What is a Service Mesh?

Adrian Otto, *Google*   Max Saltonstall, *Google*

Abstract

Adopting a microservices architecture to modernize your applications has numerous benefits, but it also presents a number of new challenges. Container orchestration engines such as Kubernetes help to address many of these concerns, but it's not the tool to solve all concerns. For example, how do you find out why your application is suddenly running slow? How will one service find a remote service in another cluster? How can I limit which services should be allowed to communicate with each other? How to efficiently distribute traffic throughout my cluster as it grows and shrinks? How can I release a new version of software to only a single geography? How can I stop trusting my network completely? All of these challenges can be addressed using a service mesh. This paper explains what a service mesh is, how it's different from the tools you've already heard about, and how you can solve each of these challenges using one. Istio, an open source project complimentary to Kubernetes, allows you to set up a service mesh of your own to start learning more about how it works. We also explain how Google has used these solutions to address challenges in our own systems using a range of best practices for managing distributed systems both for software developers, and system operators.

1.1 Definition

A service mesh is a distributed system that controls the configuration and behaviour of all your microservices.

This works by centralizing the control for data going in and out of your services, and proxying all the traffic between them. The service mesh brings benefits by acting as a single source of data collection about inter-service communication and traffic in and out of the microservices ecosystem.

2.1 Benefits

2.2 Scale and Efficiency

Using a service mesh allows for increased scale for your microservices, without adding extra burdens to the development of the individual services themselves. To adapt to large swings in demand or usage, your service mesh can handle load balancing for your microservices, distributing requests and avoiding overloaded processes. Using traditional load balancing techniques that concentrate traffic at physical or logical reverse proxies limits overall system scalability at the points of your network where traffic is concentrated. Imagine having a system that offers the same capability, but accomplishes it in a way that every server in the infrastructure can evenly balance a portion of the traffic. This way your network capacity can scale proportionally to the number of servers you use without having to reason about how to elegantly distribute that traffic.

2.3 Redundancy, Reliability, and Advanced Deployment

A service mesh can also enforce redundancy standards on the services it supports, to maintain reliable, accessible services to the rest of the ecosystem, even while it's being updated. One of the benefits of microservices is the ability to update your systems more frequently with less risk of regression. This is because changes are made to smaller, simpler units of code. Enjoying this benefit means that you can update your systems with lots of changes much more frequently than if you used a monolithic software design. If your software is constantly being upgraded, you might intentionally overprovision each of your microservices so that there is enough redundancy to run the service even when it's conducting an upgrade to a new version. Imagine using a service mesh to manage the allocation of resources in accordance with your predefined target for available capacity, and relying on your container management platform, such as Kubernetes to maintain the right capacity and redundancy level during those times that you are not rolling out a software update.

Now let's explore another deployment related concern.

When you want to update your software, you might want to test your new release using a canary process where the new version is gradually introduced. This allows you to gain confidence in a new version before deciding to release it to your entire user base. Service mesh tools can manage new release versions of a service by testing them out on a small percentage of traffic first, gradually increasing the exposure as the rollout goes smoothly. This creates a smooth flow from one version to another, with automated rollback if a new version creates new errors or issues. This capability can also help you canary test with specific targets, clients or regions.

Another problem introduced by microservices architectures is that some services may be more sensitive to connection concurrency constraints whereas others are less sensitive. For example, you may have a web service that gracefully handles thousands of concurrent HTTP connections and a data persistence service that can only gracefully handle dozens of concurrent connections. Your service mesh can control this for you so only an optimal number of concurrent connections are routed to your data persistence service, while more are allowed to your web application service. This is easier than configuring an additional microservice of connection concentrators to wire your services together. Your service discovery logic can be simplified by eliminating the connection concentrators.

2.4 Security: Authentication, Authorization, and Encryption

Dividing a monolithic app into microservices creates a whole new set of security concerns, and relies on each service owner to manage authentication, authorization and encryption. To simplify this, a service mesh can globally apply policy that you define centrally. This concern is beyond the scope of what systems like Kubernetes were designed for, but it’s a perfect fit for a service mesh. Using a service mesh to apply a security policy across your entire network brings finer-grained control than perimeter firewall-style security protection. In dynamic environments, new instances of your microservices may be created several times per minute. Configuring your firewalls this rapidly may be impractical or impossible, if they have a programmable control mechanism at all. This challenge typically results in the use of perimeter only policies to control access to microservices that are not frequently updated. If all of your microservices are able to freely communicate with all of your other microservices, then if only one of them were compromised, it may lead to a breach to your entire system. If you have a way to dynamically control which services should be allowed to communicate with which others on which ports, protocols, and request types, and each microservice required valid authorization, the attack surface of your system would be dramatically reduced. This is another way to use a service mesh.

2.5 Insight

Just about every single process program has some form of runtime debugger tool, no matter what language it was written in. If you start peeling off parts of your application into microservices, and running a multi-process distributed system connected by messaging, it becomes more challenging to debug problems that involve multiple services. Using a service mesh allows you to systematically track and correlate communications between your various services in a way that they can be tracked and charted. Istio provides a built-in feature for visualizing request flows so you can zero in on performance bottlenecks, and quickly narrow down where to look when something is not performing well. Here is an example of the sort of visualization you get for free when you use Istio as your service mesh.

3.1 Lessons learned

For many years, Google has employed distributed systems designs in order to build solutions that work at very large scale. Network functions such as load balancing are widely distributed throughout our infrastructure so network flows and computational processing of our network packets are not concentrated in hotspots, but instead shared widely. The systems that control this functionality are logically centralized, but managed as a distributed system. For example, configuration and policy can be managed centrally, and globally enforced throughout the system through a distributed control plane. This allows for low latency handling of network connections throughout our infrastructure. You can do this too. We have released Istio as an open source software project, and engaged a growing community of contributors to help address this openly so that you can enjoy the same benefits that Google has evolved through years of learning and refinement.

Istio is designed to work with a variety of system types, including both virtualization environments, and container environments. Early versions of Istio demonstrate integration with Kubernetes using a sidecar pattern. Your traffic is directed to a lightweight transparent reverse proxy that runs in each of your pods, and applies your desired policies and features. An API service allows you to configure the proxies centrally.

You might have an initial reluctance to introducing a transparent proxy in front of your services. Proxying your traffic in all your pods may come at a small performance cost, but it's totally worth it, because it enables a much more powerful management paradigm for your systems. Your gain in overall system efficiency and the simplification of your security configuration will probably offset the proxying overhead.

3.2 Next steps

Ready to begin?

Find one of your applications using Kubernetes, or one that you can containerize without too much trouble, get your developers to go through the Istio codelab, and then start modifying your pods to add service mesh capabilities. You can download the open source code freely (and even contribute if you have suggestions or ideas).