服务网格设计与优化总体研究

聂瑞[1)]

[1)](华中科技大学计算机学院 武汉市 中国 430074)

摘 要 目前，服务网格被广泛应用于由大量微服务组成的数据中心应用中。服务网格是一个专用的基础层架构，用于管理分布式应用程序中各个微服务间的通信。通过将网络通信的复杂性抽象化，服务网格使开发人员能够专注于应用程序逻辑，而不必处理网络代码的复杂性。它提供了一种一致且灵活的的处理跨服务通信的方式，并允许实现高级的流量管理策略、安全策略和可观测机制。随着互联网连接速度的增加和网络设备规模的扩大，服务网格面临几个问题：1，如何将服务网格扩大到更大规模 2，如何提高通信效率 3，如何拓宽使用场景 4，如何降低硬件开销。围绕这几个问题，本文介绍分析了近三年共 5 篇服务网格相关论文。

关键词 服务网格 远程过程调用 控制平面 负载均衡 区域化 分片服务

A Survey on Service Mesh Design and Optimization

Rui Nie[1)]

[1)](Huazhong University of Science and Technology, Wuhan, China,430074)

Abstract Nowadays, Service mesh is widely used in data center applications consisting of a large number of microservices. The Service mesh is a specialized base layer architecture for managing communication between individual microservices in a distributed application. By abstracting the complexity of network communication, Service mesh enables developers to focus on application logic without having to deal with the complexity of network code. It provides a consistent and flexible way to handle cross-service communication and allows for the implementation of advanced traffic management policies, security policies, and observability mechanisms. As the speed of Internet connections increases and the size of network devices expands, Service mesh faces several problems: 1, how to scale up Service mesh to larger sizes 2, how to improve the communication efficiency 3, how to broaden the usage scenarios 4, how to reduce the hardware overhead. Focusing on these issues, this paper presents and analyzes a total of five service mesh related papers in the last three years.

Key words Service Mesh; Remote Procedure Call; Control Plane; Load Balancing; Regionalization; Sharded Service

1 引言

现在的数据中心应用通常由许多相互联系的微服务（Microservice）组成。对持续集成和持续部署的需求不断提升，催生了微服务架构。一个应用程序被分解为一组服务，每个服务分别独立进行开发和部署。

服务网格（Service Mesh）是一种在服务间路由（route）RPC（Remote Procedure Call）消息的热门方法。服务网格通常包含一个数据平面和一个控制平面：

- 数据平面（Data Plane）：数据平面指的是部署在每个服务实例旁边的一组 Sidecar 边车代理组成的网络，用于与系统中的其他服务进行通信。它充当服务与网络的中

间人。Sidecar 代理处理入站和出站流量，拦截通信并提供其他功能。

- 控制平面（Control Plane）：控制平面是服务网格的集中管理和配置层。它负责控制和协调 Sidecar 代理的行为。它提供了一个控制平面 API，允许管理员配置流量管理、安全性和可观测性的策略、规则和设置。

图 1-1 给出了传统服务网格结构的一个示例。

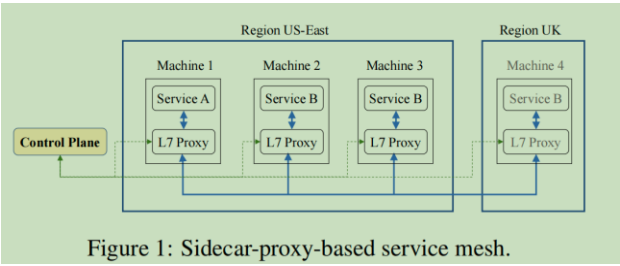

Figure 1: Sidecar-proxy-based service mesh.

图 1-0-1 传统服务网格边车模型

随着互联网连接速度的增加和网络设备规模的增大，服务网格面临几个问题：1，如何将服务网格扩大到更大规模，使其能够包含更多的服务器、跨越更多的地区 2，如何提高通信效率，减少数据流转过程中的不必要处理 3，如何拓宽使用场景，例如支持越来越常见的分片服务 4，如何降低硬件开销。第一点可以从网格结构设计出发。第二点可以从 rpc 通信模式出发。第三点可以从网格功能部件出发。 第四点交织在前面三点中。下面本文介绍分析几篇今年相关论文，探讨这些问题的成因和解决方案。

2 如何将服务网格扩大到更大规模

2.1 控制平面

【Orion】[1]提到，Google 建立的 Orion 控制平面系统是围绕一个模块化的微服务架构设计的，该架构带有一个中央发布-订阅数据库，以实现分布式、紧密耦合的软件定义网络控制系统。Orion 支持基于意图的管理和控制，具有高度可扩展性，并适应全局控制层次结构。

多年来，Orion 已经在生产中趋于成熟，在收敛性能(速度提高了 40 倍)、吞吐量(每秒处理 116 万次网络更新)、系统可扩展性(支持 16 倍大的网络)和数据平面可用性(在 Jupiter 和 B4 中分别减少了 50 倍和 100 倍的不可用时间)方面不断提高，同时保持了高开发速度，每两周发布一次。

Orion 是 Google 的第二代控制平面，负责配置、管理和实时控制 Google 所有的数据中心(Jupiter)、校园和私有广域网(B4)网络。

Orion 的架构包括三部分：Orion 核心，路由引擎和 Orion 应用框架。

Network information base（NIB）是所有 Orion 应用程序的意图存储库。它被实现为一个集中式的内存数据存储，其副本可以在发生故障时重建状态。NIB 与发布-订阅机制相结合，在 Orion 应用程序之间共享状态。外部使用相同的基础结构来收集 NIB 中的所有更改，以方便调试。NIB 必须满足以下要求:

•对外依赖程度低。由于 Orion 编程的网络支持所有高级计算和存储服务，因此它本身不能依赖于高级服务。

•事件排序的顺序一致性。为了简化应用之间的协调，所有应用必须以相同的顺序看到事件。

NIB 由一组 NIB 实体表组成，其中每个实体描述域的本地或外部的其他应用程序或观察者感兴趣的一些信息。实体类型包括:

•已配置的网络拓扑。它们捕获已配置的身份，并绘制各种网络拓扑元素之间的关系图。示例包括端口表、链路表、接口表和节点表。

•网络运行时状态。可以是拓扑状态、转发状态（如 ProgrammedFlow 表）、协议状态（如 LLDPPeerPort 表)、统计信息(如:PortStatistics 表)。

•Orion 应用程序配置。每个应用程序的配置被捕获为一个或多个 NIB 表，例如 LLDPConfig。

Orion 把每个 NIB 实体的模式表示为协议缓冲区消息。提供了一个简单地 RPC API 来操作 NIB 表。

拓扑管理器设置并报告网络数据平面拓扑的运行状态。通过订阅来自交换机的事件，将当前拓扑写入 NIB 中的表，并定期查询交换机的端口统计信息。

流管理器执行流状态协调，确保交换机中的转发状态与 Orion 应用程序计算的预期状态匹配。

Openflow Frontend（OFE）复用连接到 Orion 域中的每一个交换机。

Packet-io 向数据平面发送或接收控制消息。

路由引擎(Routing Engine, RE)是 Orion 的域内路由控制器应用，提供常见的路由机制，如 L3 多路径转发、负载均衡、封装等。

Orion 应用程序框架是每个 Orion 应用程序的基础。该框架确保开发人员使用相同的模式编写应用程序，以便将一个 SDN 应用程序的控制流知识转换到所有应用程序。此外，该框架还提供了所有部署中的所有应用程序所需的基本功能(例如，领导者选举、NIB 连接、运行状况监控)。

可用性是网络和 SDN 控制器的基本特征。Orion 应用程序作为独立的二进制文件运行，分布在网络控制服务器机器上。这确保了应用程序与其他应用程序中的错误隔离(例如，导致崩溃的内存损坏)。

除了隔离之外，在三个不同的物理机器上复制每个应用程序可确保计划(例如维护)和计划外(例如电源故障)中断的容错能力。应用程序框架通过在 leader 选举和应用程序的生命周期回调之上提供抽象来促进复制。

Orion 是服务网格控制平面的一个典型例子，使用 NIB、路由引擎、应用框架三大件来处理服务网格通信。NIB 及相关服务发现、监控机制把有关全局信息的收集、判断和处理等任务从应用程序客户端、服务器剥离开来，从一定程度上降低了扩展服务规模的难度。然而另一篇文章提到了其中的不足。

【Service Router】[2]中提到，传统的服务网格采用传统的软件定义网络方法，使用一个中心控制平面来配置每个边车代理的路由表，见图 1-1。这个中心控制平面具有双重功能，一是产生并维护全局路由元数据，二是管理每个 l7 路由器，无法扩展至超大规模。前文提到的来自 Google 的 Orion 控制平面也直接负责配置、管理和实时控制所有的数据中心，从而在扩展性上不可避免地受到限制。本文把前一个任务继续留给控制平面，而把后一个任务去中心化，整合到 l7 路由器的功能中，让它们自行配置并管理自己。这样中心控制平面就能轻松扩展应用到更大规模。

该文章提出了 Meta 的 Service Router（SR）服务网格模型，如图 2-1 所示。

在 SR 中，顶层的各个控制器完成不同的任务，并独立更新路由信息仓库 Routing Information Base（RIB），而不关心配置或管理底层的 L7 路由器。RIB 的概念和 Orion 的 NIB 概念具有相似性。中间的分布层复制出 RIB 的许多副本，来应对来自数百万 L7 路由器的洪流般的访问请求。初始情况下一个 L7 路由器的路由表是空的，当它接收到指向一个服务的 RPC 请求时，就从某个 RIB 副本中取回该服务的路由信息，并订阅该服务路由信息的后续变动。

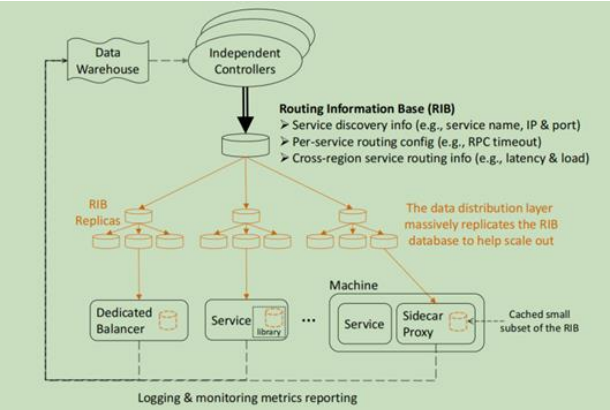

图 2-0-1 Service Router 服务网格结构

2.2 区域化

如何在大量地理分布的数据中心区域之间分配服务能力、降低全局开销？由于微服务架构所产生的复杂服务依赖关系图，以及因地区而异的容量可用性和硬件组合，这个问题变得更加复杂。

从历史上看，区域化一直是通过缓慢的手动过程来解决的，大型服务的所有者直接与云提供商协商容量分配和分配。然而，随着服务数量和规模的不断增长，这些手工流程变得越来越站不住脚，并且通常会导致所有相关方的过度劳动，以及次优结果。

在图 1-1 所示的传统服务网格边车模型中，虽然控制平面能够在不同区域间路由 RPC 消息，但其负载均衡机制盲目选择负载最轻的服务器作为目标，缺少多样化的度量手段，以及对区域位置延迟等负面效果的考虑。

【Service Router】提到，SR 的负载均衡策略基于 pick-2 算法。pick-2 算法从候选池中随机抽取两个服务器，并选择负载较轻的一个作为 RPC 目标。单独的 pick-2 无法适用于跨地理位置分布的服务网格。因此 sr 做了三点工作来对其进行改进：1，随机抽取两个服务器时考虑其区域位置。2，从特定服务器集中随机抽取两个，而非从所有服务器中抽取，以此最大化服务器复用。3，使用适应性方法根据工作性质预估负载。

->位置感知：引入地理环概念，排除高延迟服

务器，并在剩下的服务器中选择。每个服务可以给出一系列延迟递增的“环”，例如[ring1:5ms], [ring2:15ms], [ring3:35ms], [ring4:infinity]。延迟检测服务 lms 周期性检测区域间延迟，客户端通过 cms 得到这些信息，并优先从低延迟环进行服务器取样。

->RPC 连接复用：测试表明建立一个 TLS/TCP 连接需要花费 1.6ms 并占用双端各 14kb 内存，所以复用是有必要的。然而传统的 pick-2 算法让这一点变得困难。所以每个客户端从服务器集的一个很

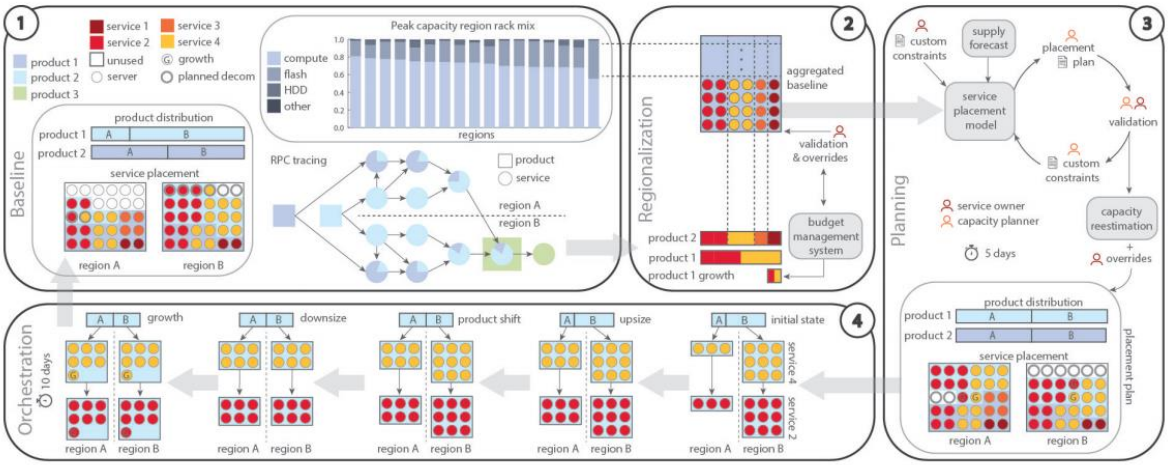

Figure 5: **Flux's end-to-end workflow.**

图 2-0-2 Flux 的端到端工作流示意

小的子集中进行选择。一个挑战是如何让每个客户端选择自己的目标服务器池并让全局负载达到一个平衡状态。SR 中客户端使用 Rendezvous Hashing 来选择自己的 k 个目标服务器。

->适应性负载预估：SR 默认使用待处理请求数量来衡量服务器负载，同时也支持 CPU、GPU、磁盘占用率等度量。一个客户端有两种方式确定一个服务器的负载，一是在确定最终目标服务器之前询问候选服务器，这样会带来额外开销。二是让服务器在响应报文中包含负载信息然后缓存在本地以供后用，这样可能会使用过期失真的负载信息造成负载不均。为取长补短，SR 让服务器在每条响应中包含当前负载信息，客户端只有在发送请求前发现缓存的负载记录非常新鲜才会使用。记录不新鲜时，如果询问服务器负载的耗时明显低于服务器处理一条请求的耗时，那就先询问再发送请求。否则，随机从两个候选服务器中选择一个。

【Flux】[3]提到，Meta 建立了一个叫做 Flux 的系统来实现容量分区的自动化，将其从自下而上的手动过程转变为自上而下的自动化过程。Flux 使用 RPC 跟踪来识别服务容量模型，并使用这些模型来计算跨数十种产品、涉及数百万台服务器的数千种服务的最佳联合容量和流量分配计划。这些计划是由一个系统精心策划的，该系统在数十个地区持续安全有效地重新平衡服务能力和产品流量。

图 2-2 为 flux 的端到端工作流示意图。Flux 首先通过 rpc 跟踪实现产品到服务的容量归属，然后进行服务能力与产品流量的联合区域化，最后进行全局容量编排并执行。

Flux 为每个区域定义了一个基线，将每个服务的峰值容量足迹按部分归因于不同的产品。该基线通过组合另外两个基线产生：

1，容量使用基线。在每台服务器上运行分析器，生成一个数据集，将资源使用归因于特定服务。每分钟采样一次。涵盖不同资源类型，如 CPU 和 SSD。

2，需求基线。使用抽样 RPC 跟踪来重建由每个服务处理的请求的调用图。确定了一组产品网关，作为每个产品的流量入口点。到达这些网关的流量全局且可独立路由，通过 Meta 的共享流量管理系统管理。

Flux 通过建模一个分配问题来计算联合服务容量和交通分区计划。该问题由 MIP 混合整数规划

求解器解决。分配问题约束：

1，每个地区的产能分配不可超过其可用供应。

2，各业务需要满足的全球容量需求。同时还要考虑服务提供者的额外要求。

编排器负责：

1，按照正确以来顺序执行服务和交通分配。

2，持续监测产品水平和服务水平指标，确保它们正常。

3，根据需要把异常相关操作委托给人工操作员。

4，执行负载测试来验证编排。

3 如何提高通信效率

【Service Router】提到，SR 使用传统的边车（Sidecar）和远程代理（Remote Proxy）模式路由 1% 的 RPC 请求，剩下的 99% 通过直接链接到服务执行环境（Service Executable）的名为 SRlib 的路由库直接在客户端和服务端间路由。由于大幅减少了和代理交互的额外程序，这样做显著降低了硬件开销。

现有的服务网格使用边车代理来传递请求，这样会带来额外的路由环节，例如数据序列化和反序列化。SR 通过 SRlib 来提供服务网格的相关功能，去除了对代理的依赖。不过这种方式需要更改服务源代码，这在有些情况下不太能做到。比如用 erlang 编写的服务不能链接 SRlib。

为满足服务的多样化需要，SR 中同时存在不同类型的 L7 路由器，包括 Istio-style Sidecar Proxies, AWS-ELB-style Dedicated Load Balancers, gRPC-style Lookaside Load Balancers, 如果把所有对 SRlib 的使用转变为代理，需要新增加数十万台机器。

每个机器上都运行有一个 RIB 守护程序，维护该机器上每个 RPC 客户端所需要的部分 RIB 所组成的 RIB 子集，称为 mini-RIB。SRlib 向 RIB 守护程序请求某个服务的路由信息，rib 守护程序从 某个 RIB 副本拉取该信息并保存到本地机器磁盘，且订阅该服务的路由信息更新，最后把这部分元数据送给 SRlib，SRlib 同样向 RIB 守护程序订阅该服务的路由信息更新，并把路由信息缓存在机器内存中，这样就不用每次都请求 RIB 守护程序。

当一个服务发生改变（创建，删除，转移等）时，集群管理器通知 全局注册系统 Global Registry System（GRS） 更新 RIB。更新立即被推送到所有 RIB 副本，进而推送到每个订阅了该服务的 RIB 守护程序。每个包含该服务的区域有各自的集群管理器，这些集群管理器通知 GRS 更新该服务的同一条服务注册记录，因而每个 RPC 请求可能被路由到不同的区域的副本。

集群管理器 Cluster Manager 负责侦测服务所在服务器的异常状况，并通过 RIB 通知给客户端。

Service Router 把路由库链接到执行环境来提高通信效率，另一篇文章则针对更精细的 RPC 调用提出一种新的管理方法。

【RPC as MSS】[4]提到，远程过程调用(RPC)是云计算中广泛使用的抽象。程序员为每个远程过程指定类型信息，编译器生成链接到每个应用程序的存根代码，以便将参数编组和反编组到消息缓冲区中。然而，越来越多的应用程序和服务操作团队需要对服务之间的 RPC 流具有高度的可见性和控制，这导致许多安装使用边车或服务网格代理来实现可管理性和策略灵活性。这些边车通常涉及检查和修改 stub 编译器刚刚精心组装的 RPC 数据，从而增加了不必要的开销。此外，升级各种应用程序 RPC stub 以使用高级硬件功能(如 RDMA 或 DPDK)是一个漫长而复杂的过程，并且通常与边车策略控制不兼容。

该文章提出、实现并评估了一种新的方法 mRPC，其中 RPC 编组和策略执行作为系统服务而不是作为链接到每个应用程序的库来完成。应用程序像以前一样向 RPC 系统指定类型信息，而 RPC 服务执行策略引擎并仲裁资源使用，然后编组针对底层网络硬件功能定制的数据。

mRPC 也支持实时升级，因此策略和编组代码都可以透明地更新到应用程序代码。与使用 Sidecar 相比，mRPC 将标准微服务基准 DeathStarBench 的速度提高了 2.5 倍，同时具有更高水平的策略灵活性和可用性。

随着基于 RPC 的分布式应用程序扩展到大型、复杂的部署场景，越来越需要改进 RPC 通信的可管理性。该文章将管理需求分为三类:1)可观察性:提供详细的遥测，使开发人员能够诊断和优化应用程序性能。2)策略执行:允许运营商将自定义策略应用于

RPC 应用程序和服务(例如，访问控制，速率限制，加密)。3)可升级性:支持软件升级(例如，bug 修复和新功能)，同时最大限度地减少应用程序的停机时间。

一个自然要问的问题是:是否有可能在不更改现有 RPC 库的情况下添加这些属性?对于可观察性和策略执行，最先进的解决方案是使用 Sidecar(例如 Envoy 或 Linkard)。

Sidecar（边车）是一个独立的进程，它拦截应用程序发送的每个数据包，重建应用程序级数据(即 RPC)，并应用策略或启用可观察性。

然而，由于冗余的 RPC (un)编组，使用 Sidecar 引入了大量的性能开销。例如，在 gRPC+Envoy 中，这种 RPC (un)编组，包括 HTTP 帧和 protobuf 编码，占端到端延迟开销的 62-73%。在该文章的评估中，使用边车将第 99 百分位 RPC 延迟增加 180%，并将带宽减少 44%。使用 Sidecar 将编组步骤的数量增加了三倍(从 4 到 12)。此外，Sidecar 方法在很大程度上与对网络硬件进行高效应用级访问的新兴趋势不兼容。

使用边车意味着必须在应用程序和边车之间复制数据缓冲区，从而降低了对网络进行零拷贝内核旁路访问的好处。

该文章构建的系统 mRPC 实现了 RPC 即托管服务的抽象，同时保持了与传统 RPC 库(例如 gRPC、Thrift)相似的端到端语义。mRPC 的目标是快速，支持灵活的策略实施，并为应用程序提供高可用性。

图 3-1 显示了 mRPC 体系结构和工作流的高级概述，将其分解为三个主要阶段:初始化、运行时和

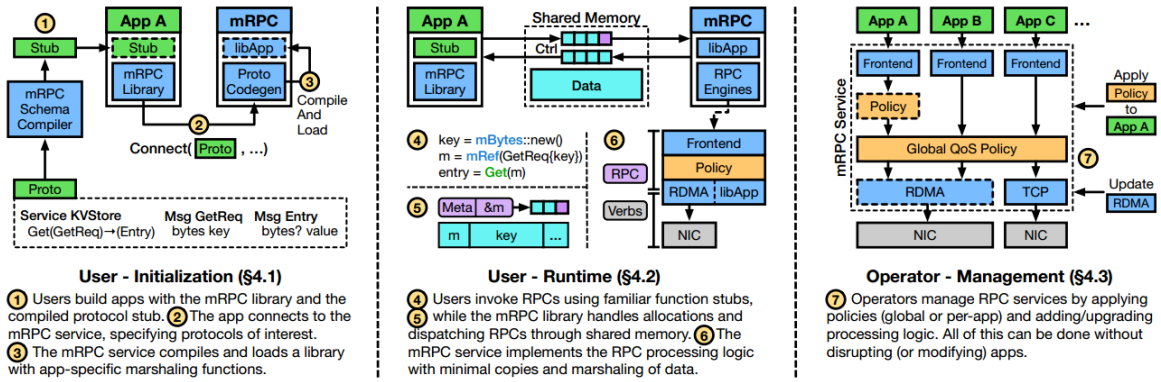

Figure 2: Overview of the mRPC workflow from the perspective of the users (and their applications) as well as infrastructure operators.

图 3-1 mRPC 工作流

管理。mRPC 服务作为非根用户空间进程运行，可以访问每个应用程序所需的网络设备和共享内存区域。在每个阶段中，重点关注同时运行 RPC 客户机应用程序和 mRPC 服务的单个机器的视图。RPC 服务器也可以与 mRPC 服务一起运行。在这种情况下，可以使用特定于 mrpc 的封送。然而，mRPC 也支持灵活的编组，使 mRPC 应用程序能够使用众所周知的格式(例如 gRPC)与外部对等体进行交互。在相关评估中，关注的是客户机和服务器都使用 mRPC 的情况。

4 如何拓宽应用场景

【Service Router】提到，分片和复制是构建可伸缩服务的两大关键技术。在 Meta 的场景中，大多数 RPC 请求都指向分片服务。传统的代理方式在处理对分片服务的请求时可能发生错误。SR 把对分片的支持视为首要任务，并使用一个统一框架来同时支持分片和复制。分片通常和应用程序逻辑紧密联系，所以本文的方案在服务网格和服务之间建立一个简单而泛用的抽象，强迫对关注点进行区分，这样 SR 就可以在不知道应用程序逻辑的情况

下路由消息。

图 4-1 是 Global Registry Service（GRS）保存在 RIB 中的服务注册记录，包含分片服务和非分片服务。

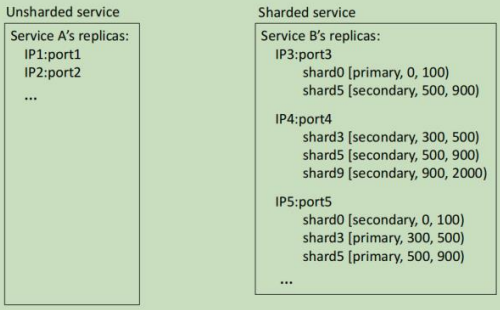

Figure 7: Examples of GRS' service registry records.

一个服务指定一个 128 位的键空间如何被划分到不同的分片，每个分片能够独立地在容器++间进行复制和迁移。每个分片副本都绑定一个角色，例如“primary role”、“secondary role”。

图 4-2 是进行分片服务调用的一个示例。

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

sr 发现 key618、secondary 存在于 ip3port3 和 ip4port4 的分片 5 中，并根据负载均衡策略选择其中一个。

作为对比，现存的泛用服务网格不支持分片服务，因而应用程序需要自己实现相关逻辑。

Shard-Map 的角色上的另一个方案是一致哈希，给定一组服务器，用哈希确定处理每个请求 key 的服务器。这样做不需要维护 Shard-Map 表，但无法支持高级分片服务，因为一旦在负载策略下出现分片的迁移时，哈希就会失效。SR 同时支持一致哈希和 Shard-Map，并且在 Meta 中，采用 Shard-Map 的服务数量是采用一致哈希的 5.4 倍。还有一个方案是客制化 Lookaside-service。这种方案灵活性最高，能够把分片发现和选择逻辑完全从服务网格中剥离开来。最初 Meta 里有些服务拥有者因为灵活性选择了这个方案，但后来发现 维护 Lookaside-service 的负担太重，且 Shard-Map 和一致哈希结合起来足以满足他们的绝大部分分片服务需要。

5 如何降低硬件开销

从系统结构设计和通信方式角度出发，Service Router 已经在前文提到，若使用传统 Sidecar、Lookaside Proxy 或 Remote Proxy 方案配置应用层路由器，会增加序列化、反序列化等环节，分布式管理系统也会带来额外机器开销。所以 Service Router 采用直接链接到执行环境的路由库 SRlib 直接建立客户端和服务器通信。

除了直接优化服务网格中的部件或过程外，还有一种间接降低硬件开销的方式：池化。

【Twine】[5]介绍了 Facebook 的集群管理系统 Twine，它已经在生产环境中运行了十年。

Twine 帮助将基础设施从一组专用于个人工作负载的定制机器池转变为具有可替换硬件的大规模共享基础设施。

计算作为一种实用工具的出现导致组织将其工作负载整合到共享的基础设施上，这是一个运行任何工作负载的公共资源池。集群管理系统帮助组织通过自动化、标准化流程管理集群。

集群管理系统在过去十年中取得了很大的进步，从 Mesos、Borg 到 Kubernetes。然而，现有系统在支持大规模共享基础设施方面仍然存在局限性:

1，它们通常关注于孤立的集群，对跨集群管理的支持有限。

2. 他们很少咨询应用程序的生命周期管理操作，这使得应用程序维护其可用性变得更加困难。例如，它们可能在应用程序构建另一个数据副本之前不知不觉地重新启动应用程序，从而导致数据不可用。

3. 它们很少允许应用程序向共享机器提供其首选的自定义硬件和操作系统设置。

缺乏定制可能会对共享基础设施上的应用程序性能产生负面影响。

4. 他们通常更喜欢具有更多 cpu 和内存的大型机器，以便堆栈工作负载并提高利用率。如果管理不善，未充分利用的大型机器会浪费电力，而电力通常是数据中心的有限资源。

这些限制可能导致共享基础设施的承诺无法实现:(1)人为地将共享范围限制在一个集群;(2)和

(3)强调了共享基础设施对标准化的偏好与应用程序对定制的需求之间的紧张关系;(4)要求将重点从单机利用率转向全局优化。

一个区域由多个数据中心组成，一个数据中心通常被划分为由数万台机器组成的集群，这些集群通过高带宽网络连接。与 Borg 和 Kubernetes 一样，每个集群的隔离控制平面会导致搁浅的容量和操作负担，因为工作负载无法轻松地跨集群移动。例如，集群中的耗电作业可能触发功率上限，影响服务吞吐量，直到人工将有问题的作业转移到其他集群。

为了解决上述问题，该文章扩展了单个 Twine 控制平面来管理一个区域内所有数据中心的 100 万台机器。

无处不在的共享基础设施的目标导致 Facebook 做出了一些与常规做法相反的决策。例如，Twine 不是为每个集群部署一个隔离的控制平面，而是扩展单个控制平面来管理一个地理区域内所有数据中心的 100 万台机器，并透明地跨集群移动作业。

Twine 在共享的基础设施中容纳特定于工作负载的定制，并且这种方法进一步偏离了常见的实践。TaskControl API 允许应用程序与 Twine 协作来处理容器生命周期事件，例如，在滚动升级期间首先重启 ZooKeeper 部署的追随者，最后重启它的领导者。主机配置文件捕获硬件和操作系统设置，工作负载可以调优以提高性能和可靠性;Twine 动态地为工作负载分配机器，并相应地切换主机配置文件。

最后，与在大型机器上优先堆叠工作负载以提高利用率的传统观念相反，我们普遍部署具有单个 CPU 和 64GB RAM 的节能小型机器，以实现更高的每瓦特性能，并且我们利用自动缩放来提高机器利用率。

通过把公共设施抽象为统一资源池，为资源池管理机制增加针对服务的定制化功能，并广泛部署使用节能小型机器，Twine 让公共设施的利用率大大提高，并显著降低了一些场景的设施用电。对于一些大型集群来说，电力开销甚至比机器开销更重要。

6 总结

服务网格已经成为互联网服务的基础设施，如何在现有基础上进一步优化服务网格：扩大规模、提升效率、拓宽应用、降低开销，以应对更加复杂的条件和更加密集的流量，是设施提供商、系统设计者迫切考虑的问题。

Service Router 通过分散路由信息库 RIB、让 L7 路由器自行配置，使控制平面伸缩性得到加强，并通过 SRlib 减少了通信开销。Flux 建立了一个全局容量-需求编排系统让地理位置分散的服务和应用能够被合理安排。mRPC 通过系统程序成批处理 RPC 消息，消除了边车程序的额外代价，并摆脱了 RPC 库的低可操作性。SR 通过抽象出分片服务框架，通过 RIB 中的分片服务注册表引导路由器路由对分片的请求，提供了对分片服务的内建支持。Twine 通过把共享设施抽象为资源池，降低了设施的空闲率，从而间接降低了硬件开销。

参 考 文 献

[1] Charles Killian, Waqar Mohsin, Henrik Muehe, JoonOng, Leon Poutievski, Arjun Singh, Lorenzo Vicisano,et al. Orion: Google’s Software-Defined NetworkingControl Plane. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21),pages 83–98, 2021.

[2] Saokar H , Demetriou S , Magerko N ,et al.ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta[J]. 2023.

[3] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew,Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, JustinMeza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. Global CapacityManagement with Flux. In Proceedings of the 17thUSENIX Symposium on Operating Systems Design andImplementation, 2023.

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

[5] Chunqiang Tang, Kenny Yu, Kaushik

Veeraraghavan,Jonathan Kaldor, Scott Michelson, Thawan Kooburat,Aravind Anbudurai, Matthew Clark, Kabir Gogia, LongCheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, TuomasPelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A UnifiedCluster Management System for Shared Infrastructure.In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, pages 787–803. USENIX Association, 2020.

附录 汇报记录

不好意思，老师和同学在我汇报后都没有向我提出任何问题！老师只说让我以后把 PPT 做得稍微好看一点。

如果让我自己提几个问题给自己然后自己回答，那也不是不行。但我通过这些论文对服务网格的了解只局限于一个宏观的、系统架构和设计上的概览。恐怕目前无法提出什么像样的问题。

硬要我提一些问题的话，那我会问：1，服务网格出现之前服务和集群是怎么相处并行使功能的？2，服务网格在互联网商业环境中是什么样的角色？每家公司都有不同的服务网格吗？谁用谁的服务网格要向谁给多少钱？3，RPC 通信在微服务中是怎么起作用的？RPC 通信库有哪些？不同的公司或服务会魔改吗？RPC 传输过程？不同的 RPC 拦截处理加工再转发机制是怎么样的？4，怎么测量区域间延迟，更进一步，怎么测量跨区域不同服务间的延迟，并让该测量结果有效？5，公共设施抽象成资源池是怎么做到的？怎么处理不同硬件间的差异？处理差异所带来的额外开销是否值得？6，控制平面的消息如何进行同步和共识？怎么处理由局部网络环境等带来的信息差异和状态差异？
等等。
回答这些问题需要仔细研究很久，我暂时没有能力给出解答！