北京工業大學
BEIJING UNIVERSITY OF TECHNOLOGY

文 献 翻 译

（简装版）

姓　名　　田昕峣
学　号　　15051204

日　期　　2019 年 6 月

英文文献翻译

目录

英文文献翻译

What Is a Service Mesh?

A service mesh is a configurable, low-latency infrastructure layer designed to handle a high volume of network-based interprocess communication among application infrastructure services using application programming interfaces (APIs). A service mesh ensures that communication among containerized and often ephemeral application infrastructure services is fast, reliable, and secure. The mesh provides critical capabilities including service discovery, load balancing, encryption, observability, traceability, authentication and authorization, and support for the circuit breaker pattern.

The service mesh is usually implemented by providing a proxy instance, called a sidecar, for each service instance. Sidecars handle interservice communications, monitoring, and security-related concerns – indeed, anything that can be abstracted away from the individual services. This way, developers can handle development, support, and maintenance for the application code in the services; operations teams can maintain the service mesh and run the app.

Istio, backed by Google, IBM, and Lyft, is currently the best-known service mesh architecture. Kubernetes, which was originally designed by Google, is currently the only container orchestration framework supported by Istio. Vendors are seeking to build commercial, supported versions of Istio. It will be interesting to see the value they can add to the open source project.

Istio is not the only option, and other service mesh implementations are also in development. The sidecar proxy pattern is most popular, as illustrated by projects from Buoyant, HashiCorp, Solo.io, and others. Alternative architectures exist as well: Netflix's technology suite is one such approach where service mesh functionality is provided by application libraries (Ribbon, Hysterix, Eureka, Archaius), and platforms such as Azure Service Fabric embed service mesh-like functionality into the application framework.

Service mesh comes with its own terminology for component services and functions:

Container orchestration framework. As more and more containers are added to an application's infrastructure, a separate tool for monitoring and managing the set of containers – a container orchestration framework – becomes essential. Kubernetes seems to have cornered this market, with even its main competitors, Docker Storm and Mesosphere DC/OS, offering integration with Kubernetes as an alternative.

Services and instances (Kubernetes pods). An instance is a single running copy of a microservice. Sometimes the instance is a single container; in Kubernetes, an instance is made up of a small group of interdependent containers (called a pod). Clients rarely access an instance or pod directly; rather they access a service, which is a set of

identical instances or pods (replicas) that is scalable and fault-tolerant.

Sidecar proxy. A sidecar proxy runs alongside a single instance or pod. The purpose of the sidecar proxy is to route, or proxy, traffic to and from the container it runs alongside. The sidecar communicates with other sidecar proxies and is managed by the orchestration framework. Many service mesh implementations use a sidecar proxy to intercept and manage all ingress and egress traffic to the instance or pod.

Service discovery. When an instance needs to interact with a different service, it needs to find – discover – a healthy, available instance of the other service. Typically, the instance performs a DNS lookup for this purpose. The container orchestration framework keeps a list of instances that are ready to receive requests and provides the interface for DNS queries.

Load balancing. Most orchestration frameworks already provide Layer 4 (network) load balancing. A service mesh implements more sophisticated Layer 7 (application) load balancing, with richer algorithms and more powerful traffic management. Load-balancing parameters can be modified via API, making it possible to orchestrate blue-green or canary deployments.

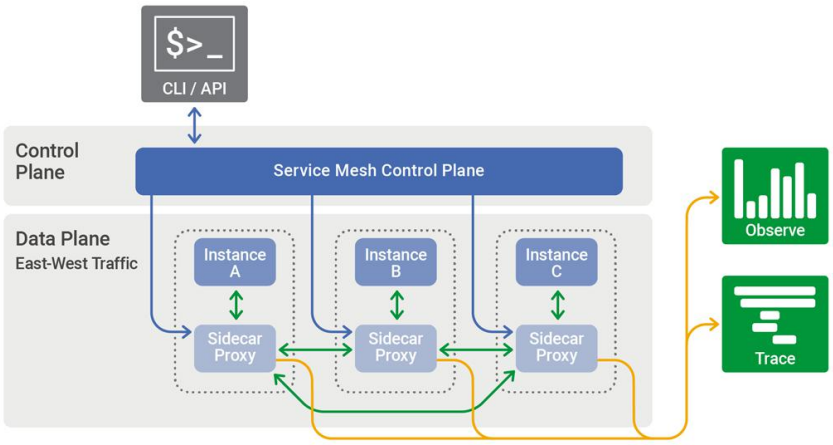
Encryption. The service mesh can encrypt and decrypt requests and responses, removing that burden from each of the services. The service mesh can also improve performance by prioritizing the reuse of existing, persistent connections, which reduces the need for the computationally expensive creation of new ones. The most common implementation for encrypting traffic is mutual TLS (mTLS), where a public key infrastructure (PKI) generates and distributes certificates and keys for use by the sidecar proxies.

Authentication and authorization. The service mesh can authorize and authenticate requests made from both outside and within the app, sending only validated requests to instances.

Support for the circuit breaker pattern. The service mesh can support the circuit breaker pattern, which isolates unhealthy instances, then gradually brings them back into the healthy instance pool if warranted.

The part of a service mesh application that manages the network traffic between instances is called the data plane. Generating and deploying the configuration that controls the data plane’s behavior is done using a separate control plane. The control plane typically includes, or is designed to connect to, an API, a command-line interface, and a graphical user interface for managing the app.

英文文献翻译



The control plane in a service mesh distributes configuration across the sidecar proxies in the data plane

A common use case for service mesh architecture is solving very demanding operational problems when using containers and microservices. Pioneers in microservices include companies like Lyft, Netflix, and Twitter, which each provide robust services to millions of users worldwide, hour in and hour out. (See our in-depth description of some of the architectural challenges facing Netflix.) For less demanding application needs, simpler architectures are likely to suffice.

Service mesh architectures are not ever likely to be the answer to all application operations and delivery problems. Architects and developers have a great many tools, only one of which is a hammer, and must address a great many types of problems, only one of which is a nail. The NGINX Microservices Reference Architecture, for instance, includes several different models that give a continuum of approaches to using microservices to solve problems.

The elements that come together in a service mesh architecture – such as NGINX, containers, Kubernetes, and microservices as an architectural approach – can be, and are, used productively in non-service mesh implementations. For example, Istio was developed as a complete service mesh architecture, but its modular design means developers can pick and choose the component technologies they need. With this in mind, it's worth developing a solid understanding of service mesh concepts, even if you're not sure if and when you'll fully implement a service mesh application.

英文文献翻译
什么是服务网格(Service Mesh)?

服务网格 即一个基础设施层，用于处理服务间通信。
这是 Service Mesh 概念的提出者 William Morgan 在他的文章中给出的定义
译者注。

服务网格(Service Mesh)是指用于微服务应用的可配置基础架构层(configurable infrastructure layer)。它使每个 service 实例之间的通信更加流畅、可靠和迅速。服务网格提供了诸如服务发现、负载均衡、加密、身份鉴定、授权、支持熔断器模式(Circuit Breaker Pattern)以及其他一系列功能。

服务网格的实现通常是提供一个代理实例，我们称之为"sidecar"。sidecar 包含在每一个 service 之中。sidecar 主要处理 service 间的通信、监控、以及一些安全相关的考量 —— 任何可以从服务本体中抽象出来的安全方面的部分。通过这种方式，开发者可以在服务中专注于开发、支持以及维护；运维人员可以维护服务网格并运行 app。

Istio(由 Google、IBM、Lyft 公司在背后进行支持) 是目前最广为人知的一款服务网格架构。Kubernetes(由 Google 最早进行设计并开源) 是目前 Istio 唯一支持的容器组织框架。

服务网格对于其组件 service 和功能有如下术语：

1. 容器组织框架(Container orchestratiob framework):

随着越来越多的容器被加入到应用的组织架构中，一个用于监控和管理这一系列容器的独立工具，即：容器组织框架，就变得不可或缺了。Kubernetes 由于考虑到市场原因(Docker Swarm 和 Mesosphere DC/OS 是其主要竞争者)，Istio 并非强制安装。因此安装时是否集成 Istio 变成了可选项。

2. Service 与 Service 实例(Service Instance):

确切来讲，开发者创建的并非是一个 service，而是一个由 service 定义的或者框架定义的实例。那些 app 创建的 service 实例由此而来，并且真正干活的是这些实例。然而，"service"一词经常被用来同时指代“定义 service 的东西”和“service 实例”这两者。

3. Sidecar 代理(Sidecar Proxy):

sidecar proxy 指专“注于每个具体的 service 实例”的一个代理实例。它与其他的 sidecar proxy 通讯并由容器组织框架(比如 Kubernetes)进行统一管理。

4. 服务发现(Service discovery):

当一个实例需要与其他 service 进行通讯时，它需要“寻找并发现(find & discovery)”另一个健康、可用的 service 实例。容器管理框架(container managment framework)维护着一个时刻准备接收请求的实例列表。

5. 负载均衡(Load balancing):

在一个服务网格中，负载均衡自底向上地工作着。由服务网格维护的可用实例列表可以进行打分、评级、并选出“最闲”的 service，这就是在高层次上的负载均衡。

6. 加密(Encryption):

服务网格可以加密和解密服务间的请求与相应，并从 service 中将这些步骤分离，从而减轻每个 service 内的负担。服务网格可以通过优先再利用已经存在的、持续的连接(connection)来提高性能，以便减少“新建连接”时高昂的计算开销。

7. 认证与权限(Authentication and Authorization):

服务网格可以授权并认证从外来的或是 app 内服的各种请求，并且只发送那些有效的请求给 service。

8. 支持熔断器模式(Support for the circuit breaker pattern):

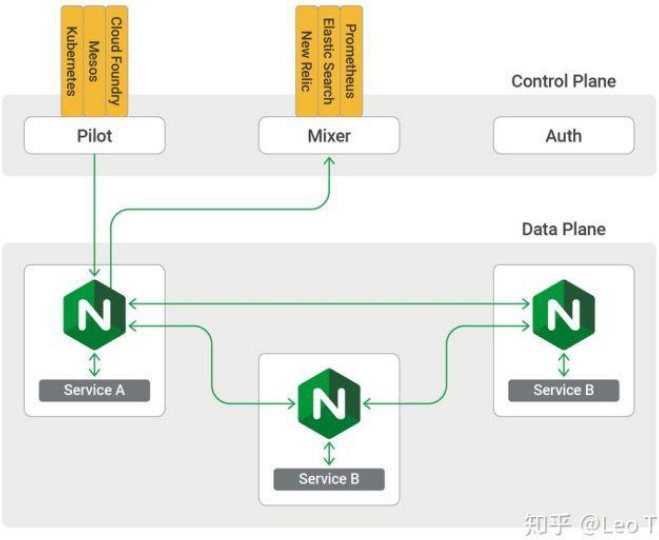
服务网格能够支持熔断器模式，它能隔离那些不健康的实例，并逐渐将那些有保证的实例再次添加进健康实例池(healthy instance pool)。

data plane 与 control plane
data plane 即数据流动的、工作真正被处理的部分
control plane 即管理监控这些工作运行的部分
译者注。

英文文献翻译

服务网格中工作被完成的部分(包括 service 实例、sidecar 代理以及它们间的沟通)被称作 data plane 。但是服务网格中也会包含一个监控管理层，我们称它为 control plane。

control plane 处理的工作诸如：创建新实例、终止不健康或者不需要的实例、服务监控、集成监控与管理、实施应用范围的策略(application-wide policies)以及优美地将整个应用作为一个整体结束掉。典型的 control plane 包括、或被设计成用于连接 API、命令行工具、或者一个用于管理整个应用的可视化用户界面。



上图为 nginMesh —— 一个基于 Istio 的服务网格，包括了 control plane，data plane 以及 使用 Nginx 作为 sidecar Proxy

Nginx 有一个可以兼容 Istio 架构的服务网格，我们称它为 nginMesh 。上图基于 nginMesh 的架构展示了 Nginx 作为 sidecar Proxy 的角色，与其他典型的 Istio 组件共同发挥作用。

一个典型的网格服务架构场景是，当你使用容器和微服务去解决一个对性能要求很严苛的应用开发工作时。比如使用微服务架构的先锋们——Lyft，Netflix 和 Twitter 这些公司，他们需要健壮的 service 去服务全世界上百万的用户，每时每刻(可以参考 Netflix 利用这种架构的实际应用场景。对于那些较少访问需求的应用，上图这种简单的架构就足够了。

服务网格架构不可能永远是所有应用开发和交付问题的解决方案。架构师和开发者有许多很棒的工具，但要分清哪些是锤子、哪些是钉子。Nginx 微服务相关架构，包含了多种不同的模型，提供了解决微服务问题的整体方案。

英文文献翻译

这只是服务网格架构中的一个元素——就像 Nginx、容器、Kubernetes 和微服务作为一整个架构级的方案一样——可以被，也已经被用于富有成果的没有集成服务网格的实施中。比如 Istio 这个服务网格架构，它就设计成模块化的，因此开发者可以选择是否真的需要用它。将上面这些概念记在你的脑海中，当你开发一个坚实的应用时，服务网格的这些概念是很有价值的。即使你并不非常确切地明白它们。

英文文献翻译

What Is Layer 7 Load Balancing?

Load balancing can be performed at various layers in the Open Systems Interconnection (OSI) Reference Model for networking. Here we offer an overview of two load-balancing options at two different layers in the model.

Differences Between Layer 4 and Layer 7 Load Balancing

Layer 4 load balancing operates at the intermediate transport layer, which deals with delivery of messages with no regard to the content of the messages. Transmission Control Protocol (TCP) is the Layer 4 protocol for Hypertext Transfer Protocol (HTTP) traffic on the Internet. Layer 4 load balancers simply forward network packets to and from the upstream server without inspecting the content of the packets. They can make limited routing decisions by inspecting the first few packets in the TCP stream.

Layer 7 load balancing operates at the high-level application layer, which deals with the actual content of each message. HTTP is the predominant Layer 7 protocol for website traffic on the Internet. Layer 7 load balancers route network traffic in a much more sophisticated way than Layer 4 load balancers, particularly applicable to TCP-based traffic such as HTTP. A Layer 7 load balancer terminates the network traffic and reads the message within. It can make a load-balancing decision based on the content of the message (the URL or cookie, for example). It then makes a new TCP connection to the selected upstream server (or reuses an existing one, by means of HTTP keepalives) and writes the request to the server.

Benefits of Layer 7 Load Balancing

Layer 7 load balancing is more CPU-intensive than packet-based Layer 4 load balancing, but rarely causes degraded performance on a modern server. Layer 7 load balancing enables the load balancer to make smarter load-balancing decisions, and to apply optimizations and changes to the content (such as compression and encryption). It uses buffering to offload slow connections from the upstream servers, which improves performance.

A device that performs Layer 7 load balancing is often referred to as a reverse-proxy server.

英文文献翻译

An Example of Layer 7 Load Balancing

Let’s look at a simple example. A user visits a high-traffic website. Over the course of the user’s session, he or she might request static content such as images or video, dynamic content such as a news feed, and even transactional information such as order status. Layer 7 load balancing allows the load balancer to route a request based on information in the request itself, such as what kind of content is being requested. So now a request for an image or video can be routed to the servers that store it and are highly optimized to serve up multimedia content. Requests for transactional information such as a discounted price can be routed to the application server responsible for managing pricing. With Layer 7 load balancing, network and application architects can create a highly tuned and optimized server infrastructure or application delivery network that is both reliable and efficiently scales to meet demand.

How Can NGINX Plus Help?

NGINX Plus and NGINX are the best-in-class load-balancing solutions used by high-traffic websites such as Dropbox, Netflix, and Zynga. More than 358 million websites worldwide, including the majority of the 100,000 busiest websites, rely on NGINX Plus and NGINX to deliver their content quickly, reliably, and securely.

As a software load balancer, NGINX Plus is much less expensive than hardware-based solutions with similar capabilities. The comprehensive Layer 7 load balancing capabilities in NGINX Plus enable you to build a highly optimized application delivery network.

When you place NGINX Plus in front of your web and application servers as a Layer 7 load balancer, you increase the efficiency, reliability, and performance of your web applications. NGINX Plus helps you maximize both customer satisfaction and the return on your IT investments.

英文文献翻译
什么是七层负载均衡?

负载均衡可以在 OSI(Open Systems Interconnection)网络模型中的很多层次上被实现。在这里，我们将提供这个网络模型中，两种不同层次的负载均衡选择的概览。

四层负载均衡(Layer 4 Proxy)和七层负载均衡(Layer 7 Proxy)的区别

四层负载均衡(L4 load balancing):

主要工作于处于 OSI 模型中间位置的传输层(transport layer)，它主要处理消息的传递，而不管消息的内容。在互联网上，TCP 就是 HTTP 传输方式的四层协议(Layer 4 Protocol)。四层负载均衡只针对由上游服务发送和接收的网络包，而并不检查包内的具体内容是什么。四层负载均衡可以通过检查 TCP 流中的前几个包，从而决定是否限制路由。

七层负载均衡(L7 load balancing):

主要工作于处于 OSI 模型顶层位置的应用层(application layer)，它主要处理每条消息中的真正内容。在互联网上，HTTP 是网络通讯中占据主导地位的七层协议(Layer 7 Protocol)。七层负载均衡在路由网络传输时比四层负载均衡更加复杂和巧妙，特别适合像 HTTP 这种基于 TCP 传输的方式。 一个七层负载均衡器终止网络传输并读取消息中的内容。它可以基于消息中内容(比如 URL 或者 cookie 中的信息)来做出负载均衡的决定。之后，七层负载均衡器建立一个新的 TCP 连接来选择上游服务(或者再利用一个已经存在的 TCP 连接，通过 HTTP keepalives 的方式)并向这个服务发出请求。

七层负载均衡的优势

七层负载均衡的 CPU 密集程度比基于包的四层负载均衡更高，但是在现代服务中却极少降低其性能。七层负载均衡能够让均衡器做更小的负载均衡决定，并且会根据消息的内容(比如压缩和加密)利用最优化方式做出改变。它运用缓存的方式来卸载上游服务较慢的连接，并显著地提高了性能。

使用七层负载均衡的设备经常被用于反向代理。

英文文献翻译

一个七层负载均衡的例子

让我们来看一个简单的例子吧：用户访问一个繁忙的网站。在这个用户 session 的航向上，他或她可能会请求静态内容——比如图片或者视频，动态内容——比如新闻递送，甚至是事务型信息——比如外卖点单的状态。七层负载均衡允许均衡器依据请求自身的信息进行路由，比如被请求内容的类型。所以现在一个针对图片和视频的请求可以被路由到存储并高度优化的多媒体内容服务器上。对于事物型信息比如商品折后价，可以被路由到响应管理价格的应用服务器上。用了七层负载均衡，网络和应用的架构师可以建立一个高速调整且高度优化的、针对需求可靠且可有效延展的服务基础架构或应用递送网络。

让 Nginx Plus 助你一臂之力！

Nginx Plus 和 Nginx 可能是负载均衡领域中最好的解决方案，它被许多高访问量的网站所使用，比如 Dropbox、Netflix 和 Zynga。全世界超过 358 万个网站，包括 10 万个最繁忙的主流网站，都可依靠着 Nginx Plus 和 Nginx 来快速、可靠、安全地传送着他们的内容。

作为一个基于软件的负载均衡器，Nginx Plus 和那些基于硬件的解决方案功能相仿，但是价格却便宜的多。Nginx Plus 中全面的七层负载均衡功能完全能够帮助你建设一个高度优化的应用传送网络。

当你将 Nginx Plus 作为一个七层负载均衡器置于你的网站或应用之前时，你就已经提高了你网站和应用的效率、可靠性和性能。 Nginx Plus 帮助你同时最大化用户满意度和你 IT 投资的回报。

英文文献翻译

Envoy vs NGINX vs HAProxy: Why the open source Ambassador API Gateway chose Envoy

NGINX, HAProxy, and Envoy are all battle-tested L4 and L7 proxies. So why did we end up choosing Envoy as the core proxy as we developed the open source Ambassador API Gateway for applications deployed into Kubernetes?

It's an L7 world

In today's cloud-centric world, business logic is commonly distributed into ephemeral microservices. These services need to communicate with each other over the network. The core network protocols that are used by these services are so-called "Layer 7" protocols, e.g., HTTP, HTTP/2, gRPC, Kafka, MongoDB, and so forth. These protocols build on top of your typical transport layer protocols such as TCP. Managing and observing L7 is crucial to any cloud application, since a large part of application semantics and resiliency are dependent on L7 traffic.

The Proxy Battle

Ambassador was designed from the get go for this L7, services-oriented world, with us deciding early on to build only for Kubernetes. We knew we wanted to avoid writing our own proxy, so we considered HAProxy, NGINX, and Envoy as possibilities. At some level, all three of these proxies are highly reliable, proven proxies, with Envoy being the newest kid on the block.

Evaluating Proxies

We started by evaluating the different feature sets of the three proxies. We soon realized that L7 proxies in many ways are commodity infrastructure. All proxies do an outstanding job of routing traffic L7 reliably and efficiently, with a minimum of fuss. And while they weren't at feature parity, we felt that we could, if we had to, implement any critical missing features in the proxy itself. After all, they're all open source!

We took a step back and reconsidered our evaluation criteria. Given the rough functional parity in each of these solutions, we refocused our efforts on evaluating each project through a more qualitative lens. Specifically, we looked at each project's community, velocity, and philosophy. We focused on community because we wanted a vibrant community where we could contribute easily. Related to community, we wanted to see that a project had good forward velocity, as it would show the project would quickly evolve as customer needs evolved. And finally, we wanted a project

that would align as closely as possible with our view of a L7-centric, microservices world.

HAProxy

Several years ago, some of us had worked on Baker Street, an HAProxy-based client-side load balancer inspired by AirBnb's SmartStack. HAProxy is a very reliable, fast, and proven proxy. While we were happy with HAProxy, we had some longer-terms concerns around HAProxy. HAProxy was initially released in 2006, when the Internet operated very differently than today. The velocity of the HAProxy community didn't seem to be very high. For example, v1.5 added SSL after four years. We ourselves had experienced the challenges of hitless reloads (being able to reload your configuration without restarting your proxy) which were not fully addressed until the end of 2017 despite epic hacks from folks like Joey at Yelp. With v1.8, the HAProxy team has started to catch up to the minimum set of features needed for microservices, but 1.8 didn't ship until November 2017.

NGINX

NGINX is a high-performance web server that does support hitless reloads. NGINX was designed initially as a web server, and over time has evolved to support more traditional proxy use cases. NGINX has two variants, NGINX Plus, a commercial offering, and NGINX open source. Per NGINX, NGINX Plus "extend[s] NGINX into the role of a frontend load balancer and application delivery controller." Sounds perfect! Unfortunately, though, since we wanted to make Ambassador open source, NGINX Plus was not an option for us.

NGINX open source has a number of limitations, including limited observability and health checks. To circumvent the limitations of NGINX open source, our friends at Yelp actually deployed HAProxy and NGINX together.

More generally, while NGINX had more forward velocity than HAProxy, we were concerned that many of the desirable features would be locked away in NGINX Plus. The NGINX business model creates an inherent tension between the open source and Plus product, and we weren't sure how this dynamic would play out if we contributed upstream. (Note that HAProxy has a similar tension with Enterprise Edition, but there seems to be less divergence in the feature set between EE and CE in HAProxy).

Envoy Proxy

Envoy is the newest proxy on the list, but has been deployed in production at Lyft,

Apple, Salesforce, Google, and others. In many ways, the release of Envoy Proxy in September 2016 triggered a round of furious innovation and competition in the proxy space.

Envoy was designed from the ground up for microservices, with features such as hitless reloads (called hot restart), observability, resilience, and advanced load balancing. Envoy also embraced distributed architectures, adopting eventual consistency as a core design principle and exposing dynamic APIs for configuration. Traditionally, proxies have been configured using static configuration files. Envoy, while supporting a static configuration model, also allows configuration via gRPC/protobuf APIs. This simplifies management at scale, and also allows Envoy to work better in environments with ephemeral services.

We loved the feature set of Envoy and the forward-thinking vision of the product. We also discovered the community around Envoy is unique, relative to HAProxy and NGINX. Unlike the other two proxies, Envoy is not owned by any single commercial entity. Envoy was originally created by Lyft, and as such, there is no need for Lyft to make money directly on Envoy. Matt Klein, creator of Envoy, explicitly decided that he would not start an Envoy platform company. There is no commercial pressure for a proprietary Envoy Plus or Envoy Enterprise Edition. As such, the community focuses only on the right features with the best code, without any commercial considerations. Finally, Lyft has donated the Envoy project to the Cloud Native Computing Foundation. The CNCF provides an independent home to Envoy, insuring that the focus on building the best possible L7 proxy will remain unchanged.

A year later …
We couldn’t be happier with our decision to build Ambassador on Envoy. The rich feature set has allowed us to quickly add support for gRPC, rate limiting, shadowing, canary routing, and observability, to name a few. And in the cases where Envoy’s feature set hasn’t met our requirements (e.g., authentication), we’ve been able to work with the Envoy community to implement the necessary features.

The Envoy code base is moving forward at an unbelievable pace, and we’re excited to continue taking advantage of Envoy in Ambassador. Stay tuned as we continue iterating on Ambassador!

英文文献翻译

Envoy 、 Nginx 和 HAProxy，微服务中的通信代理该如何抉择?

Nginx, HAProxy 和 Envoy 都是久经沙场的 L4 和 L7 代理。我们最终选择了 Envoy 作为我们各种应用以及“开源工具 Ambassador API ”的一部分，并部署进 Kubernetes 的原因到底是什么？

当今的主宰是 L7

在今天这个以云为中心的世界上，业务逻辑基本都是分布在许多转瞬即逝的微服务之中。这些服务需要通过网络与其他服务进行交流。而被这些服务所运用的核心网络协议被称为“ 七层网络协议(Layer 7 Protocols)” ，它们包括: HTTP, HTTP/2, gRPC, Kafka, MongoDB 等等。这些协议建立在经典的传输层协议(比如 TCP)之上。由于很大一部分应用的语义(semantics)和恢复工作(resiliency)都依靠 L7 传输，因此管理和观测 L7 对于任何云应用都是至关重要的。

Proxy 大战

Ambassador 从设计之初就是冲着 L7 和面向服务去的，而且我们最初只决定在 Kubernetes 上创建。我们清楚地明白必须避免自己从头写一个 proxy ，所以我们考虑了 HAProxy , Nginx 和 Envoy 作为可选项。在很多方面，这三个 proxy 都是高可靠、经过考验的，其中 Envoy 还是他们之中最年轻的那个。

评估这三个 Proxy

我们从评估这三个 proxy 的不同特点开始。很快我们就意识到 L7 代理在很多方面都是一个“商业架构”。所有 proxy 都很出色，而且在路由 L7 传输时都非常可靠、高效，用在这里甚至还有点小题大做。以及尽管他们在特性上略有出入，我们依然觉得我们应该、或者必须亲手实现一些这些 proxy 本身没有的重要特性。毕竟，它们都是开源工具！

我们退了一步并重新考虑起我们的评估标准。在它们的功能特性上已然有了一个初步认识，我们将评估的重点重新聚焦在了这三个方面：准确来讲，我们主要参考每个 proxy 的社区、迭代速度和设计哲学。我们关注社区是因为，一个生机勃勃的社区可以使我们贡献代码更加轻松方便。关于社区，我们还想看到这款 proxy 有一个良好的迭代频率，这显示出这款 proxy 能够快速进化发展出用户需要的功能。最后，我们想要这款 proxy 能够尽可能紧密地贴合我们微服务世界中以 L7 为中心的愿景。

英文文献翻译

HAProxy

很多年前，我们当中有人在 Baker Street 工作，一个基于 HAProxy 的客户端负载均衡器被 AirBnb's SmartStack 发明出来了。HAProxy 是一个非常可靠，迅速而且经历考验的 proxy 。尽管我们使用 HAProxy 很开心，但我们对于 HAProxy 由更为长久的考量。HAProxy 的首个版本于 2006 年发布，而当时的互联网工作方式与今天的差别很大。HAProxy 社区的迭代速度看起来并不快。举个例子， v1.5 版本在 4 年后才加入了 SSL 功能。我们已经经历过了热部署(无需重启服务就能让你的新配置载入)的挑战，而这个挑战直到 2017 年年底也没有被完全解决(尽管坊间流传它们的代码库被 Joey at Yelp 给骇了一把)。在 v1.8 这个版本， HAProxy 的团队已经开始赶制微服务架构所需要的最基本功能，但是直到 2017 年 11 月这个 v1.8 版本还是没被鼓捣出来。

Nginx

Nginx 是一个高性能的 web server 并且支持热部署。 Nginx 最初被设计成一个 web server，并且时过境迁逐渐进化成了一个支持更加传统的 proxy 用例。Nginx 有两个版本——收费的商业版本 Nginx Plus 和免费的开源版本 Nginx open source 。对于 Nginx ， Nginx Plus 是 “拥有前置负载均衡器和应用传递控制的扩展版 Nginx ”。听起来棒极了！但是很不幸，我们想要让 Ambassador 完全开源，而 Nginx Plus 并不是我们的菜。

Nginx open source 有一些限制，其中包括了“可观测性”和“健康诊断”。为了规避这些 Nginx open source 的限制，我们在 Yelp 的朋友实际上将 HAProxy 和 Nginx 部署在一起了。

更通俗的来讲，尽管 Nginx 比 HAProxy 迭代速度快很多，我们担心很多我们需要的特性都锁在了 Nginx Plus 中。Nginx 的商业模型使得 Nginx open source 和 Nginx Plus 之间的继承关系有不少矛盾，而且如果向上游贡献代码，我们并不确定我们的劲头会被如何榨干。(注意 HAProxy 同样是有“企业版”的，但是它的“企业版”和“社区版”的差别看起来并没有 Nginx 这么大)

Envoy Proxy

Envoy 在我们的选项中是最年轻的，但它已经被 Lyft，Apple，Salesforce，Google 和其他大型 IT 企业用于生产环境。在很多方面，Envoy 于 2016 年 9 月的发布触发了代理领域的一轮激烈创新和竞争。

英文文献翻译

Envoy 自底向上都是为了微服务而设计的，具有诸如热部署，可观测性，可恢复性和先进的负载均衡等特征。Envoy 同样拥抱分布式架构，应用最终一致性(eventual consistency)作为核心设计原则以及为配置项暴露动态 API (dynamic API)。传统意义上，proxy 应该被静态的配置文件所配置。然而 Envoy 不但支持静态配置文件的模型，而且允许通过 gRPC 和 protobuf API 进行配置。这会使得在扩容时的管理变得异常简单，而且也会允许 Envoy 更好的在转瞬即逝的微服务环境中工作。

我们爱死 Envoy 这一系列特性和产品版本超前的设计理念了。我们也发现 Envoy 的社区相较 HAProxy 和 Nginx 是独特的。不像其他两个 proxy，Enovy 并不被任何商业实体所拥有。Envoy 最早是由 Lyft 创建的，因此同样地，Lyft 也没必要用 Envoy 直接赚钱。Envoy 的创造者 Matt Klein 明确表示他不会为 Envoy 成立一个公司来赚钱。并没有任何经济压力需要 Envoy 的所有者整出来个 “Envoy Plus” 或者 “Envoy EE” 。因此，Envoy 的社区只专注于“如何用最好的代码开发出正确的功能”，而不需要总把心思放在怎么赚钱上。最后，Lyft 将整个 Envoy 工程捐赠给了 CNCF (Cloud Native Computing Foundation)。CNCF 给 Envoy 提供了一个独立的家，来确保 Envoy 专注于建设最棒的 L7 proxy 并始终保持其领先地位。

选择了 Envoy 的一年之后...

没有什么能比决定使用 Envoy 来创建 Ambassador 更令我们开心的了。其丰富的特性集让我们快速地添加 gRPC、速率限制(rate limiting)、隐匿功能(shadowing)、金丝雀测试(canary routing)和可观测性等一系列功能的支持。对于那些 Envoy 的特性没能满足我们需求的部分(比如权限问题)，我们已经能和 Envoy 社区一起工作并实现必要的功能特性了。

Envoy 的代码库以一种难以置信的节奏大步向前迈进着，我们对 Envoy 持续提供给 Ambassador 的好处非常激动。保持这个节奏，我们将会继续利用 Envoy 迭代 Ambassador！

英文文献翻译

Service Discovery in a Microservices Architecture

You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook – Microservices: From Design to Deployment. Also, please look at the new Microservices Solutions page.

This is the fourth article in our series about building applications with microservices. The first article introduces the Microservices Architecture pattern and discussed the benefits and drawbacks of using microservices. The second and third articles in the series describe different aspects of communication within a microservices architecture. In this article, we explore the closely related problem of service discovery.

Why Use Service Discovery?

Let's imagine that you are writing some code that invokes a service that has a REST API or Thrift API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network locations of service instances are relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated.

In a modern, cloud-based microservices application, however, this is a much more difficult problem to solve as shown in the following diagram.

Service discovery is difficult in a modern, cloud-based microservices application because the set of instances, and their IP addresses, are subject to constant change

Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism.

There are two main service discovery patterns: client-side discovery and server-side discovery. Let's first look at client-side discovery.

The Client-Side Discovery Pattern

When using client-side discovery, the client is responsible for determining the network locations of available service instances and load balancing requests across

them. The client queries a service registry, which is a database of available service instances. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

The following diagram shows the structure of this pattern.

With client-side service discovery, the client determines the network locations of available service instances and load balances requests across them

The network location of a service instance is registered with the service registry when it starts up. It is removed from the service registry when the instance terminates. The service instance's registration is typically refreshed periodically using a heartbeat mechanism.

Netflix OSS provides a great example of the client-side discovery pattern. Netflix Eureka is a service registry. It provides a REST API for managing service-instance registration and for querying available instances. Netflix Ribbon is an IPC client that works with Eureka to load balance requests across the available service instances. We will discuss Eureka in more depth later in this article.

The client-side discovery pattern has a variety of benefits and drawbacks. This pattern is relatively straightforward and, except for the service registry, there are no other moving parts. Also, since the client knows about the available services instances, it can make intelligent, application-specific load-balancing decisions such as using hashing consistently. One significant drawback of this pattern is that it couples the client with the service registry. You must implement client-side service discovery logic for each programming language and framework used by your service clients.

Now that we have looked at client-side discovery, let's take a look at server-side discovery.

The Server-Side Discovery Pattern

The other approach to service discovery is the server-side discovery pattern. The following diagram shows the structure of this pattern.

With the server-side service discovery, the load balancer queries a service registry about service locations; clients interact only with the load balancer

The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

The AWS Elastic Load Balancer (ELB) is an example of a server-side discovery router. An ELB is commonly used to load balance external traffic from the Internet. However, you can also use an ELB to load balance traffic that is internal to a virtual private cloud (VPC). A client makes requests (HTTP or TCP) via the ELB using its DNS name. The ELB load balances the traffic among a set of registered Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers. There isn't a separate service registry. Instead, EC2 instances and ECS containers are registered with the ELB itself.

HTTP servers and load balancers such as NGINX Plus and NGINX can also be used as a server-side discovery load balancer. For example, this blog post describes using Consul Template to dynamically reconfigure NGINX reverse proxying. Consul Template is a tool that periodically regenerates arbitrary configuration files from configuration data stored in the Consul service registry. It runs an arbitrary shell command whenever the files change. In the example described by the blog post, Consul Template generates an nginx.conf file, which configures the reverse proxying, and then runs a command that tells NGINX to reload the configuration. A more sophisticated implementation could dynamically reconfigure NGINX Plus using either its HTTP API or DNS.

Some deployment environments such as Kubernetes and Marathon run a proxy on each host in the cluster. The proxy plays the role of a server-side discovery load balancer. In order to make a request to a service, a client routes the request via the proxy using the host's IP address and the service's assigned port. The proxy then transparently forwards the request to an available service instance running somewhere in the cluster.

The server-side discovery pattern has several benefits and drawbacks. One great benefit of this pattern is that details of discovery are abstracted away from the client. Clients simply make requests to the load balancer. This eliminates the need to implement discovery logic for each programming language and framework used by your service clients. Also, as mentioned above, some deployment environments provide this functionality for free. This pattern also has some drawbacks, however.

Unless the load balancer is provided by the deployment environment, it is yet another highly available system component that you need to set up and manage.

The Service Registry

The service registry is a key part of service discovery. It is a database containing the network locations of service instances. A service registry needs to be highly available and up to date. Clients can cache network locations obtained from the service registry. However, that information eventually becomes out of date and clients become unable to discover service instances. Consequently, a service registry consists of a cluster of servers that use a replication protocol to maintain consistency.

As mentioned earlier, Netflix Eureka is good example of a service registry. It provides a REST API for registering and querying service instances. A service instance registers its network location using a POST request. Every 30 seconds it must refresh its registration using a PUT request. A registration is removed by either using an HTTP DELETE request or by the instance registration timing out. As you might expect, a client can retrieve the registered service instances by using an HTTP GET request.

Netflix achieves high availability by running one or more Eureka servers in each Amazon EC2 availability zone. Each Eureka server runs on an EC2 instance that has an Elastic IP address. DNS TEXT records are used to store the Eureka cluster configuration, which is a map from availability zones to a list of the network locations of Eureka servers. When a Eureka server starts up, it queries DNS to retrieve the Eureka cluster configuration, locates its peers, and assigns itself an unused Elastic IP address.

Eureka clients – services and service clients – query DNS to discover the network locations of Eureka servers. Clients prefer to use a Eureka server in the same availability zone. However, if none is available, the client uses a Eureka server in another availability zone.

Other examples of service registries include:

etcd – A highly available, distributed, consistent, key-value store that is used for shared configuration and service discovery. Two notable projects that use etcd are Kubernetes and Cloud Foundry.

consul – A tool for discovering and configuring services. It provides an API that

allows clients to register and discover services. Consul can perform health checks to determine service availability.

Apache Zookeeper – A widely used, high-performance coordination service for distributed applications. Apache Zookeeper was originally a subproject of Hadoop but is now a top-level project.

Also, as noted previously, some systems such as Kubernetes, Marathon, and AWS do not have an explicit service registry. Instead, the service registry is just a built-in part of the infrastructure.

Now that we have looked at the concept of a service registry, let's look at how service instances are registered with the service registry.

Service Registration Options

As previously mentioned, service instances must be registered with and deregistered from the service registry. There are a couple of different ways to handle the registration and deregistration. One option is for service instances to register themselves, the self-registration pattern. The other option is for some other system component to manage the registration of service instances, the third-party registration pattern. Let's first look at the self-registration pattern.

The Self-Registration Pattern

When using the self-registration pattern, a service instance is responsible for registering and deregistering itself with the service registry. Also, if required, a service instance sends heartbeat requests to prevent its registration from expiring. The following diagram shows the structure of this pattern.

With the self-registration pattern for service discovery, a service instance registers and deregisters itself with the service registry

A good example of this approach is the Netflix OSS Eureka client. The Eureka client handles all aspects of service instance registration and deregistration. The Spring Cloud project, which implements various patterns including service discovery, makes it easy to automatically register a service instance with Eureka. You simply annotate your Java Configuration class with an @EnableEurekaClient annotation.

The self-registration pattern has various benefits and drawbacks. One benefit is that it is relatively simple and doesn't require any other system components. However, a major drawback is that it couples the service instances to the service registry. You

must implement the registration code in each programming language and framework used by your services.

The alternative approach, which decouples services from the service registry, is the third-party registration pattern.

The Third-Party Registration Pattern

When using the third-party registration pattern, service instances aren't responsible for registering themselves with the service registry. Instead, another system component known as the service registrar handles the registration. The service registrar tracks changes to the set of running instances by either polling the deployment environment or subscribing to events. When it notices a newly available service instance it registers the instance with the service registry. The service registrar also deregisters terminated service instances. The following diagram shows the structure of this pattern.

With the third-party registration pattern for service discovery, a separate service registrar registers and deregisters service instances with the service registry

One example of a service registrar is the open source Registrator project. It automatically registers and deregisters service instances that are deployed as Docker containers. Registrator supports several service registries, including etcd and Consul.

Another example of a service registrar is NetflixOSS Prana. Primarily intended for services written in non-JVM languages, it is a sidecar application that runs side by side with a service instance. Prana registers and deregisters the service instance with Netflix Eureka.

The service registrar is a built-in component of deployment environments. The EC2 instances created by an Autoscaling Group can be automatically registered with an ELB. Kubernetes services are automatically registered and made available for discovery.

The third-party registration pattern has various benefits and drawbacks. A major benefit is that services are decoupled from the service registry. You don't need to implement service-registration logic for each programming language and framework used by your developers. Instead, service instance registration is handled in a centralized manner within a dedicated service.

One drawback of this pattern is that unless it's built into the deployment environment, it is yet another highly available system component that you need to set up and manage.

Summary

In a microservices application, the set of running service instances changes dynamically. Instances have dynamically assigned network locations. Consequently, in order for a client to make a request to a service it must use a service-discovery mechanism.

A key part of service discovery is the service registry. The service registry is a database of available service instances. The service registry provides a management API and a query API. Service instances are registered with and deregistered from the service registry using the management API. The query API is used by system components to discover available service instances.

There are two main service-discovery patterns: client-side discovery and service-side discovery. In systems that use client-side service discovery, clients query the service registry, select an available instance, and make a request. In systems that use server-side discovery, clients make requests via a router, which queries the service registry and forwards the request to an available instance.

There are two main ways that service instances are registered with and deregistered from the service registry. One option is for service instances to register themselves with the service registry, the self-registration pattern. The other option is for some other system component to handle the registration and deregistration on behalf of the service, the third-party registration pattern.

In some deployment environments you need to set up your own service-discovery infrastructure using a service registry such as Netflix Eureka, etcd, or Apache Zookeeper. In other deployment environments, service discovery is built in. For example, Kubernetes and Marathon handle service instance registration and deregistration. They also run a proxy on each cluster host that plays the role of server-side discovery router.

An HTTP reverse proxy and load balancer such as NGINX can also be used as a server-side discovery load balancer. The service registry can push the routing

information to NGINX and invoke a graceful configuration update; for example, you can use Consul Template. NGINX Plus supports additional dynamic reconfiguration mechanisms – it can pull information about service instances from the registry using DNS, and it provides an API for remote reconfiguration.

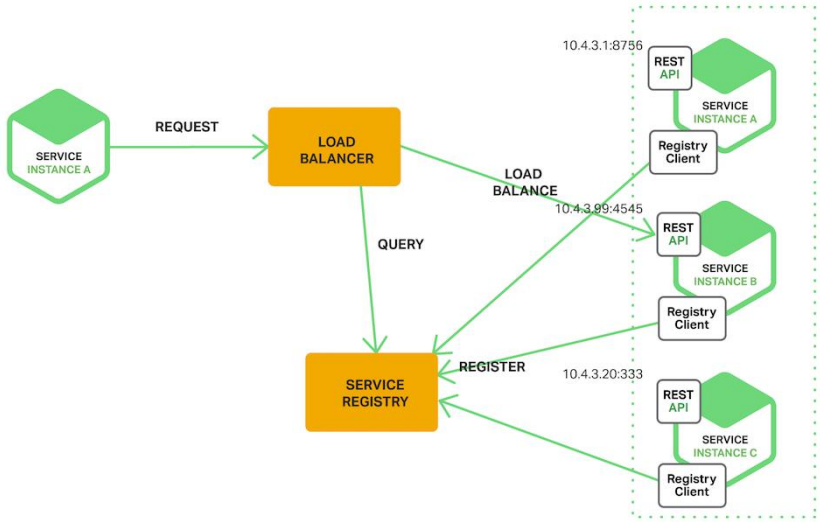
In future blog posts, we'll continue to dive into other aspects of microservices. Sign up to the NGINX mailing list (form is below) to be notified of the release of future articles in the series.

英文文献翻译
如何处理变化无常的容器网络位置?

为什么要使用“服务发现”?

让我们设想一个场景:你正在写一些需要调用 REST API 或者 Thrift API 服务的代码。为了发起一个请求,你的代码需要知道这个服务实例(service instance)在网络上的位置(IP 地址和相应的 port)。在传统运行于物理机上的应用中,服务实例的网络位置是相对固定的。例如,你的代码可以从一个偶尔更新的配置文件中获取网络位置。

然而,在一个现代的、基于云的微服务应用中,这是一个当然难以解决的问题,如下图所示:



服务实例的网络位置是被动态分配的。此外,服务实例集由于自动扩容、故障和升级时常动态改变。而其结果就是,你的客户端代码需要使用一个更加“精心设计”的服务发现机制。

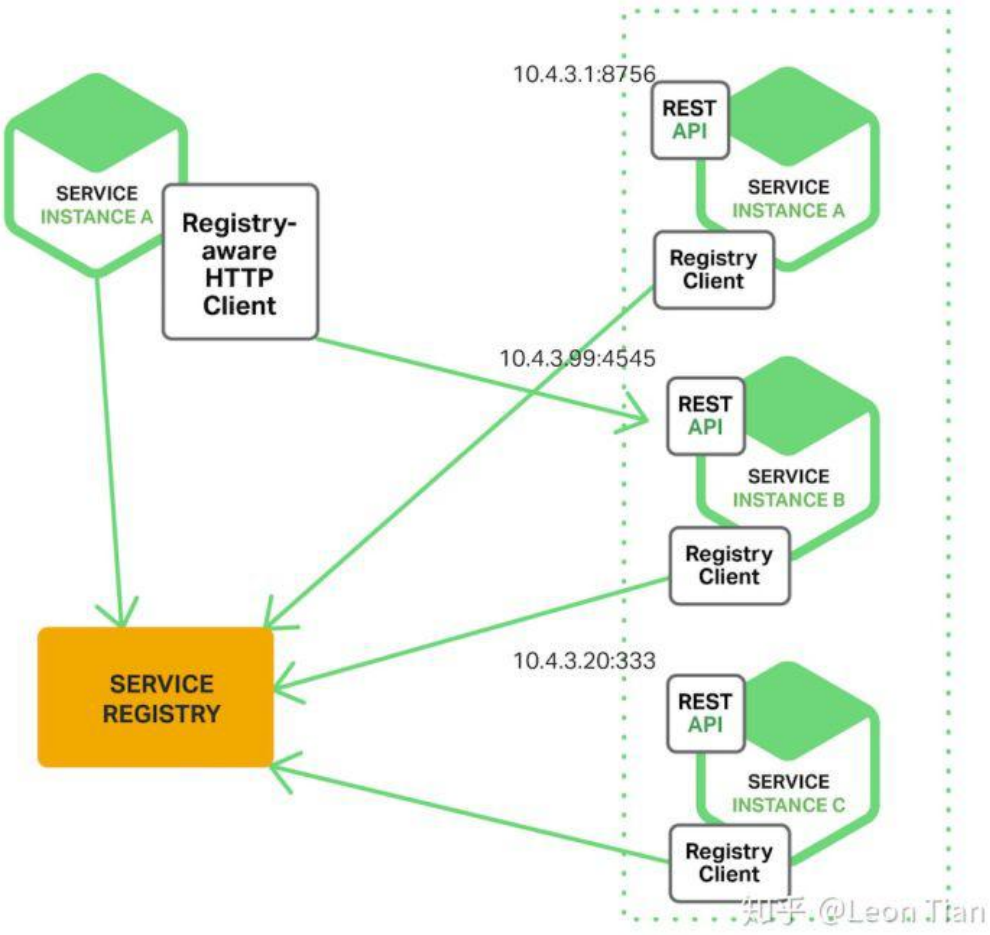
目前有两种主要的服务发现模型(service discovery pattern): 客户端侧发现(client-side discovery , 以下简称“客户端发现”)和服务端侧发现(server-side discovery , 以下简称"服务端发现")。让我们先来看客户侧发现。

客户端发现模式(The Client-Side Discovery Pattern)

当使用客户端发现时,客户端负责决定网络位置之间的可用服务实例和负载均衡请求。客户端需要一个服务注册器(service registry),其实它就是一个存储

着可用服务实例的数据库。之后，客户端使用某种负载均衡算法来选择可用实例中的一个并发起请求。

下图展示了这种模型的结构：



当服务实例启动时，服务实例的网络位置就被服务注册器注册了。当服务实例结束时，相应信息就被从服务注册器中移除。服务实例的注册可以使用心跳机制(heartbeat mechanism)这种经典的定期更新机制。

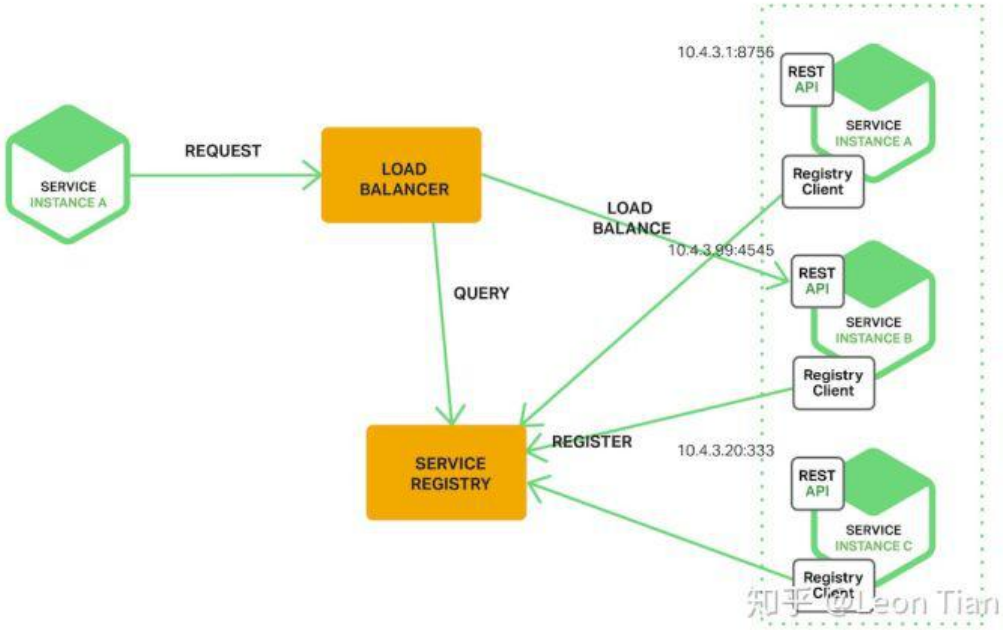
Netflix OSS 提供了一个很棒的客户端发现模型的案例。Netflix Eureka 是一个服务注册器。它提供一个 REST API 来管理服务实例的注册和查询可用实例。Netflix Ribbon 是一个 IPC 客户端，它与 Eureka 一起工作，在可用服务实例之间将请求负载均衡。我们将在后面更加深入地讨论 Eureka 。

客户端发现模式有很多好处和不足。这种模式相当直接，而且除了服务注册器以外，就没有其他需要改动的部分了。此外，因为服务端知道所有的可用实例，

因此它可以做出高明的、精确到具体应用的负载均衡决策，比如持续地使用哈希算法。但是这种模式的一个重要不足则是它将客户端和注册器耦合在一起。你必须根据你的客户端，使用特定的编程语言和框架来亲手实现客户端的服务发现逻辑。

服务端发现模式(The Server-Side Discovery Pattern)

另一种服务发现方法是服务端发现模式。下图展示了这种模式的结构：



客户端通过一个负载均衡器向服务发起一个请求。负载均衡器查询服务注册器并路由每一个请求至相应的可用服务实例。和客户端发现相同，服务实例的注册和注销都通过服务注册器。

亚马逊弹性负载均衡器(AWS Elastic Load Balancer , ELB)是服务端发现路由器的一个例子。一个 ELB 通常被用于进行从互联网来的外网传输的负载均衡。然而。你也可以用 ELB 对虚拟私有云(virtual private cloud , VPC)的内部进行传输的负载均衡。一个客户端通过 ELB 使用它的 DNS 名发起请求(HTTP 或 TCP)。ELB 对注册过的 EC2 实例(Elastic Compute Cloud instance , 弹性计算云实例) 或 ECS 容器(EC2 Container Service container) 进行传输的负载均衡。注意，这里并没有一个分离出来的服务注册器，取而代之的是 EC2 实例和 ECS 容器都被 ELB 本身注册。

HTTP 服务和负载均衡器比如 Nginx Plus 和 Nginx 同样能被当作服务端

发现的负载均衡器。比如，这篇博客详细阐述了使用 Consul 模型来动态地再配置 Nginx 反向代理。Consul 模型是一个从存储在 Consul 服务注册器(Consul service registry)中获取数据的、定时再启动的、可任意配置的配置文件。每当文件被改变时，它可以运行任何 shell 命令。在这个例子中，Consul 框架生成了一个 nginx.conf 文件，它配置了反向代理，并运行一个让 Nginx reload 配置文件的命令。能动态再配置 Nginx Plus 的更复杂的实现使用了 HTTP API 或 DNS。

一些诸如 Kubernetes 和 Marathon 的可部署环境在集群的每个主机上运行一个 proxy 。这个 proxy 扮演着“服务端发现的负载均衡器”的角色。为了向服务发起请求，客户端通过使用着主机 IP 和相应服务 port 的 proxy ，按照这个 proxy 指定的线路发送请求。之后，这个 proxy 对于运行在集群某处的可用服务实例就变成透明的了。

同样，服务端发现模式也有一些好处和不足。这种模式的一个最大好处就是，服务发现的细节被从服务端抽象出来了。客户端仅仅向负载均衡器发起请求。这消除了客户端上“使用具体编程语言和框架来实现服务发现”的逻辑。此外，正如上面所说的，一些可部署环境(K8s 等)免费提供了这种功能。当然，这种模式也有它自己的不足。除非负载均衡器被可部署环境所提供，否则它又是另一个需要你搭建和管理的、需要高可用性的系统组件。

服务注册器

服务注册器是服务发现的关键。它其实是一个保存着服务实例网络位置的数据库。一个服务注册器需要高可用性和实时更新。传统模式下，客户端可以缓存从注册器获取到的网络位置信息。然而，这些信息最终会失效而且服务端不再能发现服务实例。结果就是，我们需要一个“由使用备份机制的服务器集群组成的“服务注册器，来持续地维护着网络位置信息。

前面说过，Netflix Eureka 就是一个很好的服务注册器例子。它为服务实例的注册和查询提供了一个 REST API 。一个服务实例使用 POST 请求注册它的网络位置。每 30 秒它必须通过 PUT 请求刷新自己的注册信息。一项注册过的信息会被 DELETE 请求或由于注册信息超时 (比如服务实例超过 30 秒未更新，则服务注册器判定实例已经消亡)而被移除。你可能已经猜到了，客户端可以通过使用 GET 请求取回已经注册的服务实例。

Netflix 通过在每台 Amazon EC2 可用区域中运行一个或多个 Eureka 服务来实现高可用性。每一个在 EC2 实例上运行的 Eureka 服务都拥有一个弹性 IP

地址(Elastic IP address)。DNS TEXT 记录被用于存储 Eureka 集群的配置信息，即从 EC2 可用区域到 Eureka 服务网络位置列表的一个映射。当一个 Eureka 服务启动时，它请求 DNS 取回 Eureka 集群的配置信息，定位自己的 peers，并分配给自己一个未使用的弹性 IP 地址。

Eureka 客户端——服务和服务的客户端——请求 DNS 来发现 Eureka 服务的网络位置。客户端倾向于在相同的可用区域使用 Eureka 服务。然而，如果区域中没有可用的，客户端会在其他的可用区域使用 Eureka 服务。

其他关于服务注册器的例子包括：

etcd: 一个高可用，分布式，持久化，键-值存储的服务注册器，常被用于分享配置信息以及服务发现。两个使用 etcd 的著名项目为 Kubernetes 和 Cloud Foundry。

consul: 一个用于服务发现和配置的工具。它提供了一个 API ，它允许客户端注册并发现服务。consul 能执行健康检查(health check)并判断服务的可用性。

Apache Zookeeper: 一个被广泛应用的、高性能的、用于分布式应用的协调器。Apache Zookeeper 起源于 Hadoop 的子项目，但现在已经是一个顶级项目了。

同样，如上所述，一些诸如 Kubernetes, Marathon 和 AWS 的系统不会拥有实现一个具体的服务注册器。相反，服务注册器是组织结构的一个内建部分(即: Kubernetes 不会用代码亲自实现服务注册功能，而是将 etcd 集成进自己的结构中从而实现服务发现的功能)。

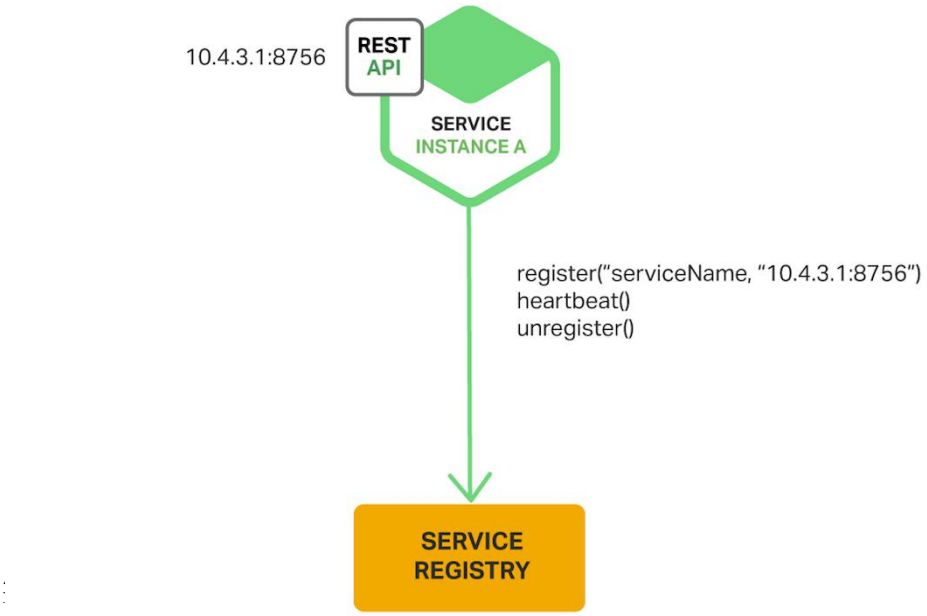
至此，我们已经明白了服务注册器的概念，下面让我们看看服务实例是如何被服务注册器注册的。

服务注册信息的选择

之前提过，服务实例必须被服务注册器注册或注销。而处理注册和注销有许多种不同方法。有一种选择是服务实例自己注册自己，我们称之为“自注册模式(self-registration pattern)”。另一个选择是使用其他系统组件来管理服务实例的注册信息，我们称之为“第三方注册模式(third-party registration pattern)”。让我们先来看看自注册模式。

自注册模式

当使用自注册模式时，一个服务实例向服务注册器的注册和注销都由它自己负责。同样地，如果需要，服务实例发送心跳请求(heartbeat requests)以防它的注册信息过期。下图展示了这种模式的结构：

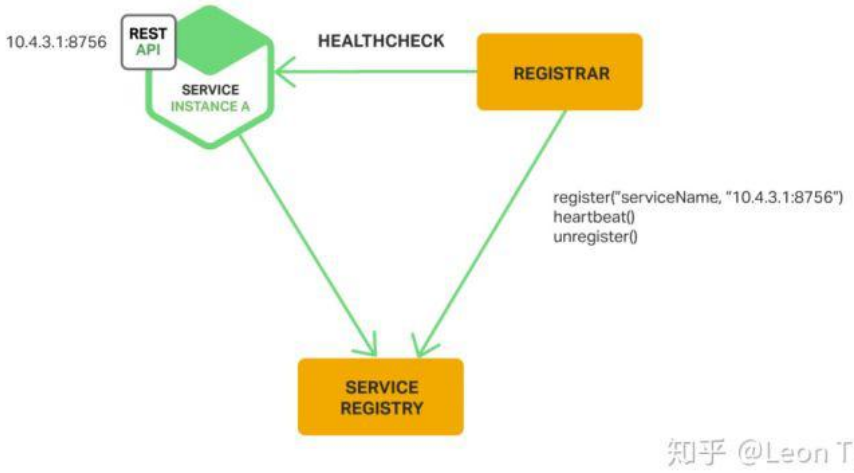


自注册模式有很多好处和不足。其最大的好处，就是它相当简单而且不需要其他任何系统组件。然而，它的主要缺点就是将服务实例和服务注册器耦合了。在你的服务中，你必须亲手用特定的代码和框架来实现用于注册的代码。

而另一种可选方法，将服务从服务注册中解耦，就是第三方注册模式。

第三方注册模式

当使用第三方注册模式时，服务实例并不负责向服务注册器注册它们自己。相仿，另一个被称为 “服务注册助理(service registrar)” 的系统组件处理这些注册。服务注册助理通过“检测部署环境(polling the deployment environment)”或“订阅事件(subscribing events)”的方式，跟踪所有运行实例的变化。当发现一个新的可用服务实例时，服务注册助理使用服务注册器注册这个实例。同样，服务注册助理也会注销已经结束掉的服务实例。下图展示了这种模式的结构：



服务注册助理的一个例子是开源项目 Registrator。它自动地注册和注销使用 docker 容器部署的服务实例。Registrator 只需很多种服务注册器，包括 etcd 和 Consul。

另一个服务注册助理的例子是 NetflixOSS Prana。主要为那些使用无 JVM (non-JVM)的编程语言所写的服务，Prana 是一个跑在服务身边的 sidecar 应用 (类似于 K8s 的 Pod 中同时运行一个微服务应用和一个代理应用)。Prana 使用 Netflix Eureka 注册和注销服务实例。

服务注册助理是部署环境的一个内建组件。由自扩容组(Autoscaling Group) 创建的 EC2 实例都可以被 ELB 自动注册。Kubernetes 的服务是自动注册并为了被发现而自动变得可用的。

第三方注册模式同样有很多好处和不足。主要的好处是服务与服务注册器解耦。你不用让你的开发者亲手使用特定语言和框架实现它。取而代之的，是服务实例的注册都被一个集中控制的专用型服务(dedicated service)所处理。

这种模式的一个主要缺点是，除非它被内置在部署环境中(比如 K8s 就将其内建在其中)，否则还需要你去搭建并维护另一个高可用的系统组件。

总结

在微服务应用中，运行的服务实例集动态地变化。实例的网络位置动态地被分配。而其结果就是，为了让客户端向服务发起请求，我们必须使用“服务发现

机制”。

服务发现的关键则是“服务注册器”。这个服务注册器通常是一个记录着可用服务实例的数据库。服务注册器提供“用于管理的 API ”和“用于查询的 API ”。服务实例在服务注册器上的注册和注销使用“用于管理的 API ”。而“用于查询的 API ”被用作系统组件，来发现可用服务实例。

其实就是服务实例的注册和注销使用“用于管理的 API”，比如 post 和 delete。普通客户端想要访问一个服务，就使用“用于查询的 API”，比如 get 请求，来获取一个服务的 IP:port，从而进行访问。
译者注，方便您理解。

主要有两种主流的服务发现模式：“客户端发现”和“服务端发现”。在使用客户端发现的系统中，客户端查询服务注册器，选择一个可用实例并发起请求。在使用服务端发现的系统中，客户端通过一个路由器(或者我们称为反向代理)发起请求，而这个路由器去查询服务注册器并将请求转发至可用服务实例。

主要有两种服务实例向服务注册器注册和注销的方式。一种为“自注册模式”，它由服务实例自己向服务注册器进行注册。另一种为“第三方注册模式”，即使用其他系统组件来代表服务处理它自己的注册和注销。

在一些可部署环境中，你需要使用服务注册器搭建你自己的服务发现组织结构，诸如 Netflix Eureka，etcd 或 Apache Zookeeper。在其他可部署环境中，服务发现是内建在其中的，比如我们熟知的 Kubernetes 和 Marathon 就能处理服务实例的注册和注销。它们同样在集群的每一台主机上运行一个 proxy 来充当“服务端发现”中路由器的角色。

一个 HTTP 反向代理和负载均衡器比如 Nginx 同样可以被用作一个“服务端发现”的负载均衡器。服务注册器可以将路由信息推送至 Nginx 并激活一次优雅的配置文件升级，比如你可以使用 Consul 框架(Consul Template)。Nginx Plus 支持“可添加的动态再配置机制(additional dynamic reconfiguration mechanisms)”，它可以从注册器使用 DNS 推送服务实例的信息，并提供一个用于远程再配置的 API。

在之后的文章中，我们将会继续深入其他微服务的方面。订阅 Nginx 邮件列表(在下面)，在本系列的未来文章所发表时通知您。

英文文献翻译

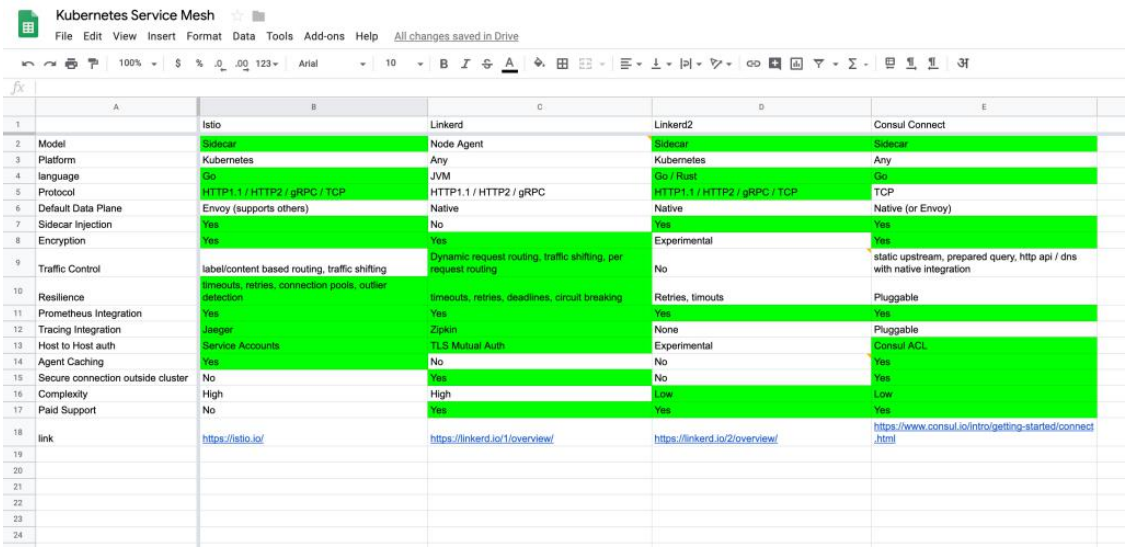
The Battle between Service Meshes

We’re going to compare every Kubernetes service mesh available today and work out who the winner is. You may have already read our Top10 list of Kubernetes applications in which case the result may be somewhat predictable.

If you’ve arrived on this page you probably already understand what a service mesh does. If you don’t then go and quickly read this article and then come back.

In this blog we’ll hopefully help you to choose from the four options available today. Like all of the content on this site it will be 100% Kubernetes specific.

To help inform people about what service mesh to choose I’ve put together a quick table of features.



	Istio	Linkerd	Linkerd2	Consul Connect
Model	Sidecar	Node Agent	Sidecar	Sidecar
Platform	Kubernetes	Any	Kubernetes	Any
language	Go	JVM	Go / Rust	Go
Protocol	HTTP1.1 / HTTP2 / gRPC / TCP	HTTP1.1 / HTTP2 / gRPC	HTTP1.1 / HTTP2 / gRPC / TCP	TCP
Default Data Plane	Envoy (supports others)	Native	Native	Native (or Envoy)
Sidecar Injection	Yes	No	Yes	Yes
Encryption	Yes	Yes	Experimental	Yes
Traffic Control	label/content based routing, traffic shifting	Dynamic request routing, traffic shifting, per request routing	No	static upstream, prepared query, http api / dns with native integration
Resilience	timeouts, retries, connection pools, outlier detection	timeouts, retries, deadlines, circuit breaking	Retries, timouts	Pluggable
Prometheus Integration	Yes	Yes	Yes	Yes
Tracing Integration	Jaeger	Zipkin	None	Pluggable
Host to Host auth	Service Accounts	TLS Mutual Auth	Experimental	Consul ACL
Agent Caching	Yes	No	No	Yes
Secure connection outside cluster	No	Yes	No	Yes
Complexity	High	High	Low	Low
Paid Support	No	Yes	Yes	Yes
link	https://istio.io/	https://linkerd.io/1/overview/	https://linkerd.io/2/overview/	https://www.consul.io/intro/getting-started/connect.html

Many of these are so new that the documentation is lagging a little behind. If you can help fill in the gaps please add a comment to this spreadsheet and I’ll update this blog accordingly.

Linkerd

I used Linkerd extensively on DC/OS and absolutely loved it. However, times have changed and there are a couple of fundamental problems that have caused this to be a total dead-end on Kubernetes.

Linkerd is written in a JVM language which means a footprint of 110mb+ memory usage per node agent. This isn't too bad when you just run one node agent per host, but the world is moving to per pod proxy sidecars, and I think everyone realised this is too much overhead.

Linkerd also doesn't proxy TCP requests and doesn't support websockets.

On the positive end of the scale Linkerd has absolutely amazing traffic control. Read some of the documentation around Namerd and you'll see just how advanced and powerful it is. It's also one of the two service meshes that supports connections outside of the cluster.

So in summary I'd say if you only have Kubernetes to worry about then give Linkerd a miss. If you have Linkerd already in other areas and need to connect services on your Kubernetes cluster to them then it may be a valid option.

Linkerd2 (formerly Conduit)
Linkerd2 is a total rewrite of Linkerd in Golang and Rust specifically for Kubernetes. Unfortunately, as with every rewrite, you start back at the beginning again from a feature and stability perspective. Although I'm sure there are more than a few lessons learned.

Moving to Rust for the data plane proxy sidecars should help mitigate some of the bugs and should also solve the memory issues. It also supports all of the major protocols now which is a big step forward.

One interesting difference compared to other service mesh designs is the tight default coupling between the data plane and control plane services. This simplifies the configuration which I see as a positive. I also like how there is a focus on keeping the data plane latency P99's extremely low.

As mentioned at the beginning though the project just isn't at a level from a feature perspective where it can compete with something like Istio. To give just one example of something I'd consider to be fundamentally required from a service mesh: distributed tracing. This is still in in the planning stage for Linkerd2. There are many other features that other blogs have called 'table stakes' that seem to be still in the RFC stage.

Having said this, if you try Linkerd2 and are happy with the current feature set then this seems like a good investment for the future. Many people hate the high complexity of Istio and so I think over time this may become the most compelling option if it remains simple.

Update: Recent updates include sidecar injection, timeouts and retries.

Consul

The latest version of Consul now comes with the ‘connect‘ feature which can be enabled on existing clusters. Like with most of the Hashicorp tools Consul is a single Go binary that includes both the data and control plane. The main unique selling point seems to be that you can enable connect across services on Kubernetes and join them to services on vm’s outside that also run Consul. This might be attractive to some organisations. However, I don’t really see it as a big advantage based on work I’ve done in the past. Usually we leave the legacy alone and let it die, then work on new projects or migrate stuff onto Kubernetes.

Consul does seem to have a slight architectural advantage in that it operates as a full mesh with no centralised control plane services that could theoretically act as a performance bottleneck.

There is also a neat separation between layer4 and layer7. I think this separation may keep the Consul service mesh design simple while still allowing for the data layer to be split out. You can currently switch out the default data layer proxy with Envoy if you need more layer 7 features.

The default proxy is however quite lacking in features. To get tracing support, or many of the more advanced layer 7 features, you’ll need to swap out the proxy for something like Envoy. This isn’t very well documented online.

The other part to keep an eye on is the control plane configuration. Istio is notoriously complicated to configure at this layer and I see Consul has a simple ‘service access graph’ feature.

Hashicorp have blogged about differentiating in the area of security. Consul ACL’s providing host to host security is a very nice feature. Especially if you want to connect pods from inside Kubernetes outside the cluster in a secure way. The agent caching, especially for auth, apparently makes the communication performance excellent.

So just like Linkerd2 this is another one to watch. Consul connect was only released a few weeks ago and so there really aren't many howto guides online. If you're already highly invested in the Hashicorp toolchain then I'd trial this and perhaps learn about how to swap out the default proxy with Envoy.

Istio

Istio is stable and feature rich. At the time of writing Istio has 11.5k Github stars, 244 contributors and is backed by Lyft, Google and IBM. Istio has pioneered many of the ideas currently being emulated by other service meshes.

One such stand-out-feature is the automatic sidecar injection which works amazingly well with Helm charts.

There are of course some negatives which are all to do with modularity, plug-ability and ultimately complexity. You can switch out almost any component of Istio and integrate it with other systems. This all comes at the cost of a steep learning curve and plenty of scope to shoot yourself in the foot.

However, surprisingly, you can get up and running with Istio very quickly if you stick to the defaults. Configuring a test instance with minikube, helm and Istio on your laptop is less than 5 minutes of work. There are also thousands of articles online for how to configure other integrations. This is a stark contrast to the other service meshes.

The winner: Istio

It's close but I'd say if you're starting from scratch on Kubernetes which many people are then Istio is probably the best service mesh right now. The complexity is high, but not massively high when compared to what you have to manage with Kubernetes already. It has the most features and went version 1 and production ready a few months ago. It has also got the backing of Google and a massive community churning out cool blogs and integrations.

The end..

Or probably not. Perhaps this comparison was too premature. It's nice to have a competitive landscape with software and hopefully one day I can revisit this list and crown a new winner.

英文文献翻译

If anything I’ve written is technically inaccurate please drop me a message below.

To keep updated as new service meshes are released sign up for our news letter and visit our service mesh category.

英文文献翻译
四种 Service Mesh 各种特性大比拼

今天我们来比一比每个可用于 Kubernetes 的 Service Mesh 并评选出谁是最棒的。若你已经读过我们的“Top 10 Kubernetes 应用清单”，那么大概你已经能猜出来评选结果了。

阅读本页你时可能已经明白了什么是 Servicee Mesh 。如果你不太明白，就先去速览一下这篇文章然后再回来看。

在本文中，我们希望帮你从四位选手中选出目前最好的。和这个网站上所有的内容一样，这篇文章也会 100% 与 Kubernetes 有关。

为了帮大家更直观的看到你该选哪个 Service Mesh，我把这四个选手的特性放在一起做了张表：

	Istio	Linkerd	Linkerd2	Consul Connect
开发模型	Sidecar	Node Agent	Sidecar	Sidecar
运行平台	K8s	任意	K8s	任意
开发语言	Go	JVM	Go/ Rust	Go
网络协议	HTTP1.1/ HTTP2/ gRPC/ TCP	HTTP1.1/ HTTP2/ gRPC	HTTP1.1/ HTTP2/ gRPC/ TCP	TCP
默认数据面	Envoy	原生	原生	原生或Envoy
Sidecar 注入	Yes	No	No	No
加密	Yes	Yes	实验阶段	Yes
传输控制	基于标签&内容的路由/ 传输切换	动态请求路由/ 传输切换/ 针对单个请求路由	?	静态上游/ 预准备请求/ 原生的API&dns
容错机制	超时/ 重试/ 连接池/ 异常检测	超时/ 重试/ deadlines/ 熔断器	?	可插拔式
Prometheus 集成	Yes	Yes	Yes	No
跟踪机制集成	Jaeger	Zipkin	无	可插拔式
主机间的权限认证	Service Accounts	TLS Mutual Auth	无	Consul ACL
代理缓存	Yes	No	No	Yes
集群外部安全连接	No	Yes	No	Yes
复杂度	高	高	低	低
付费支持	无	有	有	知乎 @Leon Tian

Istio , Linkerd , Linkerd2 和 Consul Connect 四种 Service Mesh 对比

注：标注颜色的项目是在本条特性上具有优势的项目

它们中的很多特性还很新，以至于相关文档还没来得及更新。如果你能帮忙完善这张表格请在评论区留言，届时我会更新这张表格。

Linkerd

关键字:

最早实现 Service Mesh 故部分设计现在显得落后

内存开销大

不支持 TCP 请求

能够在集群内外进行通信

译者注。

我曾在 DC/OS 上大量应用 Linkerd 而且很爱用它。然而，时过境迁导致它有很多功能性缺陷并使得它不能在 Kubernetes 上得以应用。

Linkerd 基于 JVM 语言编写，意味着它在每个代理节点上至少占用 110 mb+ 内存。当你的主机上只运行一个代理节点时还不算太糟，但是如今每个 Pod 里都要用 Sidecar proxy 了，而且我发现每个人意识到这点后都有点上头。

Linkerd 也不支持 TCP 请求因此不能支持 websockets。

Linkerd 的在这几项里的最大亮点就是传输控制。读点关于 Namerd 的文档你就会发现这有多么先进和强大。它也是这四者中唯二支持集群外部连接的 Service Mesh 实现。

结论就是如果你使用 Kubernetes，那么就放弃 Linkerd 吧。但如果你已经在其他地方使用了 Linkerd 并需要用它来连接你 Kubernetes 集群里的服务，那么 Linkerd 将会是个不错的选择。

Linkerd2（ 官方叫法为 Conduit ）

关键字:

出现时间较晚，功能仍未齐全

数据面与控制面紧耦合，使配置简单

复杂度低

译者注。

Linkerd2 使用 Go 语言和 Rust 完全重构，为 Kubernetes 量身打造。不幸的是，随着每一次代码重写，你的功能特性和稳定性的角度都得从头开始，虽然我确信这样能让我学到更多的经验和教训。

使用 Rust 开发数据面的 Sidecar proxy 应该能在减少一些 bug 的同时解决内存问题。一个巨大的飞跃是，它现在支持所有主流协议。

与其他 Service Mesh 相比，有点一非常有趣的是，它在设计时默认将控制面和数据面紧耦合。这样做的优点是简化了配置。我同样喜欢它专注在保持数据面延迟 P99 在相当低的水平。

如开头所说，这个项目目前在功能特性的角度与 Istio 还不在一个级别。举一个我能想到的 Service Mesh 功能性需求简单例子：分布式追踪。这个功能还在 Linkerd2 的计划阶段。其他的很多特性，比如其他博客里说的“table stackes”似乎还在 RFC 阶段。

我会努力地猜测 Linkerd2 在功能特性和稳定性的角度上趋于成熟并追上 Istio 要花多久。可能要花好几年吧。

说到这里，如果你试用 Linkerd2 而且感觉它的功能特性都还不错，那么这可能是对未来的一笔不错的投入。很多人恨透了 Istio 的高复杂度，所以我觉得以后 Linkerd2 会变得非常引人入胜，前提是它保持较低的复杂度。

再编辑： 在 Linkerd2 加入 GA 后我又把这一段复读了一遍。上面的特性对比表如我所料是正确的，但是文档内容太少了。我希望大家能提供一些关于它所支持的传输控制和容错机制的细节。

Consul

关键词：
默认的原生 Sidecar 性能较低
文档极少

配置简单，复杂度地
集群内外通讯
权限机制较为完善
译者注。

Consul 的全新版本已经支持“连接”特性，它能被已经存在的集群所使用。

像大多数 Hashicorp 工具一样， Consul 是个单一的 Go 语言二进制文件，同时包含了数据面和控制面。它的主要卖点似乎是你能够在 Kubernetes 中跨服务建立连接，并将它们加入虚拟机外部跑着 Consul 的地方。这可能会对一些组织很有吸引力。然而，根据我以往的工作经验，我并不觉得这是很大的优势。通常我们将遗留资源放在一边并让它自己覆灭，之后我们减少 Kubernetes 上的容量或在新的项目上工作。

Consul 在它操作所有网格而不使用中心化的控制面服务(理论上来讲可能成为性能瓶颈)上，似乎确实有着轻量级架构的优势。

在 L4 和 L7 的拆分上它也很牛。我认为这种分离可能会保持 Consul 的简单设计且仍然保持数据层的分离。现在你可以切换到默认的数据层代理 Envoy 上，如果你需要 L7 的某些特性的话。

然而默认的代理在功能特性上面还是有所欠缺。为了得到追踪支持，或者其他更高级 L7 特性，你会将 proxy 换成其他的，比如 Envoy。在这方面，在线文档写的并不清楚。

另一个引人注目的部分是控制面的配置。在控制面上，Istio 配置的麻烦是出了名的，而且我看到 Consul 有一个简单的 "服务通道图表"特性。

Hashicorp 已经在关于安全方面的区别写了博客。Consul 的 ACL 提供了主机到主机的安全机制是很棒的，特别是当你想要以一种安全的方式连接位于 Kubernetes 内部的 Pod 到集群外部的时候。代理缓存(特别是用的权限的) 显著地让通讯性能变得非常棒。

所以和 Linkerd2 一样这是另一个值得期待的 Service Mesh 实现。Consul 连接在前几周刚被发布，而且现在还没有太多教我们如何去做的在线指导手册。如果以经典对于 Hashicorp 的工具链做过很深的调研，那么我会想试试它，没准能让我学会如何将默认的 proxy 换成 Envoy。

英文文献翻译

Istio

关键词

组件繁多，复杂度高，学习门槛高

功能完备，文档齐全

完美兼容 K8s

有 Google , IBM 和 Lyft 作为后盾

完全免费(目前)

译者注。

Istio 功能丰富且相当稳定。截止这篇博客写完，Istio 拥有着 11.5k 的 Github 收藏量，以及 244 位来自 Lyft, Google 和 IBM 的贡献者。Istio 是许多好想法的先锋，且目前一直被其他的 Service Mesh 模仿。

其中的一个鹤立鸡群的特性就是“自动 Sidecar 注入(automatic sidecar injection)”，在 Helm 的协同下，它工作的出奇地好。

当然在模块化，插件性能和超高的复杂度方面 Istio 还有很多劣势。我可以换掉 Istio 几乎任何组件并集成其他组件进系统。但这都会当来相当陡峭的学习曲线而且还有一大堆坑等你去踩。

然而令人惊奇的是，如果你只用默认配置，你将能快速搭建并使用 Istio。使用 minikube ,helm 和 Istio 在你的笔记本上配置一个测试实例不会花费超过 5 分钟。网上同样有上千篇文章来教你如何配置其他集成组件。在这点上，其他 Service Mesh 实现比起来就显得格外荒凉。

赢家： Istio

最后，我得说如果你像很多人一样准备用 Kubernetes 来摸索 Service Mesh，Istio 可能是目前最好的选择。他虽然很复杂，但与你已经搭建并管理的 Kubernetes 比起来也就不怎么复杂了。它在这四者中有最多的功能特性而且即将推出 version 1 并且在几个月前就能用于生产环境了。同样，它还有 Google , IBM 和 Lyft 以及大量产出酷炫博客和可集成组件的社区作为后盾。

结束语

我写的不一定对。也许这篇比较还不太成熟。对于软件而言，有个能相互比较的地方是很棒的。希望有朝一日我再看这个列表时会选出一个新的赢家。

如果我写的任何技术细节不正确，请在下面给我留言。

想要持续获得 Service Mesh 的最新资讯请订阅我们的新邮件并访问我们的 Service Mesh 分类。