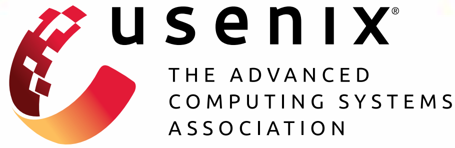


ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta

Harshit Saokar, *Meta;* Soteris Demetriou, *Meta and Imperial College London;* Nick Magerko, Max Kontorovich, Josh Kirstein, and Margot Leibold, *Meta;* Dimitrios Skarlatos, *Meta and Carnegie Mellon University;* Hitesh Khandelwal and Chunqiang Tang, *Meta*

https://www.usenix.org/conference/osdi23/presentation/saokar

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta

Harshit Saokar[1], Soteris Demetriou[1,2], Nick Magerko[1], Max Kontorovich[1], Josh Kirstein[1], Margot Leibold[1], Dimitrios Skarlatos[1,3], Hitesh Khandelwal[1], and Chunqiang Tang[1]

[1] Meta Platforms, [2] Imperial College London, [3] Carnegie Mellon University

Abstract

Datacenter applications are often structured as many interconnected microservices, and the service mesh has become a popular approach to route RPC traffic among services. This paper presents *ServiceRouter* (SR), Meta's global service mesh, which has been in production since 2012. SR differs from publicly known service meshes in several important ways. First, SR is designed for hyperscale and currently uses millions of L7 routers to route tens of billions of requests per second across tens of thousands of services. Second, while SR adopts the common approach of using sidecar or remote proxies to route 1% of RPC requests in our fleet, it employs a routing library that is directly linked into service executables to route the remaining 99% directly from clients to servers, without the extra hop of going through a proxy. This approach significantly reduces the hardware costs of our hyperscale service mesh, saving hundreds of thousands of machines. Third, SR provides built-in support for sharded services, which account for 68% of RPC requests in our fleet, whereas existing general-purpose service meshes do not support sharded services. Finally, SR introduces the concept of locality rings to simultaneously minimize RPC latency and balance load across geo-distributed datacenter regions, which, to our knowledge, has not been attempted before.

1 Introduction

The increasing need for continuous integration and deployment [25] in datacenter environments has led to the widespread adoption of the microservice architecture [**?**, 42], in which an application is decomposed into a collection of services that can be independently developed and deployed. To manage the traffic of remote procedure calls (RPCs) between these services, many organizations use a service mesh [30].

Figure 1 shows the most common form of layer-7 (L7, i.e., application layer) service mesh. In this architecture, each service process is accompanied by an L7 sidecar proxy running on the same machine, which routes RPC requests on behalf of the service. As an example, when service A on machine 1 sends requests to service B, the proxy on machine 1 will load-balance the requests across machines 2 and 3. If the autoscaling system detects an increase in load and starts a new replica of service B on machine 4, the control plane's service discovery function will notify the proxy on machine 1, which will then include machine 4 in its load-balancing targets for future requests for service B.

This paper presents Meta's global service mesh called *ServiceRouter* (SR). SR supports a comprehensive set of features, including service discovery, load balancing, failover, authentication, encryption, observability [1], overload protection [39], distributed request tracing [32], resource attribution for capacity management [16], and duplication of traffic for shadow testing. Due to space limitations, the focus of this paper is primarily on answering the following questions: (1) how to scale a service mesh to millions of L7 routers, (2) how to minimize the hardware cost of a hyperscale service mesh, (3) how to support sharded services which are essential but often overlooked, and (4) how to simultaneously minimize RPC latency and balance load in a geo-distributed service mesh.

Scalability. Traditionally, a software-defined network [18] uses a centralized control plane and a decentralized data plane. Most service meshes [10,30,37] follow this approach and use a central controller to configure the routing table of each sidecar proxy. However, this approach is not sufficiently scalable for a hyperscale service mesh. The control plane has a dual function of generating global routing metadata and managing each L7 router. We advocate for keeping the former in the central control plane, but decentralizing the latter by transferring its function to L7 routers. Each L7 router should be self-configuring and self-managing so that the central control plane can scale out easily.

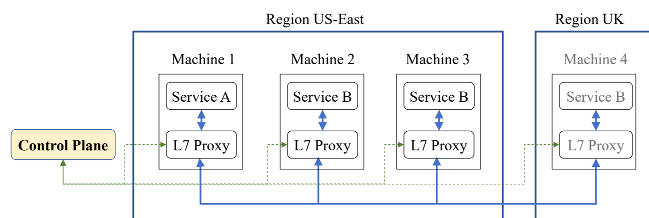


Figure 1: Sidecar-proxy-based service mesh.

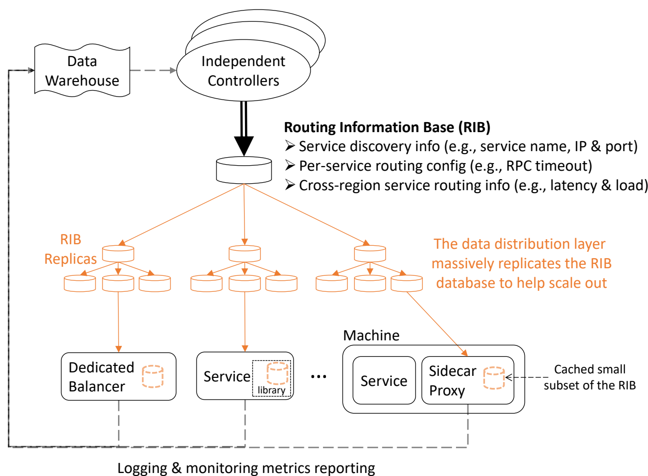


Figure 2: ServiceRouter's scalable service-mesh architecture.

Figure 2 shows SR's scalable architecture. On the top, different controllers independently execute functions such as registering services and generating a per-service cross-region routing table. Each controller independently updates the central *Routing Information Base* (RIB), and is not concerned with configuring or managing individual L7 routers. In the middle of Figure 2, the distribution layer replicates the RIB so that there are sufficient RIB replicas to handle read traffic from millions of L7 routers. At the bottom, guided by the RIB, each L7 router self-configures without the control plane's direct involvement. Initially, an L7 router's routing table is empty. When it receives an RPC request that targets a service, it fetches the routing information for the service from an RIB replica and subscribes to future RIB updates for the service.

Hardware cost. Existing service meshes [10, 30, 37] use sidecar proxies to forward requests (Figure 1). However, this approach incurs extra hardware costs due to the overhead of the extra routing hop, such as data serialization and deserialization in the proxy. Istio's benchmarking [47] shows that 0.35 vCPU can handle 1,000 requests per second. Therefore, it would need the equivalent of 1,750,000 AWS `t4g.small` VMs to route 10 billion requests per second.

SR eliminates the need for a proxy and its associated hardware cost by providing the service-mesh function through a library called *SRLib*. SRLib is linked into service executables and routes RPC requests directly from clients to servers. However, this approach requires changes to services' source code, which is not always possible. For example, our services written in Erlang cannot link SRLib into their executables.

To meet the diverse needs of services, SR enables the seamless coexistence of different types of L7 routers, including Istio-style sidecar proxies, AWS-ELB-style [5] dedicated load balancers, and gRPC-style lookaside [24] load balancers, as shown in Figure 2. The key insight that enables SR's versatility is that the controllers at the top of Figure 2 are agnostic to the L7 routers at the bottom, allowing the L7 routers to choose their own architecture.

The embedded SRLib helps us achieve significant hardware savings. Currently, 99% of RPC requests at Meta are routed by SRLib, and the remaining 1% is routed by sidecar proxies and a group of dedicated load balancers that consume thousands of machines. If we were to completely switch from using SRLib to using proxies to route 100% of our traffic, we would need to add hundreds of thousands of extra machines.

Sharded services. Sharding [34] and replication are two key techniques for building scalable services. In our fleet, the vast majority of RPC traffic is for sharded services. Despite their importance, existing general-purpose service meshes do not directly support routing for sharded services. For example, in Figure 1, assuming that Service B's replicas on machines 2, 3, and 4 host various data shards that can dynamically migrate across machines, it is possible for the proxy on machine 1 to route a request to machine 2 mistakenly, even if the request is meant for a shard on machine 3.

SR makes sharding support a top priority and uses a single framework to support both sharding and replication. As sharding is often tied to application logic, our key insight is to enforce separation of concerns by defining a simple and generic sharding abstraction between the service mesh and services. This allows SR to route traffic for different sharded services without needing to know their application logic.

Cross-region routing. Existing solutions [4, 41] are not optimized for routing across geo-distributed datacenter regions. For example, in Figure 1, should machine 1 route requests to machines 2 and 3, which have a higher load, or to machine 4, which has a longer network latency? Moreover, how to ensure that the resulting global traffic distribution for a service matches the global supply of the service's capacity in different regions? These questions have not been well answered before.

To better support cross-region routing, we introduce the concept of locality rings for services to express their preferred tradeoff between latency and load. For example, a service can instruct SR that if and only if the load in the local region goes above 70%, SR can relax the locality constraint and route some local traffic to other regions in the same continent; if the load further increases above 80%, SR can even route some local traffic to regions in a different continent. SR collects global traffic and load information for each service, computes a cross-region routing table that conforms to the requirements specified in locality rings, and disseminates the routing table to L7 routers to guide their routing. This allows SR to provide globally optimized traffic shaping for services.

Contributions. We summarize our contributions below.

- SR is designed for hyperscale. While there may be proprietary systems of a similar scale, their specifics are not publicly available, and existing open-source service meshes do not scale well [57]. We hope that our experience can be helpful to those who seek to build scalable service meshes.

- SR supports the seamless coexistence of different types of L7 routers in one service mesh, including sidecar proxies [30], dedicated load balancers [5], lookaside load balancers [24], and an embedded routing library. To save on hardware costs, SR routes 99% of RPC requests in our fleet using the embedded library. This approach, along with the scale at which we utilize it, might be unique in the industry.

- While existing service meshes exclusively focus on unsharded services, which only account for 32% of our fleet's RPC requests, SR provides built-in support for sharded services, which account for 68% of our traffic.

- Although primitive forms of locality-aware routing existed before [31], our novelty is to introduce the concept of locality rings to simultaneously minimize RPC latency and balance load across geo-distributed regions.

2 Comparison of Services Mesh Architectures

In this section, we compare different architectures of service mesh. The design space, shown in Table 1, is determined by the answers to two key questions: (1) which component fetches and caches the routing metadata, and (2) which component routes application RPC traffic. In Table 1, *Library*, *Kernel*, *Local*, and *Remote* mean that RPC routing or maintenance of routing metadata is performed by an embedded library, the kernel, a local proxy/daemon on the RPC client machine, or a remote proxy/service outside the client machine, respectively.

2.1 Different Types of L7 Routers in SR

SR allows different L7 router setups to coexist in one service mesh in order to support diverse use cases. These setups are shown in Figure 4 and explained below. Different types of L7 routers interoperate well and can send RPC requests to the same server at the same time.

SRLib. This setup is shown in Figure 4(a) and corresponds to solution (9) in Table 1. It provides the service-mesh function through a library, which is directly linked into the RPC client's executable. The library can route requests directly to servers without the need for a proxy, eliminating the extra hardware cost and routing latency of a proxy. The client only needs to fetch and cache a small part of RIB (called the *miniRIB*) that is actively used by the client.

We run a separate *RIBDaemon* on the client machine to cache miniRIB, instead of relying on SRLib to do so. This separation allows for the use of `cgroup` to provide strong isolation between a) RIBDaemon's less urgent background work that keeps miniRIB up-to-date and b) SRLib's latency-sensitive foreground work that routes RPC requests and is on the critical path of application performance. Updates to RIB can be very spiky and when those updates are pushed to miniRIB, they can cause a spike in CPU usage to process the updates. Figure 3 shows the spiky CPU usage of a production machine's RIBDaemon. When `cgroup` throttles RIBDaemon, it has little impact on SRLib because SRLib consults RIBDaemon only once on its first RPC for a service and all subsequent RPCs for the service go directly from SRLib to servers without involving RIBDaemon. In contrast, if miniRIB is managed by SRLib, `cgroup` cannot isolate the resource usage for maintaining miniRIB from the application's own resource consumption because SRLib is linked into the application.

		Which component manages and caches miniRIB?			
		Lib	**Kernel**	**Local**	**Remote**
Which component forwards application RPC traffic?	**Lib**	(1) ✗	(5) ✗	(9) SRLib	(13) SRLookaside
	Kernel	(2) ✗	(6) eBPF	(10) ✗	(14) ✗
	Local	(3) ✗	(7) ✗	(11) SRSidecarProxy	(15) SRSidecarProxy plus SRLookaside
	Remote	(4) ✗	(8) ✗	(12) ✗	(16) SRRemoteProxy

Table 1: The complete solution space for service mesh. The symbol ✗ indicates undesirable solutions.

SRLookaside. This setup, shown in Figure 4(b) and corresponding to solution (13) in Table 1, addresses the issue of RIBDaemon running on every RPC client machine and consuming resources, particularly memory. It eliminates RIBDaemon by moving the function of miniRIB management and server selection to a remote and shared *SRLookasideService*, while still routing RPCs directly from clients to servers without going through an intermediate proxy.

Historically, Meta used a large fleet of small machines with as little as 16GB memory because of their advantages in power efficiency. Accordingly, SRLookaside was developed to save memory on those small machines. Now even our small machines have at least 64GB memory and hence the usage of SRLookaside was deprecated, because the limited memory savings no longer justify the burden of maintaining the SRLookaside service.

SRSidecarProxy. This setup, shown in Figure 4(c) and corresponding to solution (11) in Table 1, incurs extra hardware costs and routing latency like Istio [30], but its implementation is much more scalable than Istio, because each SRProxy self-manages without the control plane's involvement and only caches miniRIB instead of the entire RIB. At Meta, the usage of SRSidecarProxy is mostly limited to services written in Erlang because SRLib does not directly support Erlang.

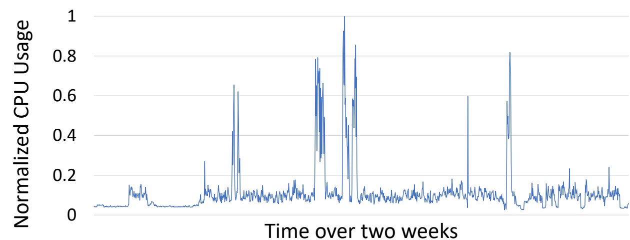


Figure 3: Spiky CPU usage of a machine's RIBDaemon.

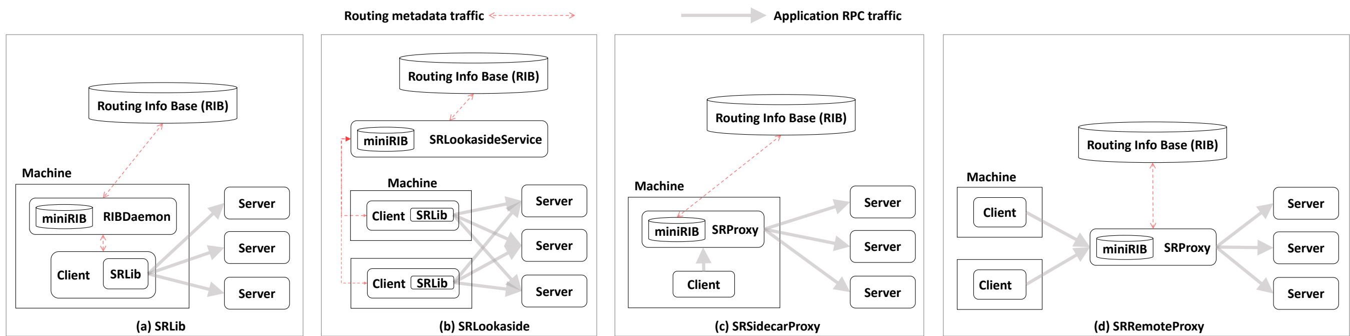


Figure 4: Service mesh design alternatives. The diagrams show how RPC clients send requests to RPC servers.

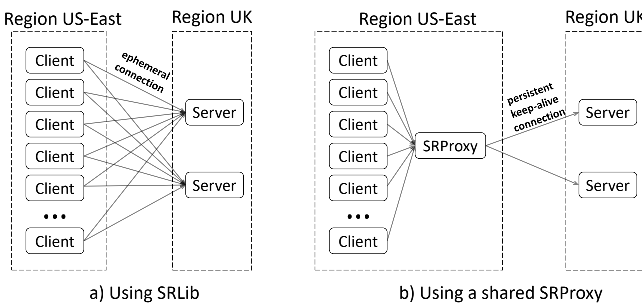


Figure 5: A shared SRProxy is better at dealing with many clients sending infrequent cross-region RPC requests.

SRRemoteProxy. This setup, shown in Figure 4(d) and corresponding to solution (16) in Table 1, is similar to AWS ELB [5]. SRRemoteProxy functions as a dedicated load balancer shared by multiple clients, reducing the number of RPC connections and increasing the reuse of keep-alive connections, as illustrated in Figure 5. Suppose there are a large number of clients and each client sends a request to a server in a remote datacenter region occasionally. Each RPC would experience a long delay due to the three rounds of cross-region round-trip time needed to establish a new TLS/TCP connection. A shared proxy eliminates this overhead by keeping a small number of cross-region connections alive and reusing them to send requests on behalf of many clients.

2.2 Comparison of L7 Routers

Next, we compare solutions in Table 1. Solutions (1)–(4) are undesirable because managing miniRIB in the library would impact the application's performance due to lack of isolation. Solutions (5), (7), and (8) are undesirable because there is no system call to access miniRIB cached in the kernel. Although solution (6) exists in the form of eBPF-based service mesh [35], its function is limited by what can be done by an eBPF program in the kernel. For example, Cilium [29]'s eBPF program can only handle L3/L4 protocols and it still has to use a sidecar proxy to handle L7 protocols. Similar to solution (6), solutions (10) and (14) are undesirable because of the difficulty of implementing advanced L7 routing features in the kernel.

Service Mesh Alternatives	A1: HW cost	A2: direct RPC	A3: fast RIB	A4: save mem	A5: unchg code	A6: share conn
SRLib	✓	✓	✓	≈	✗	✗
Sidecar Proxy	✗	✗	✓	≈	✓	✗
Remote Proxy	✗	✗	✓	✓	✓	✓
Lookaside	≈	✓	✗	✓	✗	✗

Attributes	Description
A1: HW cost	No extra hardware cost for proxy or lookaside service.
A2: direct RPC	Application RPC traffic goes from clients to servers without the overhead of going through an intermediate proxy.
A3: fast RIB	No overhead to access Routing Information Base (RIB) outside the client machine thanks to local RIB caching.
A4: save mem	No extra memory usage on the client machine thanks to the elimination of the local RIB cache.
A5: unchg code	No need for application source code modification.
A6: share conn	Benefits of multiple clients sharing a proxy, e.g., better load balancing or connection reuse (Figure 5).

Alter- natives	When to use a particular service-mesh setup	Usage at Meta
SRLib	Use SRLib for large-scale deployments where hardware costs and routing latency are most important.	99% of traffic
Remote Proxy	Use remote proxies if it benefits from multiple clients sharing a proxy, e.g., to improve connection reuse when there are many low-traffic clients (Figure 5).	Some limited use
Sidecar Proxy	Use sidecar proxies if you cannot modify application source code to use SRLib, or SRLib does not support the app's programming language (e.g., Erlang).	Only one-off use
Lookaside	Use a remote lookaside service to reduce the memory used on every client machine for caching miniRIB.	Deprecated

Table 2: Comparison of service mesh design alternatives.

Solution (12) is undesirable because it is strictly worse than (16), i.e., if routing is performed by a remote proxy, it is better to move miniRIB to the remote proxy as well. Theoretically, solution (15) uses less memory on the client machine than (11) does. However, (15) is not used at Meta since even (11) is not widely used and the added benefit of (15) is limited.

Finally, for ease of access, we summarize in Table 2 the comparison of the design alternatives.

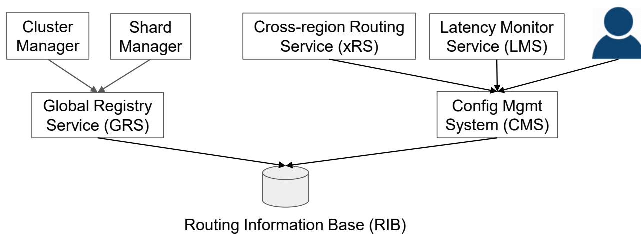


Figure 6: ServiceRouter's control-plane components.

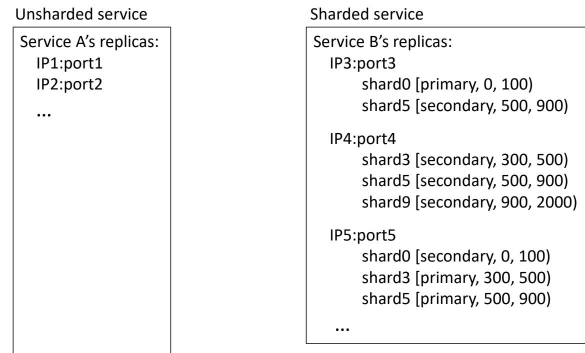


Figure 7: Examples of GRS' service registry records.

3 ServiceRouter Design

In this section, we first present an overview of ServiceRouter and then elaborate on its key design ideas.

3.1 Overview

SR supports all four types of L7 routers depicted in Figure 4. For the sidecar or remote proxy setup, we add a wrapper layer atop SRLib code to run it as a standalone proxy. SR's control-plane components are depicted in Figure 6 and further explained below.

Routing Information Base (RIB). RIB is a Paxos-based key-value store with nine Paxos acceptors distributed across five geographic regions to ensure high availability. It centrally stores routing metadata for all services running in all regions. It uses thousands of Paxos learners to create many local RIB replicas in every region to ensure high read throughput and availability even if a region is disconnected from other regions. We discuss how to scale RIB in §4.1.

Global Registry Service (GRS). GRS maintains service and shard discovery information in RIB. Figure 7 shows two example services registered at GRS. *Service A* is replicated but not sharded. When the cluster manager [53] starts or stops a container for *service A*, it informs GRS to update the list of *service A*'s replicas. We will explain SR's built-in support for sharded services in §3.3.

Configuration Management System (CMS). CMS [52] allows customization of the routing policy for each service, including RPC timeout, connection reuse, locality routing preference, etc. Services owners follow the configuration as code paradigm to author, review, and commit routing configurations. It also supports automated configuration updates. For example, the latency monitoring service (LMS) periodically aggregates and commits configuration updates related to cross-region latency to guide SRLib's routing decisions.

Cross-region Routing Service (xRS). Compared with a centralized load balancer, SRLib only has a local view of the traffic from one client and might not make globally optimal routing decisions. xRS addresses this problem by aggregating global traffic information for each service and computing a cross-region routing table, which is disseminated via RIB and consumed by SRLib to guide its routing decisions.

3.2 Service Discovery

A RIBDaemon runs on each machine and maintains a so-called miniRIB that caches the specific parts of RIB that are needed by the RPC clients running on the machine. Initially, miniRIB is empty. When SRLib wants to send an RPC request to a particular service, such as *service X*, it requests *service X*'s routing metadata from RIBDaemon. RIBDaemon fetches the metadata from a RIB replica, caches it on disk so that it can survive machine reboots, subscribes to future updates related to *service X*, and finally returns the metadata to SRLib. SRLib also subscribes to RIBDaemon for future updates and caches the metadata in memory (but not on disk) for later reuse so that it won't contact RIBDaemon on every RPC request.

When the deployment of *service X* is changed in the future, the cluster manager informs GRS to update RIB. The update is immediately pushed to all RIB replicas, which further push the update to every RIBDaemon that subscribes to *service X*'s routing metadata. Finally, RIBDaemon pushes the update to SRLib. *Service X* may be deployed in multiple datacenter regions, and its replicas in each region are managed by a different regional cluster manager. All of these cluster managers inform GRS to update the same service-registry record for *service X* so that a client's RPC request can potentially be routed to a replica in any region (§3.4.1). The RPC client of a service can choose to send requests only to servers located in the same region as the client. In this scenario, to reduce overhead, RIBDaemon subscribes only to routing updates originating from the local region.

With the help of the cluster manager, clients do not need to independently discover a server's failure through timeouts. When a server is brought down for planned maintenance, such as code deployment, the cluster manager first updates RIB to inform the clients and then stops the server. For unplanned failures, the cluster manager detects all kinds of failures, such as process crashes/hangs and machine failures, and updates RIB to inform the clients.

3.3 Support for Sharded Services

SR provides built-in support for sharded services. In Figure 7, *service B* is both sharded and replicated. We define a sim-

ple sharding abstraction between SR and services to enforce separation of concerns, so that SR can route traffic without needing to know a sharded service's internal application logic. Specifically, a service specifies how a 128-bit key space is partitioned into shards. Each shard can be independently replicated and migrated across containers. Each shard replica is associated with an abstract role, e.g., *primary* or *secondary*. In this example, *shard5* corresponds to the key range [500, 900) and its replica on *IP4:port4* serves the *secondary* role. SR is not concerned with the real semantics of the shard key or role, and merely routes requests according to a client's request.

```
SRClient *cln = SR_get_client("ServiceB", 618/*key*/, SECONDARY);
cln->foo(); // Invoke RPC for foo().
```

In this example, SR discovers that *shard5* contains key 618 and *shard5*'s *secondary* role is served by its replicas on *IP3:port3* and *IP4:port4*. SR picks one of them to serve the request according to the load balancing policy. In the service's implementation, the *primary* and *secondary* roles could be mapped to the leader and follower replicas of a database, respectively.

SR's shard-map abstraction is generic and currently supports hundreds of sharded services. Most but not all of them are managed by a common shard manager [34], which notifies GRS to update the shard registry when new shards are added or removed, or existing shards are migrated across containers.

With SR, sharded and unsharded services share and reuse all the sophisticated components in SR (Figures 2 and 6). Moreover, routing for sharded services works out-of-the-box without any additional effort. In contrast, existing general-purpose service meshes do not support sharded services, and applications have to develop their own solutions.

Design alternatives. One alternative to SR's shard-map approach is *consistent hashing* [33]. Given a list of servers, it deterministically determines the server responsible for a given key based on hashing. As a result, it does not need to store the *shard map* in Figure 7. Despite its advantage in simplicity, consistent hashing is insufficient for advanced sharding use cases, as its deterministic key assignment does not support dynamic migration of shards in response to shard load changes [2, 34]. SR provides built-in support for both consistent hashing and the shard-map approach. As shown in our previous work [34], out of the hundreds of sharded services at Meta, the number of services that choose to use a flexible shard map is 5.4 times higher than the number of services that choose to use consistent hashing, which confirms the importance and effectiveness of the shard-map approach.

Another alternative to SR's shard-map approach is to allow a service to provide its own custom lookaside-service implementation. This approach can provide maximum flexibility and separate the service's custom shard discovery and selection logic from the service mesh. Both gRPC [24] and SR's lookaside interfaces can support this approach. At Meta, some service owners were initially interested in this approach because of its flexibility. However, they ultimately did not use it because of the burden of maintaining a custom lookaside service, and also because it turns out that the shard-map approach and consistent hashing together are sufficient for nearly all sharded services.

3.4 Load Balancing

SR's load-balancing solution is based on the `Pick-2` [41] algorithm. `Pick-2` randomly samples two servers from a candidate pool and chooses the server with less load as the RPC target. However, using `Pick-2` alone is not sufficient for a geo-distributed service mesh. Therefore, we have developed three novel techniques to complement `Pick-2`: 1) Consider regional locality when sampling two random servers (§3.4.1). 2) Sample two random servers from a stable subset of servers, rather than all servers, to maximize connection reuse (§3.4.2). 3) Take an adaptive approach to load estimation based on the workload characteristics (§3.4.3). Further details on these techniques are provided in the following sections.

3.4.1 Locality Awareness

In a geo-distributed service mesh, a faithful implementation of `Pick-2` would cause long RPC latencies because it does not take regional locality into account. Our measurements show that the P50 of within-region RTT is only $116\mu s$, while the P50 of cross-region RTT is $35ms$ and the P99 is as high as $163ms$. These data emphasize the importance of considering regional locality when routing RPC requests.

Instead of `Pick-2`'s approach of randomly sampling two servers from the candidate pool, SR uses the so-called *locality rings* to filter out long-latency servers that are far from the client, and then sample from the remaining nearby servers. Each service can define a set of rings with increasing latencies, e.g., $\left[\text{ring}_1\text{: } 5ms \mid \text{ring}_2\text{: } 35ms \mid \text{ring}_3\text{: } 80ms \mid \text{ring}_4\text{: } \infty\right]$. The Latency Monitoring Service (LMS) periodically updates RTTs between regions, and RPC clients obtain them via CMS.

An RPC client uses cross-region RTTs to estimate its latency to different servers. Starting from ring_1, if the client finds any RPC server whose latency is within the latency bound for ring_i, it filters out all servers in ring_{i+1} and above, and randomly samples two servers from ring_i. If the service has no servers in ring_i, it considers servers in ring_{i+1}, and so forth. SR's default setting maps $\left[\text{ring}_1|\text{ring}_2|\text{ring}_3|\text{ring}_4\right]$ to $\left[\textit{same region}|\ \textit{neighboring regions}|\ \textit{same continent}|\ \textit{global}\right]$.

Filtering by locality rings reduces routing latencies but still has limitations due to lack of a global view. First, servers in ring_i might be overloaded while servers in ring_{i+1} are underutilized. Second, clients' local routing decisions might not lead to an optimal global traffic distribution that matches the global supply of server capacity. In particular, when a region X fails, if all clients independently decide to reroute their requests initially going to X, to X's nearest region Y, they may overload Y, bring it down, and together move onto the next region Z, and so forth, causing a domino effect.

The Cross-region Routing Service (xRS) solves these problems by using global information to compute a per-service cross-region routing table whose entry $[P_{ij}]$ means that P_{ij} fraction of the service's RPC requests originated from region R_i should be routed to region R_j. The cross-region routing table is stored in RIB and disseminated to all clients. When an RPC client wants to send a request, it follows the traffic distribution P_{ij} to randomly choose a destination region, and then applies the normal routing algorithm to select a server in the destination region.

xRS can purposely update the routing table to shift traffic out of a region in preparation for an upcoming maintenance event or in response to a disaster. While doing so, it tries to avoid overloading other regions. Guided by a PID controller [50], it gracefully shifts traffic across regions to prevent over-reaction. If there is insufficient capacity globally, it creates so-called black holes in the routing table to instruct clients to drop certain traffic instead of overloading servers.

Next, we describe how xRS computes the cross-region routing table. xRS collects traffic and load information globally, and simulates how a region's load would change if more or less traffic is routed to the region. For each service, xRS periodically fetches load information from its servers and aggregates it by region. It also collects requests per second (RPS) served by servers in each region, which is used to calculate the *RPS cost* as the ratio of a region's load to its RPS. *RPS cost* is the estimated load increase due to a unit of RPS increase. For example, the load for a region with a 60% load serving 100 RPS, would increase by 0.6% if 1 RPS is added to the region.

xRS strives to simultaneously minimize RPC latency and balance load across regions. It expands the locality rings with load thresholds, e.g., $\left[\text{ring}_1\text{: } 5ms : 55\% \mid \text{ring}_2\text{: } 35ms : 65\% \mid \text{ring}_3\text{: } 80ms : 80\% \mid \text{ring}_4\text{: } \infty : \infty\right]$. Intuitively, it means that, for example, when ring_1's load goes beyond 55%, xRS will relax its latency restriction and start to consider routing traffic to servers in ring_2, and so forth. This load-enriched locality ring information is not directly consumed by SRLib, but instead is fed to xRS to compute a per-service cross-region routing table as follows. xRS first tries to serve all requests in the source region locally, by setting $\forall i \; P_{ii} = 1$ and $\forall i \forall j \neq i \; P_{ij} = 0$. Then assisted by each region's *RPS cost*, it identifies the most loaded region and tries to follow the preference in the locality rings to move some of the region's traffic to nearby regions. This process repeats until either no regions are overloaded or all regions are equally loaded.

Currently, 46% of our services are routed using xRS' cross-region routing tables, while the rest are routed using the baseline locality rings without the routing tables. Some services choose not to use xRS due to the overhead of collecting global traffic and load information. Moreover, some services generate high traffic and require low latency, and as a result, they prefer to fail a request instead of routing it across regions.

In total, about 16% of RPC requests in our fleet are routed across regions. This emphasizes the importance for global service meshes to optimize cross-region routing, an area that is largely overlooked by existing service meshes.

Design alternative. With xRS, service owners need to apply their domain knowledge to set the thresholds for network RTT and server utilization in locality rings. To avoid the burden of setting these thresholds, an alternative approach is to use end-to-end RPC latency as the sole metric, which, in theory, would automatically consider both network RTT and server utilization. The load-balancing goal of this latency-focused approach would be to minimize the average RPC latency. Pacifici et al. [45] used a similar approach in a local cluster setting. However, SR does not follow this approach because, based on both queuing theory [11] and our production experience, modeling latency at high utilization is not robust. This implies that xRS would not be able to accurately predict how traffic shifts would affect RPC latency.

Moreover, minimizing RPC latency by trading long queuing delay at the RPC server for long cross-region network RTT is not a robust method, as it can lead to overloading of nearby servers and a high RPC error rate. We explain this through an example. Suppose a client sends requests to two servers X and Y, where X is in the same region with a $100\mu s$ RTT and Y is in a different region with a $100ms$ RTT. Further assume that it takes $1ms$ to process a request. To minimize the RPC latency, the latency-focused approach would send all requests to X, which is in the local region, until its queuing delay reaches $100ms$, and only then it would start to send requests to Y, which is in a remote region. However, with a processing time of 1ms, when the queuing delay at X reaches 100ms, X would be severely overloaded and might experience a high error rate. Overall, in a geo-distributed environment where network RTT may vary by three orders of magnitude, from $100\mu s$ to $100ms$, the latency-focused approach is not robust.

3.4.2 RPC Connection Reuse

Our measurements show that setting up a new TLS/TCP connection takes 1.6ms and consumes 14KB of memory on each side. To reduce this overhead, SR keeps the TLS/TCP connections and reuses them across different RPC requests. However, the randomization used by `Pick-2` makes connection reuse ineffective. As `Pick-2` randomly samples two servers out of all n servers for each request, over time, an RPC client communicates with all n servers. However, it is impractical to maintain keep-alive connections with all n servers when n is large because of the memory and CPU overhead required to maintain the connections.

To improve connection reuse, an RPC client chooses a stable subset of k servers out of all n servers (often $k \lll n$), and keeps reusing these k *stable servers*. Upon each RPC request, `Pick-2` samples two servers out of the k stable servers instead of all n servers. Over time, the client maintains keep-alive connections with the k stable servers.

One challenge is for each RPC client to independently choose their k stable servers while globally the load spreads

evenly across all n servers. Suppose a server on average maintains keep-alive connections with M clients. When a new server is added to the existing n servers, an ideal and stable solution should require only M clients to drop one existing server out of their list of k stable servers and add the new server to their list, so that the new server also serves M clients like other servers.

With SR, each RPC client uses Rendezvous Hashing [9,54] to select k stable servers, which achieves the ideal properties described above. Specifically, a client uses its unique client ID as hashing salt, computes the hashcodes of all servers, and chooses the k servers with the largest hashcode. Stable servers and Rendezvous Hashing together help SR maximize connection reuse. In production, over 99% of our RPC requests reuse existing connections.

Design alternative. We prefer Rendezvous Hashing over Consistent Hashing [33] because it allows SR to use weighted hashing to assign client connections proportional to a server's compute power. This in combination with a weighting mechanism to bias the `Pick-2` probability proportional to a server's compute power, solves the problem that our large fleet runs multiple generations of hardware that have very different performance characteristics. Moreover, Rendezvous Hashing achieves better load balancing. For example, when a server dies, its load is evenly redistributed to other servers even without using Consistent Hashing's virtual servers.

3.4.3 Adaptive Load Estimation

In order for `Pick-2` to choose a routing target between two candidates, it needs to know the load information. By default, SR uses the number of outstanding requests at an RPC server to represent its load. In addition, SR allows custom load metrics such as CPU, memory, disk, or any application-level metric. Currently, 77% and 18% of the RPC requests in our fleet use the number of outstanding requests and CPU usage as the load metric, respectively, while the rest use other metrics.

To determine a server's load, a client has two options: 1) Poll the server for its load right before deciding whether to send a request to the server, which incurs additional overhead and latency. 2) Have the server include its load information on its responses and then cache it at the client for later reuse, which is efficient but may result in the client using stale load information and causing load imbalance.

To strike a balance between these two approaches, SR employs an adaptive mechanism. An RPC response is always piggybacked with the server's current load information. When evaluating a server's load before sending a new request, the client uses the cached load information only if it is sufficiently fresh (method 1). Otherwise, it polls the server for its realtime load if the network RTT to the server is low compared with the server's average request-processing time (method 2). In the worst case that the cached load information is stale and the polling overhead is high, it chooses one of the two candidate servers at random. (method 3).

Design alternative. LI [13] attempts to solve the load-estimation problem by using methods 1 and 3 alone, without method 2 (polling). Data from our production system show that, with SR's adaptive mechanism, about 50%, 25%, and 25% of RPC requests end up using methods 1, 2, and 3, respectively. This confirms the usefulness of introducing the just-in-time polling method.

4 Evaluation

Our evaluation attempts to answer the following questions:

1. Does SR scale well? (§ 4.1)
2. To what extent does SRLib save hardware costs, and when should one use SRProxy versus SRLib? (§ 4.2)
3. Can SR balance load within and across regions? (§ 4.3)
4. Are sharded services important, and can SR effectively support both sharded and unsharded services? (§ 4.4)

4.1 Scalability

Hyperscale is a key design goal that distinguishes SR from most of the existing service meshes. SR currently operates in tens of datacenter regions and runs millions of L7 routers to serve tens of thousands of services. GRS globally distributes service discovery information for millions of containers and hundreds of millions of shards.

To understand the scale of individual services, we plot the number of servers used by services in Figure 8. A small fraction of services are very large while most are very small. Specifically, while 90% of services each use less than 200 servers, 2% of services each use more than 2,000 servers and the largest service uses about $90K$ servers. Figure 9 shows the RPS of services. Similarly, while most services have a low RPS, some hyperscale services process billions of RPS. These hyperscale services often demand the highest performance and most sophisticated features from SR. Overall, Figures 8 and 9 show that SR scales well for both a small number of hyperscale services and a large number of small services.

In SR's overall architecture (Figure 6), the central RIB enables separation of concerns for different components in the

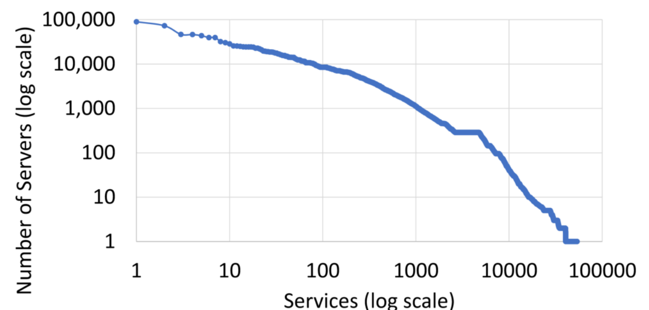


Figure 8: Number of servers used by services. Each dot represents one service. Note that both axes are in log scale.

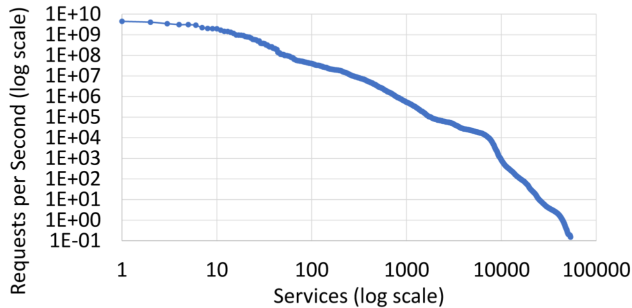


Figure 9: Requests per second by services. Each dot represents one service. Note that both axes are in log scale.

data plane and the control plane so that each component can scale out independently. However, RIB itself might become a bottleneck, primarily due to the large amount of service-discovery data stored in RIB and the associated write rate. Currently, RIB's total size is about 12GB, processing about 335 writes/second at a total data rate of about 39MB/second. The write rate is low because writes are heavily batched; in particular, sometimes thousands of updates for the service-discovery registry are batched into a single write. In the past, we used ZooKeeper as the data store for RIB, which could not scale beyond a few GB, and hence we sharded RIB. Now we use an in-house data store [6] that scales well and there is no urgency to shard RIB further. Overall, currently RIB is not a bottleneck and it can be further sharded to scale out if needed.

The distribution of RIB is fast, far from reaching any scaling bottleneck. We operate about 2,000 RIB replicas globally, which form a 2-layer data distribution tree among themselves. In our production environment, the distribution latency of an RIB update to reach clients in geo-distributed datacenter regions is 400ms/900ms/1300ms at P50/P95/P99, respectively.

Due to the propagation delay of service discovery information, any system that does service discovery and routing, not just SR, will encounter the problem of stale routing information on some clients. In the face of stale routing information, SR guarantees correctness and strives to minimize performance impact. For example, if an SR client sends a request for a specific shard to a server that no longer holds the shard, the client will receive an error and automatically retry a different server. To improve performance, SR minimizes the chance of this scenario by implementing graceful shard migration. When migrating a shard from server X to server Y, as described in our previous work [34], the shard manager first starts the shard on Y, then updates RIB to redirect clients to send traffic to Y, and finally stops the shard on X.

As long as RIB scales well, xRS, CMS, LMS, GRS, and the L7 routers can all scale out horizontally. xRS is sharded by service and can scale out horizontally. Computing the routing table for one service only takes about one second. CMS processes about 10,000 routing-configuration changes per day for about 2,500 services, and 99% of those changes are driven by automation tools. Overall, the rate of writes to CMS is far from reaching any bottleneck.

To understand the nature of routing-configuration changes, we list the types of the most frequent changes on an average day. A data pair (X%/Y) below means that every day X% of the total changes are for a specific type, which are applied to Y number of services. The top types of changes are 1) *processing timeout* (27%/1700), the server-side RPC processing timeout; 2) *locality ring* (30%/700); 3) *traffic shedding* (11%/3), the percentage of traffic to be shed for a given client ID in an overload situation; and 4) *shadow traffic* (6%/100), the percentage of production traffic to be replicated to a test service. These data demonstrate that it is easy to dynamically reconfigure the routing policies for thousands of services at the central CMS. Moreover, it shows that locality ring is an important feature that is frequently tuned for services to achieve the best cross-region routing performance.

4.2 Hardware Cost

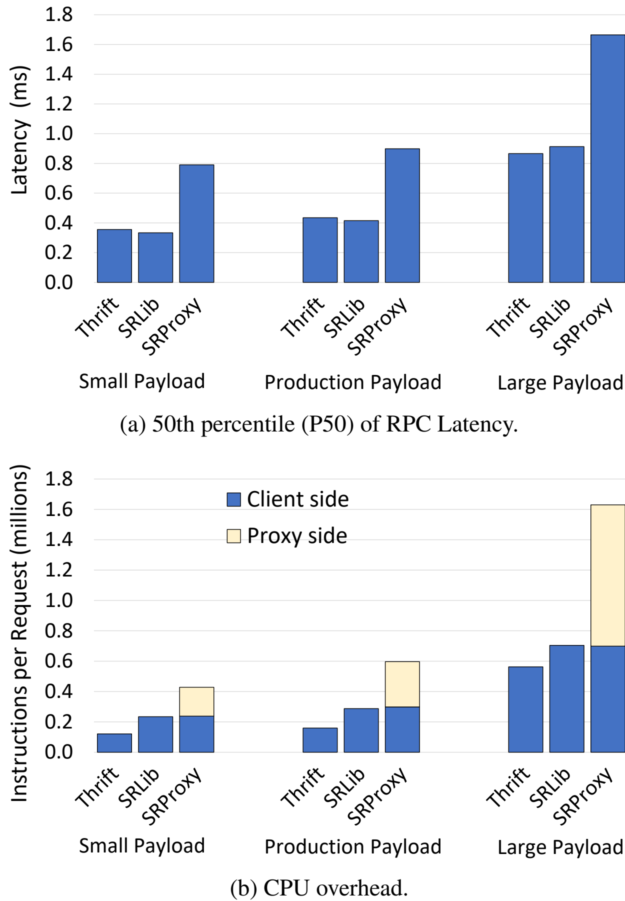
We compare the CPU overhead of SRLib and SRProxy, and use case studies to illustrate when to use SRProxy.

4.2.1 SRLib versus SRProxy

To quantify the hardware cost, we conduct an experiment to compare three RPC setups: 1) *SRLib*, where a client uses SRLib to route requests to a simple service running on 10 machines; 2) *SRProxy*, where a client sends requests to a remote SRProxy, which forwards requests to the servers; and 3) *Thrift*, where a barebone client hard-codes a most efficient way to randomly select one of the 10 servers and invokes it using the Thrift [51] RPC protocol. SRLib and SRProxy's internal implementation also use Thrift but add extra logic atop it. Therefore, Thrift represents the lower-bound baseline.

In all three setups, the RPC connections are 100% reused to avoid the connection establishment overhead. All servers used in the experiment are located in the same region to minimize the impact of network latency. We use three RPC payload sizes. The *Production* size uses requests and responses of $5.4KB$ and $6KB$, respectively, which are the average sizes of payloads in production. The *Large* and *Small* sizes use payloads that are $10x$ or $\frac{1}{10}x$ of the production payload size, respectively. We report in Figure 10 the end-to-end RPC latency and the total CPU instructions executed across the client, proxy (if used), and server when processing one RPC.

Using production-sized payloads, compared with Thrift, SRLib and SRProxy consume 80% and 273% additional CPU cycles, respectively. The overhead is high because this experiment is set up to measure almost the worst case of SRLib and SRProxy. Since the payload's data type is a trivial string, serialization and deserialization in Thrift take little time. Moreover, both the RPC client and server do not do any processing. Overall, this setup minimizes all other overhead in order to show the worst-case setup for SRLib and SRProxy. In our production environment, when aggregated across all workloads



(a) 50th percentile (P50) of RPC Latency.

(b) CPU overhead.

Figure 10: Comparison of latency and CPU overhead across three setups: the Thrift baseline, SRLib, and SRProxy.

running on all servers, SRLib consumes 36% additional CPU cycles compared to Thrift. This is much lower than the 80% overhead observed in this worst-case experiment.

For the SRProxy setup using production-sized payloads, the CPU consumption is evenly split between the client and proxy. Thrift and SRLib have almost identical latency, whereas SRProxy's latency is 107% higher. Using large-size payloads, the relative overhead of SRLib and SRProxy becomes smaller. Compared with Thrift, SRLib and SRProxy consume additional 25% and 190% CPU cycles, respectively.

This experiment shows that, across the RPC client and proxy, the SRProxy setup in total consumes more than twice the amount of CPU cycles as the SRLib setup. In our production environment, we use thousands of SRProxy machines to route 1.1% of the total RPC requests, which generate only 0.1% of the total RPC data transferred. The remaining RPC requests are routed by SRLib. If we were to completely switch from SRLib to SRProxy and route 100% of the RPC traffic by SRProxy, we would need hundreds of thousands of additional machines for SRProxy.

In the SRProxy setup with production-sized payloads, the split between CPU instructions executed in the kernel and user space is 26% versus 74%. This indicates that even if the kernel overhead could be entirely eliminated through methods like zero-copy data forwarding, it would still be insufficient to significantly reduce the proxy's overhead. Moreover, the proxy cannot perform zero-copy data forwarding because it needs to manage encryption and identity.

Using small and production-sized payloads, SRLib's latency appears to be slightly better than Thrift, but since the standard deviation is high, the small difference is mostly caused by measurement noises in our production network that serves many other production services. Lastly, we would expect to see less CPU cycles consumed by the client side of the SRProxy setup compared with the client side of the SRLib setup, as the former does less routing work. However, the difference is insignificant in this experiment because the SRLib code path is slightly better optimized by our years of investments in it.

4.2.2 Case Study of When to Use SRProxy

As shown in Figure 5, a shared SRProxy improves connection reuse, which potentially can reduce the latency of cross-region RPCs at the expense of extra hardware to host SRProxy. The tradeoff depends on the business value of the reduced latency and the cost of the extra hardware. In practice, we always carefully evaluate our customer's request of using SRProxy case by case. We present several case studies below.

E-Comm. E-Comm is a sharded ranking service used in e-commerce. Due to its tight service-level objective (SLO) for latency, all of its shards were replicated to every region to enable local access. We analyzed its traffic pattern and found that by allowing only 5% of its traffic to go across regions, we could avoid replicating 33% of its shards in every region. This would lead to significant hardware savings but at the expense of increased latency. In Figure 11, we compared E-Comm with and without SRProxy and found that SRProxy improved cross-region connection reuse and reduced the P90 latency from about 325ms to about 150ms. E-Comm's maximum per-region RPS is about 300K, which can be handled by 4 SRProxy machines since each SRProxy machine can handle about 87K RPS. In practice, about 10 SRProxy machines are needed to provide sufficient buffers for failure and unexpected

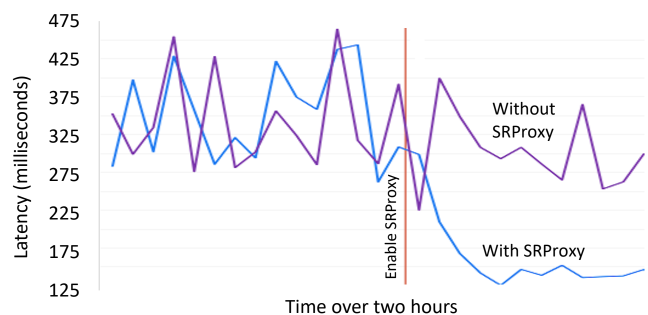


Figure 11: E-Comm's P90 latency with and without SRProxy.

load spikes. After the evaluation, we decided to enable SRProxy for E-Comm because the 10 SRProxy machines per region would allow us to save 33% of E-Comm's hardware capacity by routing 5% of its traffic across regions while still keeping its latency within its SLO.

Key-value store. This distributed key-value store has 1.5 million data shards. Accordingly, its service-discovery information includes a large shard map for these 1.5 million shards (see an example in Figure 7). It takes a lot of memory on the key-value store's clients to cache this large service's full service-discovery information. We evaluated enabling SRProxy to offload the service-discovery cache from the client machines to SRProxy, and noted that it saved on the client machines 250MB memory at P99. Moreover, SRProxy helped with connection reuse and thus latency. The clients have poor connection reuse due to the huge fanout of requests to many different shards hosted by different servers. Shared SRProxies could drastically improve connection reuse and reduce latency by 27% on average. However, due to the key-value store's high RPS, it would need 1,500 SRProxy machines. Eventually, we decided that the cost would not sufficiently justify the benefits and hence did not use SRProxy to route its traffic.

4.3 Load Balancing

SR performs load balancing both within a region and across regions. We evaluate both scenarios in this section.

4.3.1 Same-Region Load Balancing

To evaluate same-region load balancing, we selected 15 representative services that produce significant traffic within a region. 10 of these services are unsharded and 5 are sharded. We measured each service's average production load (pending requests for unsharded services and CPU usage for sharded services) across its servers and normalized the load by its mean. To evaluate whether the load evenly spreads across

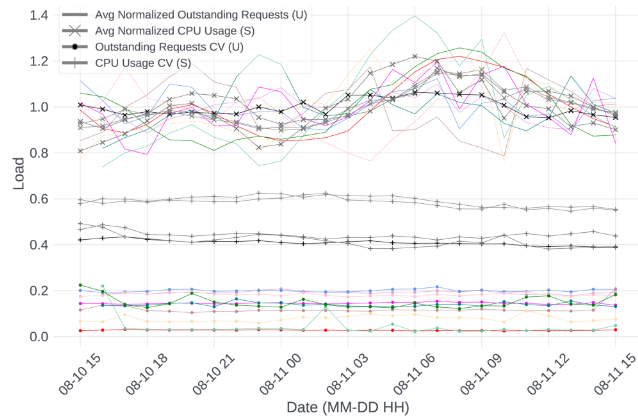


Figure 12: Load balancing within a region for unsharded (U) and sharded (S) services. The top group shows the normalized average load and the bottom two groups show the load's coefficient of variation (CV) across servers.

a service's different servers, we calculated the coefficient of variation (CV) for each service. Figure 12 summarizes the results. We observe that the load is concentrated within a narrow band for all services. Across all 15 services, the median CV is low $P50_{CV} = 0.18$ and $P95_{CV} = 0.6$. In particular, the CV for unsharded services is always low ($P50^u_{CV} = 0.13$ and $P95^u_{CV} = 0.20$), indicating that SR effectively balances the load across their servers.

The CV for sharded services is higher ($P50^s_{CV} = 0.44$ and $P95^s_{CV} = 0.61$), indicating that the load is less balanced. This is because some shards are hot (receiving a lot of traffic) while others are cold (receiving little traffic), due to the nature of data stored in the shards. As a result, even if SR perfectly routes RPC requests to different replicas of the shards, the load on the servers that host different shards may still be unbalanced. To further balance the load, it may be necessary to migrate shard replicas across containers and/or create additional replicas of the hot shards. However, these operations may have a high overhead, so our shard manager [34] performs these operations only enough to prevent server overload without attempting to perfectly balance the load. Overall, these data show that SR can use a single service mesh to balance load for both sharded and unsharded services.

4.3.2 Cross-Region Load Balancing

Locality ring. A service's locality-ring configuration (§ 3.4.1) guides SR to route requests to nearby servers when appropriate. To assess its effectiveness, we measure P90 latencies for requests that fall into different locality rings. Most services (63.8%) use SR's default locality-ring configuration: $\left[\textit{same region} \mid \textit{neighboring regions} \mid \textit{same continent} \mid \textit{global}\right]$. Interestingly, 15.4% of services simply set their locality ring as $\left[\textit{global}\right]$, meaning that they have no locality preference. Most of these services are not user facing and not sensitive to latency, but instead care more about availability. 9.7% of services set their locality ring as $\left[\textit{same region} \mid \textit{global}\right]$, meaning that they prefer a request being served in the local region, but if that's impossible, they prefer the request being served by a more lightly loaded server in any region, as opposed to a more heavily loaded server in a nearby region. The remaining 11.1% of services use their own custom locality-ring configuration.

We found that the P90 latency is 12/83/201/262 ms for requests that are served by servers in the *Region / NeighboringRegions / Continent / Global* rings, respectively. This confirms the correct behavior that the inner rings exhibit lower latencies than the outer rings. Moreover, the latency jump at each expanded ring level is significant, indicating that fine-grained locality management is helpful. Initially, our default ring configuration was $\left[\textit{same region} \mid \textit{same continent} \mid \textit{global}\right]$. As more datacenter regions were added to our infrastructure, the latency difference between regions in the same continent became more significant. Then we were able to easily introduce a new ring level, *neighboring regions*, thanks to the flexibility offered by locality rings.

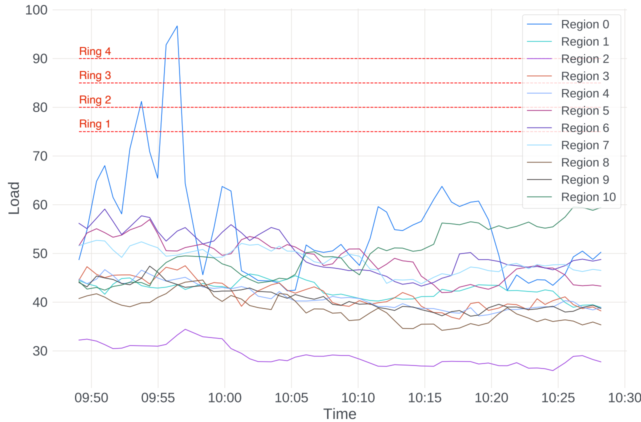
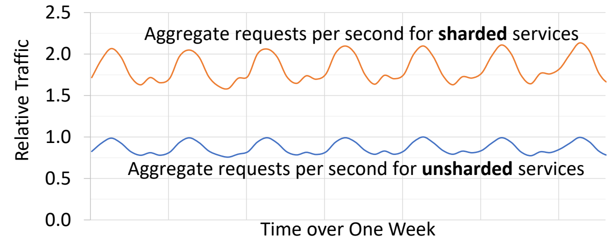
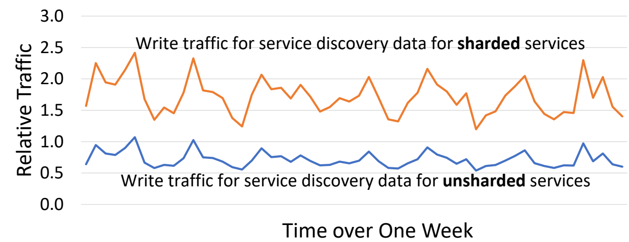


Figure 13: xRS shapes a service's traffic across several regions to prevent overloads.



(a) Data plane (i.e., application RPCs).



(b) Control plane (i.e., service-discovery updates).

Figure 14: Traffic for unsharded and sharded services, excluding memcache.

Cross-region load spillover. xRS computes a cross-region routing table per service to guide L7 routers' routing decisions (§3.4.1). To evaluate its effectiveness, we choose a service that does newsfeed fetching and ranking for one of our main products. The service uses the following locality-ring thresholds, $[\texttt{ring}_1\texttt{:}75\% \mid \texttt{ring}_2\texttt{:}80\% \mid \texttt{ring}_3\texttt{:}85\% \mid \texttt{ring}_4\texttt{:}90\%]$, where the second number (e.g., 75%) in the pairs is a load threshold. It means that if the measured load in an inner ring exceeds the threshold, xRS should compute a new routing table to shift some traffic from the inner ring to the next-level outer ring in order to reduce load in the inner ring. The load metric for the service is CPU utilization averaged across the service's all servers in a region.

In Figure 13, we report a real incident that happened to the service in production, which was not conducted by us just for the sake of experiment. The figure shows the average load across several regions for the service in the span of 40 minutes. We observe that xRS was able to shift traffic to maintain the load well below the ring_1 load threshold of 75% for most regions except Region 0 during a short spike.

At 09:53 AM, Region 0 exhibited high load (81.2%), which exceeded its ring_2 load threshold (80%). xRS evaluated shifting some traffic to Region 0's ring_2 regions (i.e., Regions 2, 8, and 10) and chose Region 2 as the target because it had the lowest load. xRS calculated that by shifting some traffic from Region 0 to Region 2, the load of Region 0 would fall below the load threshold. This resulted in a new routing table which reduced $P_{0,0}$ traffic by 5.35% and set $P_{0,2} = 5.35\%$, meaning that 5.35% of requests originating from Region 0 should be routed to Region 2. Then, the load of Region 0 fell to 70.9% and subsequently to 65.47% at 09:54 AM. The corresponding load increase in Region 2 was insignificant because the service had a large capacity footprint in Region 2.

At 09:55 AM, the load of Region 0 spiked again to 92.83% and then to 96.69% at 09:56 AM, which was above the service's ring_4 threshold (90%). In response, xRS reduced $P_{0,0}$ by 12% at first (99.24% → 86.92%) and then by another 13% (86.92% → 74.38%). The removed traffic was again added to Region 2 alone as it was the least loaded region among all regions in Region 0's ring_4. $P_{0,2}$ increased first from 0.76% to 13.08% and then from 13.08% to 25.62%. These adjustments helped the load in Region 0 to drop to 64.34% and the region became healthy again at 09:57 AM.

Overall, the whole process above was fully automated by xRS without any manual intervention. It demonstrates that xRS is effective in dynamically managing cross-region traffic.

4.4 Sharded Services

Currently, our fleet runs hundreds of sharded services [34]. Although they only account for about 3% of our tens of thousands of services, they generate more traffic than the other 97% unsharded services, because many sharded services are among our largest and highest traffic services. Specifically, most of the largest services in Figures 8 and 9 are sharded, and the two services studied in §4.2.2 are both sharded. To give a sense of scale, our fleet has millions of containers for unsharded services, and hundreds of millions of shard replicas.

Figure 14(a) shows that the aggregate RPS for all sharded services is 212% of the aggregate RPS for all unsharded services. Our memcache [38] is sharded and has the highest RPS among all our services, but it is excluded from the comparison in Figure 14(a) to avoid overshadowing other services. The RPS for memcache alone is 975% of the aggregate RPS for all unsharded services. Although memcache has a high RPS, each of its requests is very lightweight and hence memcache servers do not account for a large fraction of our fleet capacity. Figure 14(b) shows that the aggregate control-plane traffic to

update service-discovery information for all sharded services is 240% of the aggregate traffic for all unsharded services. This is because sharded services more frequently migrate shards across servers to balance load.

Overall, the traffic for sharded services overwhelmingly dominates both the data plane and the control plane of our service mesh, which highlights the importance of providing first-class built-in support for sharded service in a service mesh. Excluding memcache, RPCs for sharded services account for 68% of all RPCs, rising to 92% when memcache is included (as our memcache is sharded). Despite the importance of sharded services, all existing general-purpose service meshes ignore sharded services and exclusively focus on unsharded services. Our key insight in supporting sharded and unsharded services in a single framework is to define a sharding abstraction between SR and services to enforce separation of concerns so that SR can route traffic without knowing a service's internal application logic.

5 Limitations of SRLib and Our Solutions

In this section, we discuss several limitations of SRLib and how we address them.

Dynamic policy updates. In addition to load balancing, SR offers a large set of service-mesh features such as overload protection [39], observability [1], distributed tracing [32], and encryption. These features are managed through dynamic policy updates, which need to be executed by L7 routers in near-real-time. Without good tooling support, deploying policy updates for a library embedded in applications could be harder than for standalone sidecar proxies. At Meta, this problem is solved by a powerful configuration management system called Configerator [52]. The policies for both SRProxy and SRLib are managed in the same way. When a policy changes, Configerator propagates the change and sends an upcall to SRLib embedded in applications. SRLib then applies the new policy immediately, without restarting the application.

Source code modification. One disadvantage of SRLib is that it requires code changes to services. Traditional RPCs use an IP address and a port number to obtain an RPC client, whereas SRLib obtains an RPC client using a service name. The code example below shows that it is straightforward to modify a traditional RPC framework to use SRLib.

```
TraditionalRPCClient *cln = get_client(IP, port);
cln->foo();  // Invoke RPC for foo().
SRClient *cln2 = SR_get_client("service_name");
cln2->foo(); // Invoke RPC for foo().
```

Moreover, source-code modification related to RPC is not unique to SRLib, and is widely adopted by hyperscalers. Regardless of how routing is done, as long as a hyperscaler's RPC framework does not entirely rely on the standard but slow DNS for service discovery, they have to modify their application code to integrate with their custom service-discovery system. Examples of this include Google's Borg Name Service [55], Netflix's Eureka [17], LinkedIn's Rest.li Dynamic Discovery [49], Twitter's Finagle [19], Uber's Hyperbahn [27], and Airbnb's Synapse [3]. The prevalence of custom service-discovery systems, which often require source-code modifications to use, suggests that this approach is practical as long as the changes are simple and limited to RPC's narrow interface.

Library code deployment. Deploying a new version of SRLib is more difficult than deploying a new version of a sidecar proxy. This is because SRLib is compiled into tens of thousands of services, each with its own deployment schedule. Furthermore, in theory, it is possible that some services may not be updated for a long time, resulting in their continued use of an outdated version of SRLib. At Meta, this problem is solved by a powerful continuous software deployment tool called Conveyor [25]. With the help of Conveyor, 97% of the services at Meta are configured to deploy automatically without manual intervention, whether it is on a daily or weekly basis, or whenever a code update successfully passes all tests. Moreover, due to reasons beyond SR, it is a company mandate for all services to be deployed regularly, which ensures that services run with a recent version of SRLib.

Bugs in SRLib. If SRLib's new code has a bug, it can be difficult to instantly roll back all services. To mitigate this risk, every major code change or new feature in SRLib is gated by a configuration parameter that can be toggled live in production via Configerator [52], as shown in the example below, without requiring a software deployment or process restart.

```
// Introduce a new FEATURE_X in the SRLib code.
if (check_gate(FEATURE_X)) {
  // New code path...
} else {
  // Old code path...
}
```

In the example above, when `FEATURE_X` is updated on a central server via Configerator, the new parameter value is propagated to all SRLib instances within seconds. SRLib's next invocation to `check_gate(FEATURE_X)` returns the updated parameter value and switches the code path accordingly, without requiring a restart of the application process.

After the above new code is released into production, *check_gate(FEATURE_X)* defaults to false, as if the new code path does not exist. Configerator then manages a canary testing process where it selectively enables the new code path for a small number of replicas of a few services by setting *check_gate(FEATURE_X)* to true. If the test is successful, the new code path is gradually enabled for more services. If a bug is encountered, *FEATURE_X* can be instantly disabled for all services via a configuration change. Overall, incremental rollouts of new SRLib code gated by configuration changes allow us to mitigate the risk of SRLib bugs.

Summary. At Meta, managing widely deployed libraries (WDL) such as SRLib is largely a solved problem thanks to the help of Configerator [52] and Conveyor [25]. These tools also help manage about a dozen other WDLs, so the problem is not unique to SRLib. However, we acknowledge that, even with the help of Configerator and Conveyor, it is still more challenging to develop, deploy, and manage SRLib than sidecar or remote proxies because SRLib is linked into every service. Although SR supports both SRProxy and SRLib, we prioritize the cost savings of hundreds of thousands of machines that come with the routing-library approach, over the simplicity that comes with the proxy approach. Our experience in production demonstrates that the routing-library approach is not only cost-effective but also practical, even in highly complex environments, despite its challenges.

6 Related Work

There is an array of works from both academia and industry discussing routing and load balancing in datacenter environments, at either layer-3/4 [5,8,12,14,18,21,40,44,46] or layer 7 [3, 5, 15, 19, 20, 23, 26, 30, 36, 37, 43, 48]. Layer-3/4 load balancers can be implemented either in hardware [8,21,40] or in software [5, 14, 22, 28, 44, 46, 48]. As a layer-3 solution, anycast [56] can route requests to nearby servers, but it does not consider the servers' dynamic load. As shown in Figure 14, the majority of our traffic is for sharded services, which cannot be handled by these layer-3/4 solutions as they do not understand application shards.

More relevant to SR are layer-7 (L7) service-mesh solutions that route requests across microservices. L7 routing can inspect application-level information, enabling more advanced load balancing. L7 routing can be performed by a group of dedicated proxies [3,5,15,20]. However, using remote proxies comes with significant latency and hardware costs, so SR limits the use of SRProxy to around 1% of its traffic, only for services that can benefit the most from connection reuse.

More related to SRLib, which routes 99% of our traffic, are service meshes that distribute L7 decisions closer to the clients. eBPF [35] is efficient but is limited in its L7 capabilities. For example, Cilium [29]'s eBPF program can only handle L3/L4 protocols, and it still needs to use a sidecar proxy to handle L7 protocols. RPC frameworks such as Thrift [51], gRPC [23], and Finagle [19] are the foundations of service meshes, but they do not offer the complete capabilities needed for a geo-distributed service mesh, such as global-traffic-aware routing.

To address these limitations, more complex service meshes [15, 30, 36, 37] have been proposed. Envoy [15] is typically deployed as a sidecar proxy, and Istio [30] provides a control plane to manage Envoy proxies. We compare different service meshes in Table 2 and show that the sidecar approach is easy to deploy, but increases latency and incurs significant hardware costs. Zhu et al. [57] show that Istio adds 92% extra CPU usage and increases the latency by 185% [57]. mRPC [7] confirms that a sidecar increases the P99 RPC latency by 180% and decreases throughput by 44%. SR takes the routing-library approach to avoid the overhead of a proxy.

mRPC [7] eliminates the double marshaling overhead of the sidecar approach, by using shared memory to communicate between the application and the sidecar and by not performing marshaling in the application. However, this approach requires modifying applications to allocate memory for RPC arguments from a heap in shared memory. This can be difficult since memory allocations tend to scatter throughout an application and sometimes occur in system libraries such as `strdup()` that cannot be easily modified.

While Istio offers locality-aware routing based on static rules [31], SR dynamically computes a per-service global routing table based on global traffic. Google Slicer [2] supports service discovery for sharded services, but this function is not offered by the underlying service mesh out of the box.

7 Conclusion

We presented Meta's global service mesh, called *ServiceRouter* (SR). SR differs from other publicly known service meshes in several significant ways. First, SR scales significantly beyond previously published work, currently processing tens of billions of requests per second. This is achieved by massively replicating the routing information base (RIB) to guide L7 routers to self-configure and self-manage in a decentralized manner. Second, SR minimizes hardware costs by providing the service-mesh function out of an embedded routing library for 99% of its traffic, in contrast to the common approach of using sidecar or remote proxies alone. Third, SR introduces the concept of locality rings to simultaneously minimize RPC latency and balance load across geo-distributed datacenter regions. Finally, SR supports both sharded and replicated services through a common underlying routing framework.

Our ongoing work is focused on improving global routing in accordance with global capacity management [16] and enhancing overload protection to ensure services gracefully degrade in the event of large-scale disasters [39].

Acknowledgments

This paper presents 11 years of work by past and current members of several teams at Meta, including ServiceRouter, CSLB, SMC, and Falcon. In particular, we would like to call out the current members of the ServiceRouter team who are not on the author list: Akrama Baig Mirza, Bo Huang, Emanuele Altieri, Kenny Lau, Lijie Tang, Mikhail Shatalov, Nan Su, Nitesh Kant, Scott Diao, Tao Chen, Wei Song, and Weilun Wang. We thank all reviewers for their insightful comments, Shie Erlich for the support, as well as Andrii Grynenko, Rahul Gokul, and Stepan Palamarchuk for their contributions to some ideas presented in the paper.

References

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: Diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 739–753, 2016.

[3] Airbnb Synapse. https://github.com/airbnb/synapse.

[4] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. A survey of load balancing in cloud computing: Challenges and algorithms. In *2012 second symposium on network cloud computing and applications*, pages 137–142. IEEE, 2012.

[5] AWS Elastic Load Balancing. https://aws.amazon.com/elasticloadbalancing/.

[6] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[7] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 141–159, Boston, MA, April 2023. USENIX Association.

[8] Reuven Cohen, Matty Kadosh, Alan Lo, and Qasem Sayah. LB Scalability: Achieving the Right Balance Between Being Stateful and Stateless. *IEEE/ACM Transactions on Networking*, 30(1):382–393, 2021.

[9] Ben Coleman. Rendezvous Hashing Explained, 2020. https://randorithms.com/2020/12/26/rendezvous-hashing.html.

[10] Consul. https://www.consul.io.

[11] Robert B Cooper. Queueing theory. In *Proceedings of the ACM'81 conference*, pages 119–122, 1981.

[12] Alejandro Forero Cuervo. Load Balancing in the Datacenter, 2016. https://sre.google/sre-book/load-balancing-datacenter/.

[13] Michael Dahlin. Interpreting stale load information. *IEEE Transactions on parallel and distributed systems*, 11(10):1033–1047, 2000.

[14] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[15] Envoy Proxy. https://www.envoyproxy.io/.

[16] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. Global Capacity Management with Flux. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[17] Eureka. https://github.com/Netflix/eureka.

[18] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's Software-Defined Networking Control Plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 83–98, 2021.

[19] Finagle: A Protocol-Agnostic RPC System. https://blog.twitter.com/engineering/en_us/a/2011/finagle-a-protocol-agnostic-rpc-system.

[20] Rohan Gandhi, Y Charlie Hu, and Ming Zhang. Yoda: A highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[21] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.

[22] Introducing the GitHub Load Balancer. https://githubengineering.com/introducing-glb/.

[23] gRPC. https://grpc.io/.

[24] gRPC Lookaside Load Balancer. https://github.com/markitdigital/grpc-lookaside.

[25] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[26] HAProxy. https://www.haproxy.com/.

[27] Hyperbahn. https://github.com/uber-archive/hyperbahn.

[28] IPVS Software - Advanced Layer-4 Switching. http://www.linuxvirtualserver.org/software/ipvs.html.

[29] Isovalent. Cilium Service Mesh—Everything You Need to Know, 2022. https://isovalent.com/blog/post/cilium-service-mesh/.

[30] Istio. https://istio.io/.

[31] Istio Locality Load Balancing. https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/.

[32] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 34–50, 2017.

[33] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[34] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.

[35] Idit Levine, Christian Posta, and Yuval Kohavi. eBPF for Service Mesh? Yes, but Envoy Proxy is here to stay, 2021. https://www.solo.io/blog/ebpf-for-service-mesh/.

[36] Chien-Chih Liao, Pawel Krolikowski, and Sangeeta Kundu. Better Load Balancing: Real-Time Dynamic Subsetting, 2022. https://www.uber.com/blog/better-load-balancing-real-time-dynamic-subsetting/.

[37] Linkerd. https://linkerd.io/.

[38] Memache. https://memcached.org/.

[39] Justin Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Nazim Uddin, Chad Nachiappan, Sari Tran, Shuyang Shi, Tina Luo, Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard. Defcon: Preventing Overload with Graceful Feature Degradation. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[40] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[41] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[42] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[43] Netflix Ribbon. https://github.com/Netflix/ribbon.

[44] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, 2018.

[45] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, Malgorzata Steinder, Asser Tantawi, and Alaa Youssef. Managing the response time for multi-tiered web applications. *IBM TJ Watson Research Center, Yorktown, NY, Tech. Rep. RC23651*, 2005.

[46] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[47] Performance and Scalability of Istio. https://istio.io/latest/docs/ops/deployment/performance-and-scalability/.

[48] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[49] Rest.li Dynamic Discovery. https://linkedin.github.io/rest.li/Dynamic_Discovery.

[50] Daniel E Rivera, Manfred Morari, and Sigurd Skogestad. Internal model control: PID controller design. *Industrial & engineering chemistry process design and development*, 25(1):252–265, 1986.

[51] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 5(8):127, 2007.

[52] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343, 2015.

[53] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 787–803. USENIX Association, 2020.

[54] David Thaler and Chinya V Ravishankar. A name-based mapping scheme for rendezvous. In *Technical Report CSE-TR-316-96, University of Michigan*, 1996.

[55] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM European Conference on Computer Systems*, 2015.

[56] Scott Weber and Liang Cheng. A survey of anycast in ipv6 networks. *IEEE Communications Magazine*, 42(1):127–132, 2004.

[57] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting Service Mesh Overheads. In *arXiv preprint arXiv:2207.00592*, 2022. https://arxiv.org/pdf/2207.00592.pdf.