Service Mesh

And The Natural Evolution of Microservices

Service Mesh

And The Natural Evolution of Microservices

Marco Palladino

Service mesh is one of the hottest topics in technology right now and with good reason. Service mesh represents the next innovative leap in transitioning from centralized architectures to de-centralized architectures. Despite this, what we may perceive to be a new technology with service mesh is actually a repackaging of existing technologies in a novel way. With service mesh, we are taking the functionality of a traditional API gateway and deploying it in a new pattern.

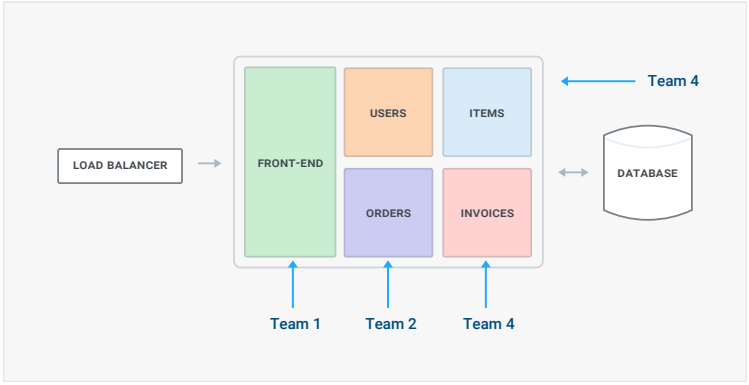
In following the evolution of microservices, containers, and serverless, we are all likely familiar with the shift away from large monoliths to smaller, more agile services. Despite this, for many of us, it can be a challenge to understand exactly what service mesh is and what makes it so exciting. Understanding service mesh in the proper context requires an understanding of the evolution from monoliths to microservices.

Content

From Monolith to Modern

In the beginning, we built monoliths – massive blocks of code that housed all the components of an application. We strove to make our monoliths perfect. However, much like the evolution of automobiles, the more complex the system became, the more challenging it was to maintain a self-contained solution. The problem was that as the codebase and the application grew in terms of functionality or complexity, the more challenging it became to iterate on it. Each component of a monolith had to be tuned to work perfectly with the other components, or else the entire application would fail.

In practice, this meant multiple teams working on a single codebase, all of whom needed to be in perfect concert with each other – all the time. This led to numerous challenges in trying to rapidly deploy software. If a team wanted to make a change to a application component, it had to redeploy the entire monolith. Additionally, each new change meant adding a new point of failure.



To combat this, many of us began to decouple our monoliths and transitioned to a more API-centric enterprise, creating smaller and smaller services for public and private consumption. The rise of containers accelerated this trend by allowing us to abstract our services a level away from the underlying virtual machines, thereby enabling us to make services even smaller. The net result of this was that we could decouple our monoliths into smaller components that could be executed independently.

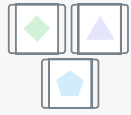
With the growing popularity of tools like Docker and Kubernetes, we've seen accelerated uptake of containers.

These tools have made it easy to decouple services and have helped us to stop thinking in terms of monoliths. With them, we can separate out the execution of our services and keep their isolation consistent. In essence, Docker and Kubernetes provide the tooling needed to enable mainstream adoption. While some companies like Netflix and Amazon transitioned without these tools, their process of decoupling monoliths was more challenging.

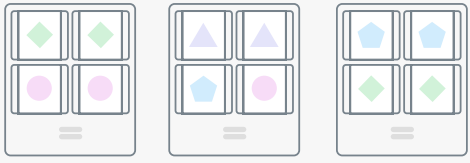
A **monolithic** application puts all its functionality into a single process...

... and scales by replicating the monolith on multiple servers.

A **microservice** architecture puts each element of functionality into a separate service...

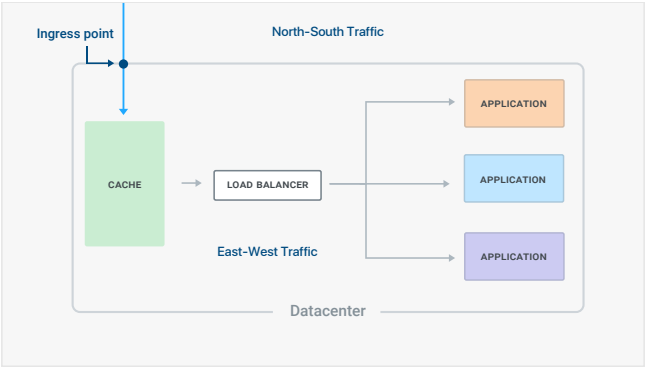


... and scales by distributing these services across servers, replicating as needed.

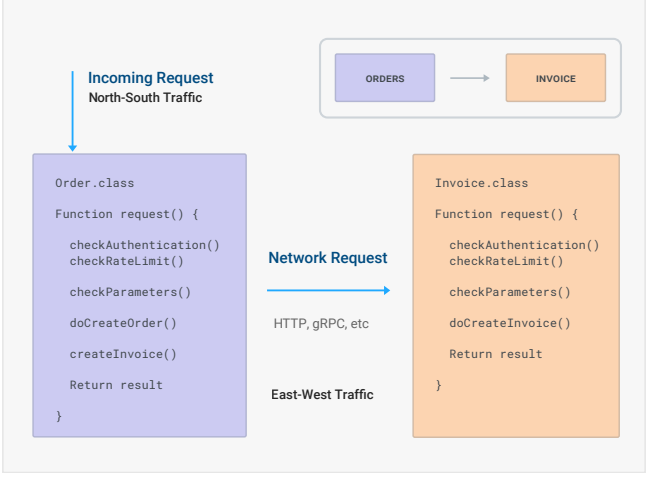


From North-South to East-West

In our old monolithic architectures, we dealt almost exclusively with north-south traffic, but with microservices, we must increasingly deal with traffic inside our data center. With monoliths, different components communicated with each other using function calls within the application. Edge gateways abstracted away common traffic orchestration functions at the edge, such as authentication, logging and rate limiting, but communications conducted within the confines of the monolith did not require any of those activities.

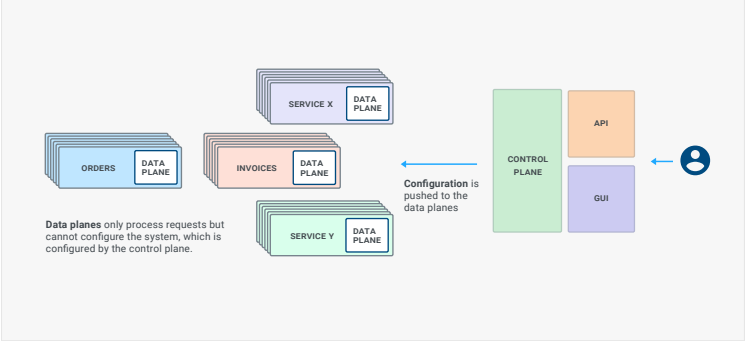


While east-west traffic presents a greater challenge due to replacing our function calls with communication over the network. It allows us to use whatever transport method we want, as we've replaced function invocations with APIs over a network. This means that the different services within our architecture don't have to know about each other – if our API is consumable, then we have flexibility with everything else. This can provide big advantages. For instance, if we're a big organization, and if we acquire another team, we don't have to worry about the coding language they use or how they do things. However, the network creates more problems than function calls since the network carries latency and is unreliable by nature.

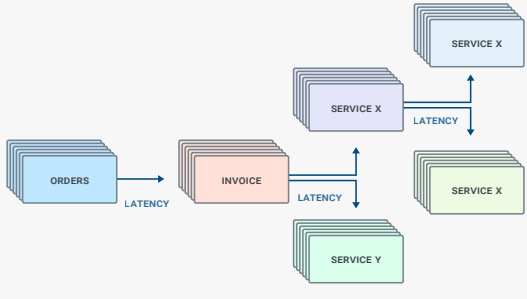


The Challenge with Traditional Gateways

With the increased east-west traffic that comes from microservices, we now need to the ability to properly orchestrate it – which is the same issue we faced with our monolith at the edge. We need to effectively route traffic, but now all of the common features like routing, authentication, and logging are daisy-chained. This complexity results in traditional gateways not handling east-west well, and necessitates the use of a smaller, more flexible gateway.



With microservices, we also have multiple instances of each service. This leads to greater complexity we must deal with in regard to service discovery. Our services need to know where to send requests, whether the network is reliable, and how to deal with too much latency, error handling, and other issues. We need to be certain that we can effectively deal with these issues as the challenges will get compounded as we increase our number of services.

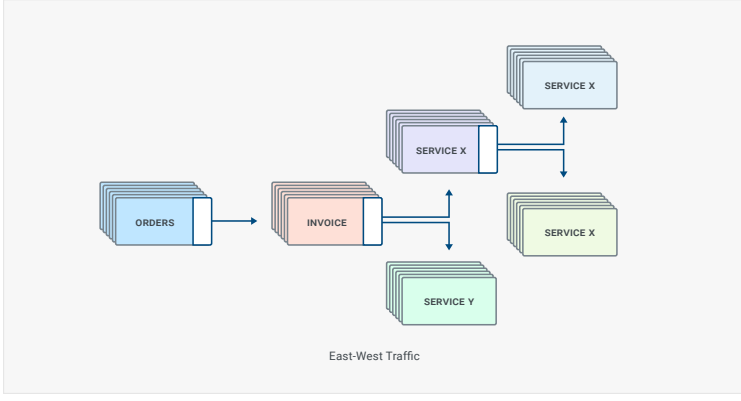


East-West Traffic

Proxies, Gateways, and the Foundations of Service Mesh

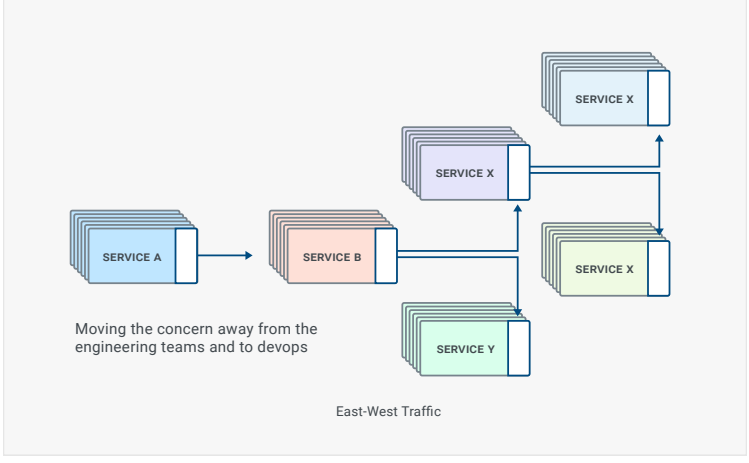
In making our services more and more granular, we increased our need for effective east-west communication. This led to a search by practitioners for solutions that could address the issues that arise. Originally, many of us thought we could use the same client library for each microservice, but this was quickly abandoned. The primary reason this solution failed was that it largely eroded the inherent value of microservices. With one client library, we would need to redeploy our services every time we updated them, which would reduce our speed of deployment and increase failure risk. Worse yet, we would also need to limit each team's ability to use the implementation of its choice as we would be running off of a single client library. We could conceivably build our client library in every language that we wanted to use, but this would quickly become impractical. Then there would be the challenges of telemetry. We would have a lot of services, but it would be difficult to accurately diagnose and address latency issues.

Since high latency would cause our architecture to fail, and for the other reasons listed above, the single client library solution was deemed infeasible.



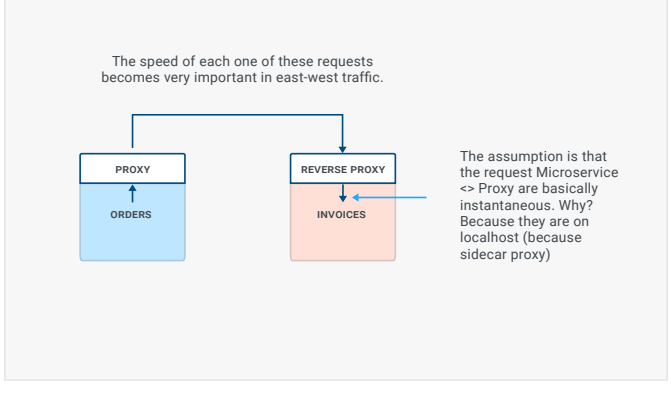
To solve these issues, we can simply add a proxy. This solution moves the concern away from engineering and to DevOps, freeing our engineering teams to focus on improving the service. The proxy process runs alongside each one of our services and abstracts the key traffic management functionality. Every time we make a request, the request goes through our proxy, and the proxy handles any issues that may arise. If our network is unreliable and latency is too high, the proxy will make sure that we can retry or fix a request that is not working.

This allows us to abstract away the traffic routing and management functionalities from the codebase and from the development team. Our service development teams no longer need to be concerned about the network because the proxy handles those concerns. This does require, however, that a proxy is injected alongside our service every time we deploy.



The Makings of a Mesh – the Sidecar

To reduce the complexity of injecting our proxy alongside each deployed service, the sidecar pattern was created. The sidecar takes advantage of the abstraction layer created by a container orchestration tool like Kubernetes. This abstraction layer exists between our containers and the virtual machines we run our containers on, and it makes our virtual machines appear as a single fabric. With Kubernetes, we can have a container be a sidecar proxy for another container, allowing the sidecar to handle network communications independently of the container running our service. This forms the foundation of our service mesh.

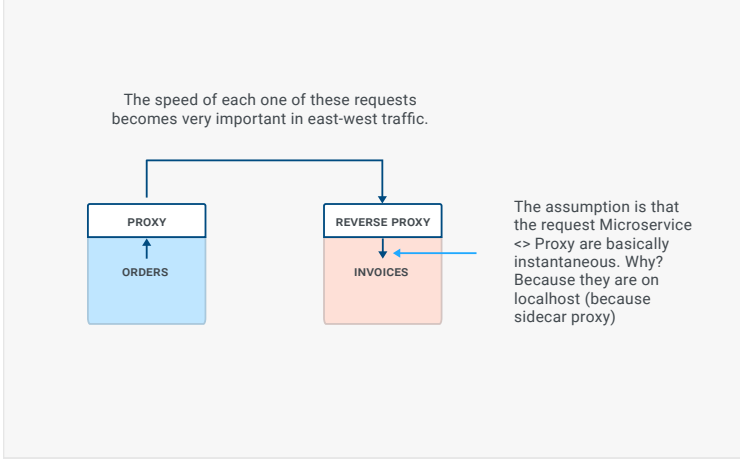


As our mesh grows, it's critical that our sidecar proxy can effectively scale with our growing number of microservices. In a containerized world, we are continually reducing the size of our services, which requires our sidecar proxy to be extremely lightweight and fast. Part of this stems from our forcing Kubernetes to push both of our containers on the local host to minimize the potential for communication problems between the service container and sidecar proxy. If our proxy is too large, we'll overburden the underlying virtual machine, and if it's too slow, we'll run the risk of introducing latency problems. Understanding that transitioning to a mesh will require fine-tuning to optimize performance, it is imperative that we be able to accurately diagnose where potential issues may arise.

Understanding our Mesh

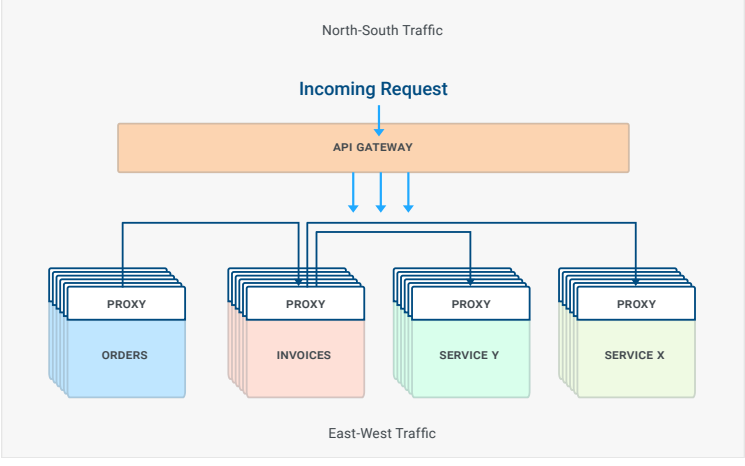
With an exponentially increasing number of east-west API calls being made between services, our ability to understand latency performance becomes critical. Fortunately, the architecture of our service mesh lends itself perfectly to tracking performance. Each of our sidecar proxies functions as both proxy and reverse proxy – a proxy for outbound communications and a reverse proxy for in-

bound. Since our sidecar functions in both capacities, it knows when communications are sent and when they are received, providing us telemetry out of the box. This allows us to actively sift traffic as it leaves one of our services and goes into another service.



Controlling our Mesh

As with any innovation in architecture, service mesh brings new challenges along with its advantages. The first question we must address is how we're going to configure all of our proxies. For instance, if we want to change the time-out for a communication between our orders and invoices services from 10 seconds to five seconds, how do we accomplish this without redeploying every instance of our sidecar proxy? The answer lies in how we separate the functions of our data plane and control plane.



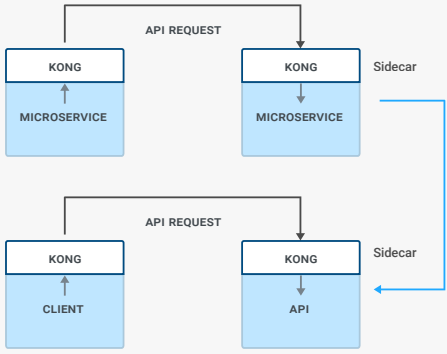
In their simplest forms, the data plane and control plane can be understood as follows: the data plane is whatever runs on the execution path of service to service requests, and the control plane pushes the configuration to our data plane. In the case of our service mesh, we would make a change to a given configuration in our control plane, and that change would be pushed out to each one of our sidecar proxies. As our control plane can identify the proxy instances associated with each one of our services, we can quickly make large-scale changes to the each of our proxy configurations without interrupting our services.

Conclusion

Service mesh offers us the same subset of traditional use cases for north-south traffic deployed in a way that better handles the increased east-west traffic generated by a microservices architecture. Our service mesh proxy can collect telemetry, handle routing and error handling, and limit access to our services in the same way that traditional gateways have handled north-south traffic for years. In essence, we're using the same technology and features, merely deployed in a different implementation.

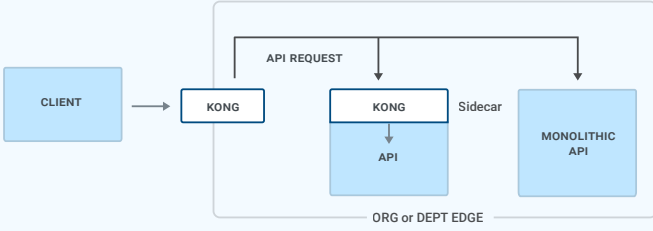
In Service Mesh the client is another microservice inside the organization.

Service Mesh



In API GW / Ingress the client is a third-party entity.

API Gateway/Ingress



For the typical enterprise, there's not likely going to be a single dominant implementation paradigm. With the risk and disruption inherent in transitioning to a new architecture pattern, adoption of service mesh, like the adoption of public cloud, is not likely to be wholesale nor instant. As we get more and more distributed, however, service mesh begins to fit a greater number of use cases. Despite this, the most likely scenario is that as we transition to a distributed architecture, we will still rely on applications built with legacy architectures to power our organization. This makes the adoption of an API management platform that works effectively with legacy and modern architectures a critical step in any digital transformation journey.

About the Author

Marco Palladino is an inventor, software developer, and internet entrepreneur based in San Francisco, California. He is the co-founder and CTO of Kong, the most widely adopted OSS API and Microservice gateway. Besides being a core maintainer, Marco is currently responsible for the design and delivery of the Kong products, while also providing the technical thought leadership around APIs and Microservices within Kong and the external community. Marco was also the co-founder of Mashape, which started in 2010 and is today the largest API marketplace in the world.

Konghq.com

Kong Inc.
contact@konghq.com

251 Post St, 2nd Floor
San Francisco, CA
94108 USA