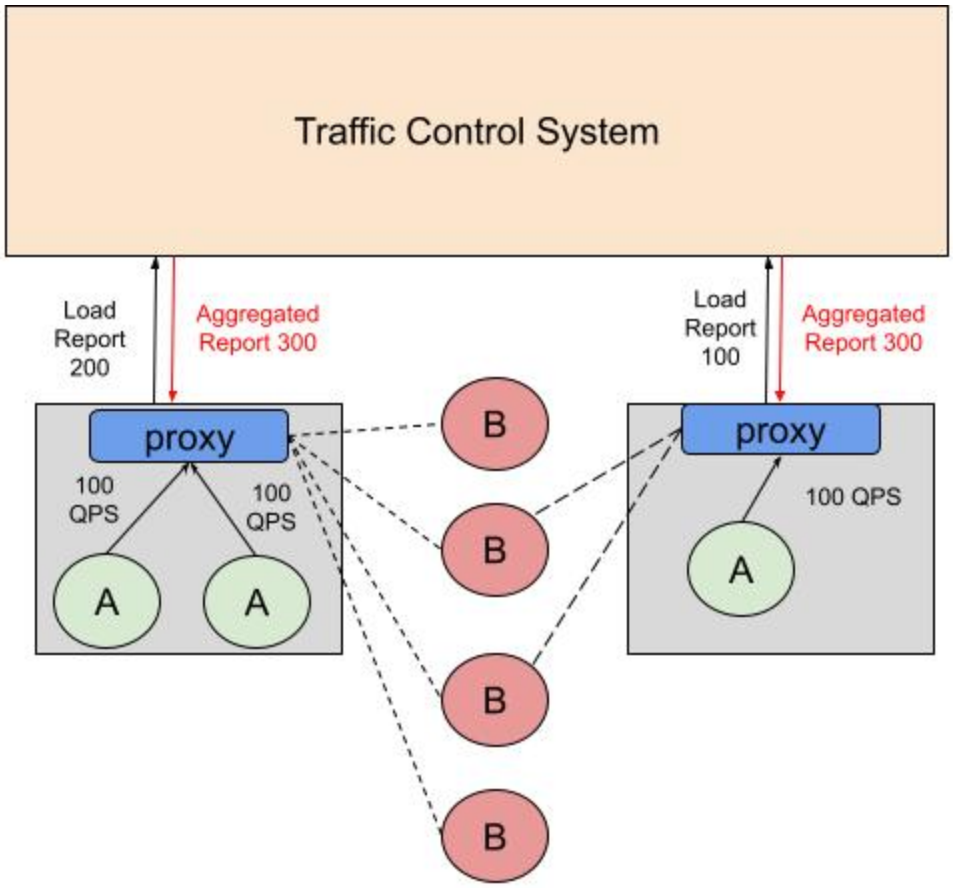
Every time when the issue happened, our team had to work with service owners to determine and update the subset size. Since each microservice has its own capacity allocation and different caller service combinations, it was non-trivial to figure out the right value. Additionally, there was no guarantee that the selected subset size would be right in a week or in a month. This gradually led to an operational nightmare for both our team and the service owners.

Improvements with Real-Time Dynamic Subsetting

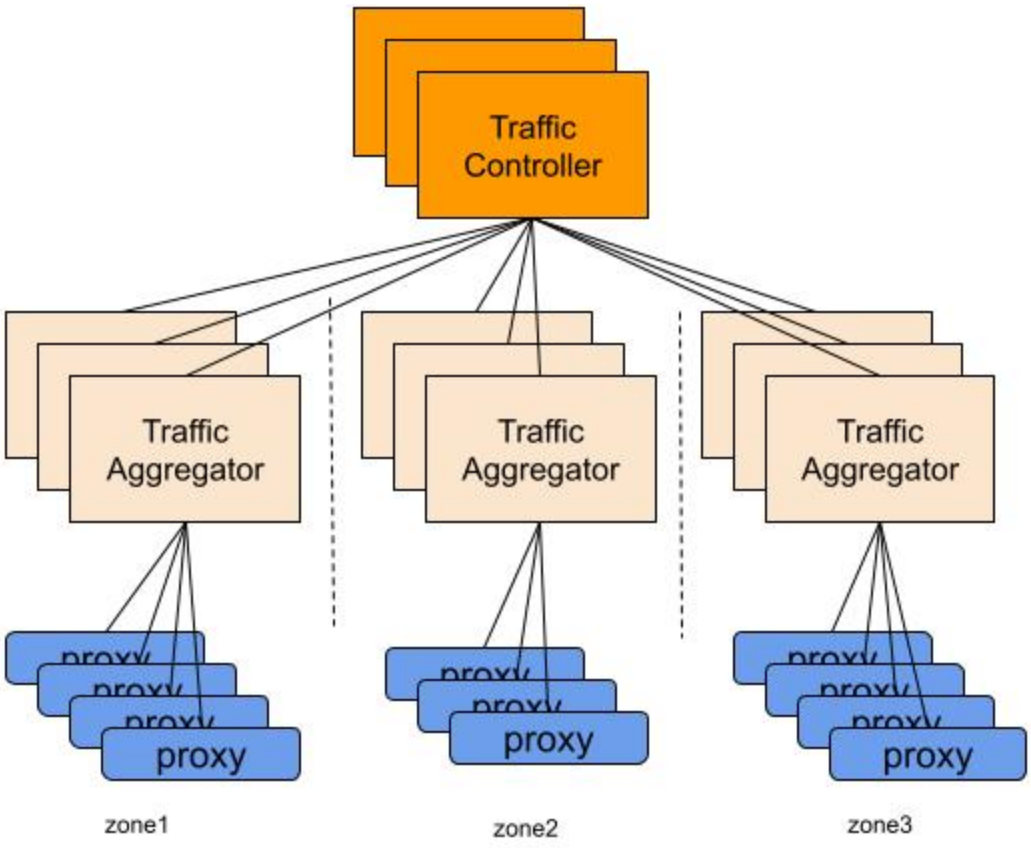
The dynamic subsetting solution we came up with leverages the existing control plane, specifically Traffic Control service.

The basic idea is: if an on-host proxy knows how much QPS a callee service is receiving, it could derive the *ratio* of load it is contributing to the overall traffic. With this information, an on-host proxy could decide its subsetting size *dynamically* based on ratio (i.e., it should expand its subsetting size if it starts to make more requests to a destination service).

Architecture



Aggregation Control Plane



In Uber Service Mesh, on-host proxies periodically upload a traffic *load report* to Traffic Control service. To horizontally scale, we have two layers of aggregation: the first consists of Aggregators that can be scaled horizontally when the number of hosts increases; the second consists of only a handful of Controllers that have a *global* view of traffic distribution. We use the global view of traffic distribution to implement several features, like better load-distribution across datacenters.

For the dynamic subsetting, we leveraged the existing architecture and started to send down aggregated traffic load reports back to on-host proxies. This provides the global view of the total load to each on-host proxy so that it knows the percentage of traffic it is contributing.

This happens in close to *real-time*—the load reports are aggregated globally and pushed back to every host-proxy within seconds. The global load report is not strictly consistent across the fleet, but in practice it's close enough.

On-Host Proxy

At this point, the on-host proxy has 3 relevant pieces of information:

- *Traffic Assignment*: the percentage of traffic to send to each pool

- This is passed down from the control plane

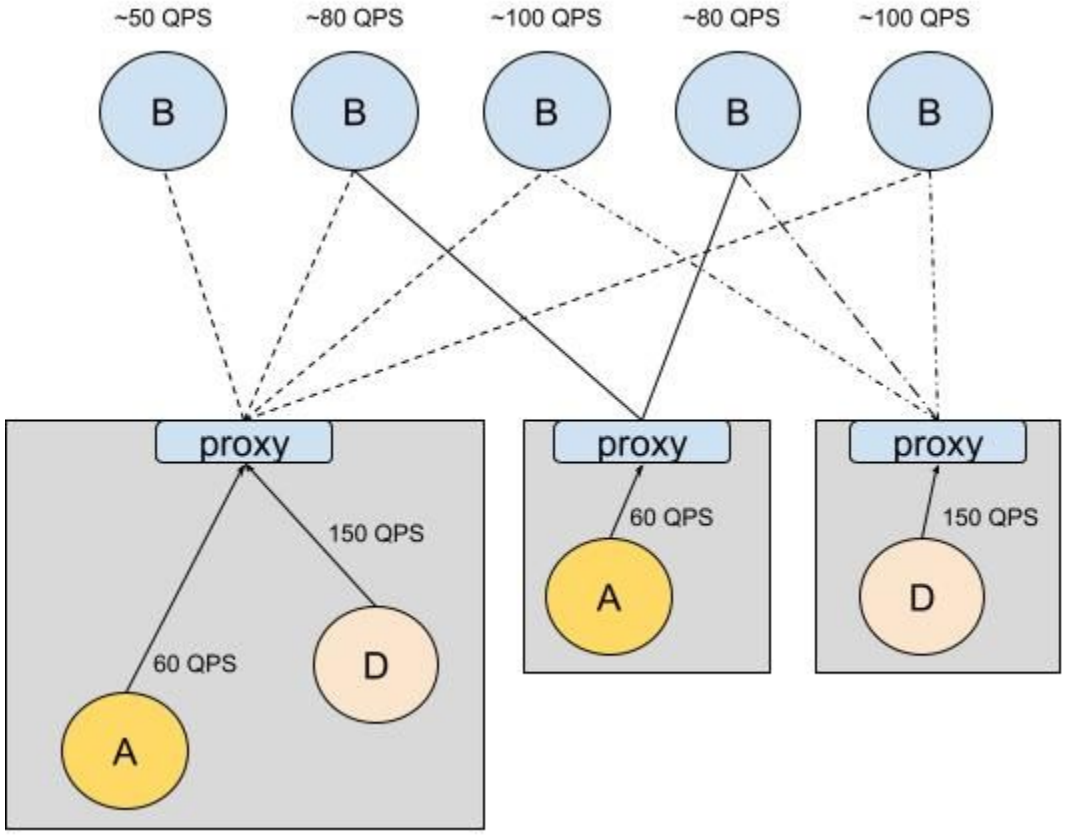
- *Load*: the amount of traffic it's sending to a target service
  - This is tracked independently by each proxy, with some post-processing to stabilize spiky traffic over time

- *Aggregate Load (**new**)*: the overall traffic a target service is receiving
  - This is passed down to the proxy from the control plane

The new piece of information allows us to dynamically adjust the subset size per proxy. We do this in close to real-time, and the desired per pool subset size is recalculated like:

desiredSubsetSize = numberOfTaskInPool * load/(aggregateLoad*assignment) * constant

This means that the proxies generating more traffic connect to more backends and spread their load across a larger subset of tasks than the more idle proxies. This allows us to load balance better, while still achieving the goal of subsetting (capping the total number of connections in the mesh). By tuning the constant number we control the number of outbound connections and thus the load balancing.



The same example, but with dynamic subsetting. Notice that each proxy has a different number of outgoing connections.

While the math itself is simple, some careful performance work was necessary. As we discussed earlier, the proxy knows neither what will get scheduled on the hosts nor what the destination services will be a priori. The proxy needs to be able to handle any request at any time, and so it's forced to monitor subset sizes of tens of thousands of pools constantly. Luckily, most of them don't change often—a typical proxy will talk to only a few hundreds of pools at a time.

The task selection still remains random—there is still no coordination between the proxies. The load imbalance improves significantly, but it's not perfect. Additionally, the subset size adjustment is typically delayed by a few seconds as the load reports propagate between proxies, the control plane, and back to proxies.

Rollout and Results

Rollout

The rollout took approximately 6 months end to end. After internal testing we started with a few early adopters, moved to batch onboarding, and then followed up with the long tail of power services that had previously customized their settings manually. The final set took a disproportionate amount of time—all of the services were typically more critical, more atypical, and more careful. We worked with the service owners individually to avoid degradations.

Problems

The first change that we made as part of rollout was adjusting the "minimum subset" size up: we configured the proxy to connect to at least a few backends, no matter the current *load*. This mitigated issues with both heterogenous callers (a caller with very low RPS that's disproportionately CPU-heavy) and "slow start" when either new workload was placed on the host or the RPS was spiky.

A few services were careful during the onboarding, so we allowed them to customize their aperture constant. At this moment only a very small number of services are running with custom configuration and, from the technical standpoint, we could likely reduce this number further. However, we found that some service owners prefer to run with their own custom, manually managed settings, especially during the initial onboarding.

The final problem we hit was the cost of maintaining TCP connections in the proxy—it was not minimal after all. A combination of multiple high-RPS instances calling multiple large-backend pools (tens of thousands of tasks) on a single host would cause thousands of open connections. This happened on a very small subset of hosts (<0.01%), but due to this Go HTTP2 issue it caused high memory usage for the proxy. We solved this by simply capping the maximum number of connections to backends and carefully rolling out the change across the fleet.

Note that the last issue occurred several months after the migration was finished—the issue only started being visible as part of an independent tchannel-to-HTTP2 migration. As more services were migrating independently, the memory pressure on the proxy grew.

Results

The primary success we'd like to call out is the reduction in maintenance cost: since the rollout (~18-12 months ago) we have had 0 cases of service owners complaining about subsetting. This on its own would have been enough to justify the project, since it significantly reduced toil on both us and service owners.

Efficiency Wins

The project significantly improved load balancing, thus resulting in sizable efficiency wins.

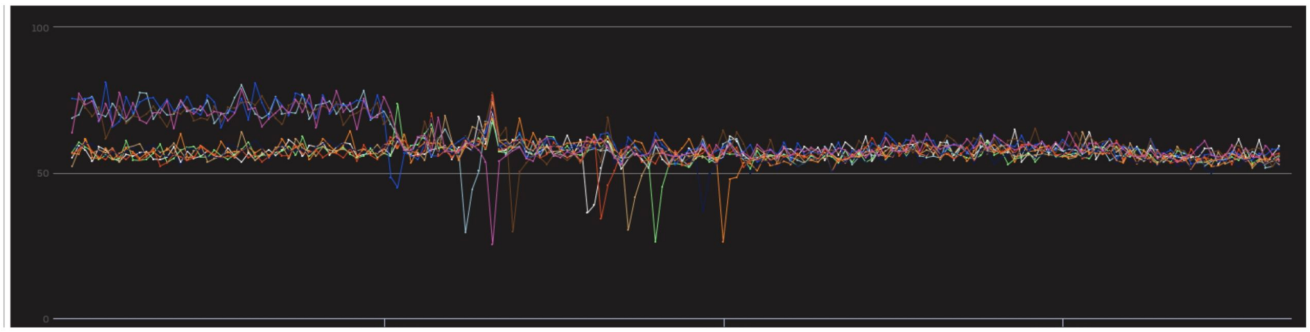
We don't have precise data to claim the exact numbers. First, we originally didn't document enough during the mass onboarding (in retrospect, we regretted not setting up a cross-region A/B test). Due to inherent noise of the mesh it is hard to correlate the data later on. More impactfully, the onboarding coincided with COVID-19. We couldn't easily correlate historical data with the rollouts because COVID had an overwhelming influence on our stats.

We do have earlier stats from manual subset size tuning by partner production engineers in the Delivery organization. With 8 larger services manually tuned, they reported 15-30% p99 CPU utilization reduction; the same services were later on onboarded to Dynamic Subsetting with no regressions, and stats for a few of them improved.

We kept precise data on the "power-services" that were the last to onboard. Out of 36 services, 17 have shown 10%+ p99 CPU utilization reduction, with some services even showing up to 40%! Since the subset sizes of these services had also been manually tuned before, we were

quite satisfied—an automatic system showed no degradations, and even outperformed humans in 50% of cases.

Below is some of the data recorded during the onboarding:



The above diagram shows the CPU utilization for deployments of a critical service; as it shows, all deployments converged to a similar CPU utilization.

We've defined an imbalance indicator (the higher, the more imbalanced) based on production metrics, and the above diagram shows the change after dynamic subsetting was applied for a critical service.

The above diagram shows the CPU utilization for deployments of a critical service; as it shows, all deployments converged to a similar CPU utilization.

We've defined an imbalance indicator (the higher, the more imbalanced) based on production metrics, and the above diagram shows the change after dynamic subsetting was applied for a critical service.

Follow-Up

The subset fanout constant was chosen fairly arbitrarily. Around a year after finishing this project we built a more comprehensive evaluation pipeline that led us to tweak (increase) the aperture constant. The testing also confirmed earlier expectations that increasing the subset sizes even more (i.e., fully connected mesh) would have diminishing results on improving the load balancing.

Potential Improvements

We discussed several potential improvements, but do not plan to implement them as the system works well enough in practice.

Since the subset change is relatively costly for the proxy, we could improve on the hysteresis of the system to reduce oscillating connection churn. In particular, we could increase the subset eagerly, but reduce it lazily. This would also help with spiky traffic patterns.

We are aware of a theoretical pathological case where a low-RPS caller interacting with a very large-backend task pool, with several other large callers, might end up not calling all backends of the service. This is because the traffic is aggregated globally, not per caller. We discussed using the ratio of pool sizes as an additional factor in the subset selection.

When starting the design of this solution, we discussed following up by replacing the random peer selection with deterministic subsetting. This would be similar to solutions described in Twitter and Google links earlier. This would likely improve the load imbalance via better subset selection, but also introduce more complexity, as cross-proxy coordination would be required.

Finally, our current implementation aggregates the load indiscriminately. Since different callers and procedures can have different resource costs on the backend, we could achieve higher precision by tracking the load by callers and/or procedures. This would introduce significant additional complexity. Also, we would not be able to account for request heterogeneity without deep request inspection, which would likely be too costly.

Around 12 months later, as part of a separate project, we implemented an improvement on the proxy (*Assisted Load Balancing*) that, as a side effect, reduced the impact of the random peer selection. It also allowed us to reduce the subset fanout constant. This will be covered in a separate post.

Conclusion

This blog presented an iterative improvement to Uber's service mesh: a *Dynamic Subsetting* system where aperture parameters of the mesh scale automatically and dynamically with the changing traffic. The system has been running in production for nearly 2 years for millions of containers. We achieved the result of not only reducing toil work but also making amazing efficiency wins. It was a great win for Uber, since we were able to reuse the existing service mesh stack and improve on pre-existing load reports to deliver a significant improvement to the stack. We find that the novel coordinator-less approach contributes to both resiliency and scalability of the system.

Acknowledgements

Thanks to Prashant Varanasi and *Ling Yuan* for their contributions in the design, implementation, and rollout of the project.

Chien-Chih Liao

Chien-Chih Liao is a Staff Software Engineer on Uber's Software Networking team. His contributions include traffic control, traffic load balancing, data center failover, and resilience features for Uber's service mesh.

Posted by Chien-Chih Liao, Pawel Krolikowski, Sangeeta Kundu

Category:
Engineering