ISOVALENT
now part of cisco

How eBPF will solve Service Mesh – Goodbye Sidecars

Thomas Graf | Published: Dec 08, 2021 | Updated: Jul 11, 2023 | Isovalent

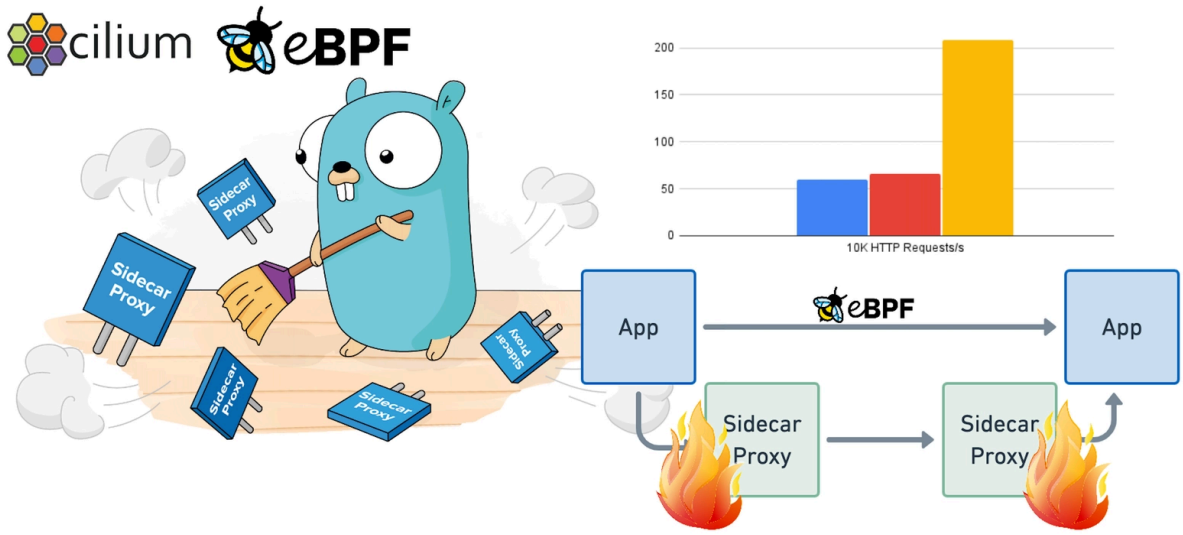


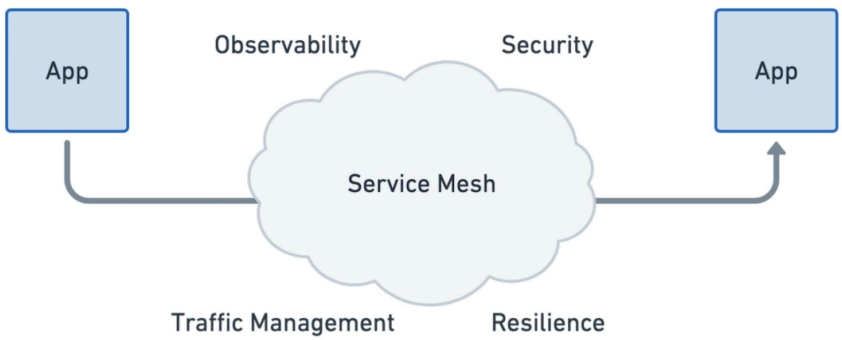
Table of contents

Further Reading

Service mesh is a concept describing the requirements of modern cloud native applications with regards to communication, visibility, and security. Current implementations of this concept involve running sidecar proxies in each workload or pod. This is a pretty inefficient way of solving these requirements. In this post, we will look at an alternative to the sidecar model that provides a transparent service mesh with high efficiency at low complexity, with the help of eBPF.

What is Service Mesh?

With the introduction of distributed applications, additional visibility, connectivity, and security requirements have surfaced. Application components communicate over untrusted networks across cloud and premises boundaries, load-balancing is required to understand application protocols, resiliency is becoming crucial, and security must evolve to a model where sender and receiver can authenticate each other's identity. In the early days of distributed applications, these requirements were resolved by directly embedding the required logic into the applications. A service mesh extracts these features out of the application and offers them as part of the infrastructure for all applications to use and thus no longer requires to change each application.



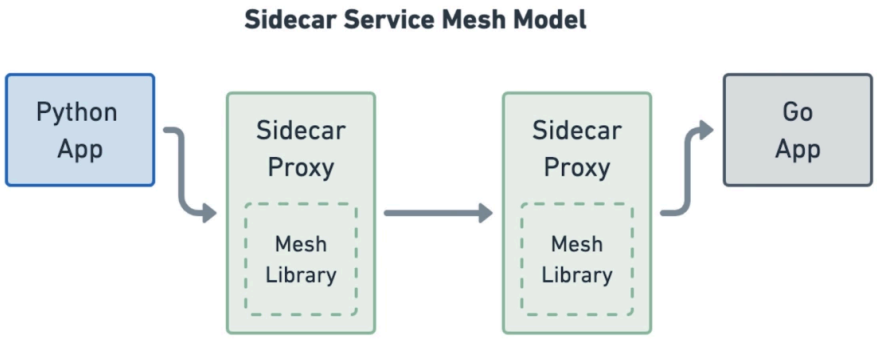
Looking at the feature set of a service mesh today, it can be summarized as follows:

- **Resilient Connectivity:** Service to service communication must be possible across boundaries such as clouds, clusters, and premises. Communication must be resilient and fault tolerant.
- **L7 Traffic Management:** Load balancing, rate limiting, and resiliency must be L7-aware (HTTP, REST, gRPC, WebSocket, ...).
- **Identity-based Security:** Relying on network identifiers to achieve security is no longer sufficient, both the sending and receiving services must be able to authenticate each other based on identities instead of a network identifier.
- **Observability & Tracing:** Observability in the form of tracing and metrics is critical to understanding, monitoring, and troubleshooting application stability, performance, and availability.
- **Transparency:** The functionality must be available to applications in a transparent manner, i.e. without requiring to change application code.

In earlier days, service mesh functionality was typically implemented as libraries, requiring each application in the mesh to link to a library written in the application's language framework. Similar things have happened in the early days of the Internet: back in the day, applications used to ship their own TCP/IP stack! As we'll discuss in this article, service mesh is evolving to become a kernel responsibility, much as the networking stack did.

Today, service meshes are commonly implemented using an architecture called the sidecar model. This architecture encapsulates the code implementing the functionality described above into a layer 4 proxy, and then relies on traffic from and to services to be redirected into this so-

called sidecar proxy. It is called a sidecar because there is a proxy attached to each application, much like a sidecar attaches to a motorbike.



Service mesh is is embedded in a proxy running outside of the application.

The advantage of this architecture is that services are no longer required to implement service mesh functionality themselves. This is beneficial if many services are deployed written in different languages or if you are running 3rd-party applications that are immutable.

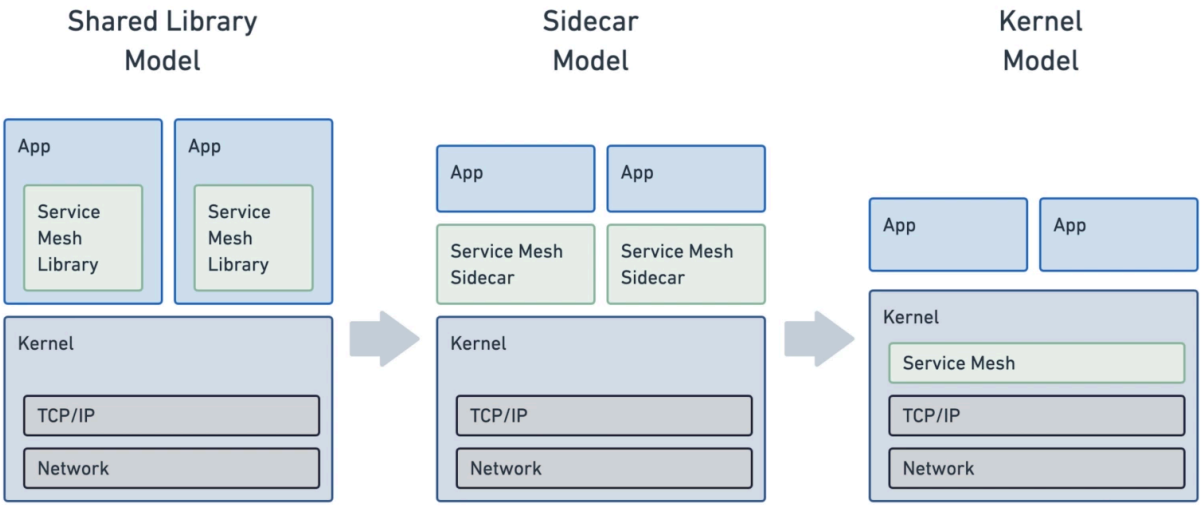
The downsides of this model are the vast number of proxies, many additional network connections, and a complex redirection logic to feed network traffic into the proxies. On top of that, there are also limitations in what type of network traffic can be redirected to a layer 4 proxy. Proxies are limited in what network protocols they can support.

A history of connectivity moving into the Kernel

Providing secure and reliable connectivity between applications has been the responsibility of the operating system for decades. Some of you may remember TCP Wrappers and tcpd from earlier Unix and Linux days. It could be considered the original sidecar. tcpd allowed users to transparently add logging, access control, host name verification, and spoof protection to applications without modifying them. It used libwrap, and, in an interesting parallel to the service mesh story, this same library was what applications previously linked with to provide these capabilities. What tcpd brought to the table was the ability to transparently add the functionality to existing applications without modifying them. Eventually,

all of this functionality found its way into Linux itself and became available to all applications in a more efficient and powerful manner. Today, this has evolved to what we know as iptables.

However, iptables is clearly not suitable to solve the connectivity, security, and observability requirements of modern applications as it operates exclusively on the network level and lacks any understanding of the application protocol layer. Naturally, the path of least resistance was to go back to the library model, then the sidecar model. Now we are at the point where it makes sense to support this natively in the operating system for optimal transparency, efficiency, and security.

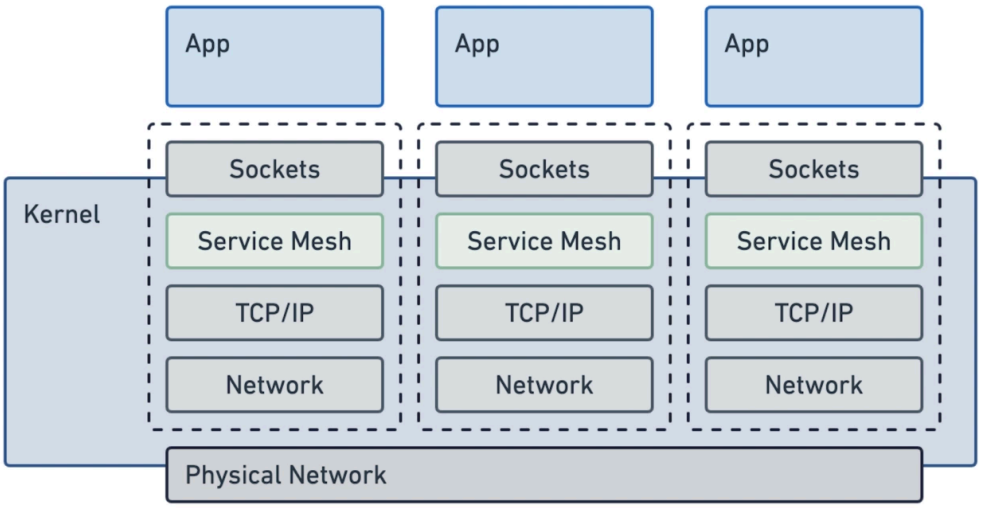


What used to be connection logging back in the days of tcpd is now tracing. Access control on IP level has evolved into authorization on the application protocol level, for example with JWT. Host name verification has been replaced with much stronger authentication such as mutual TLS. Network load-balancing has been extended with L7 traffic management capabilities. HTTP retries are the new TCP retransmissions. What used to be solved with blackhole routes is called circuit breaking today. None are fundamentally new but the required context and control have evolved.

Extending the Kernel Namespace Concept

The Linux kernel already has a concept to share common functionality and makes it available to many applications running on the system. This concept is called namespaces and it forms the foundation of container technology as we know it today. Namespaces (the kernel kind, not the Kubernetes version) exist for a variety of abstractions including file systems, user management, mounted devices, processes, network, and several more. This is what allows individual containers to be presented with a different view of the file system, a different set of users, and what allows multiple containers to bind to the same network port on a single host. This concept is expanded with the help of cgroups to apply resource management and prioritization for resources like CPU, memory, and the network. From the perspective of cloud native application developers, cgroups and resources are tightly integrated into the concept we know as "containers".

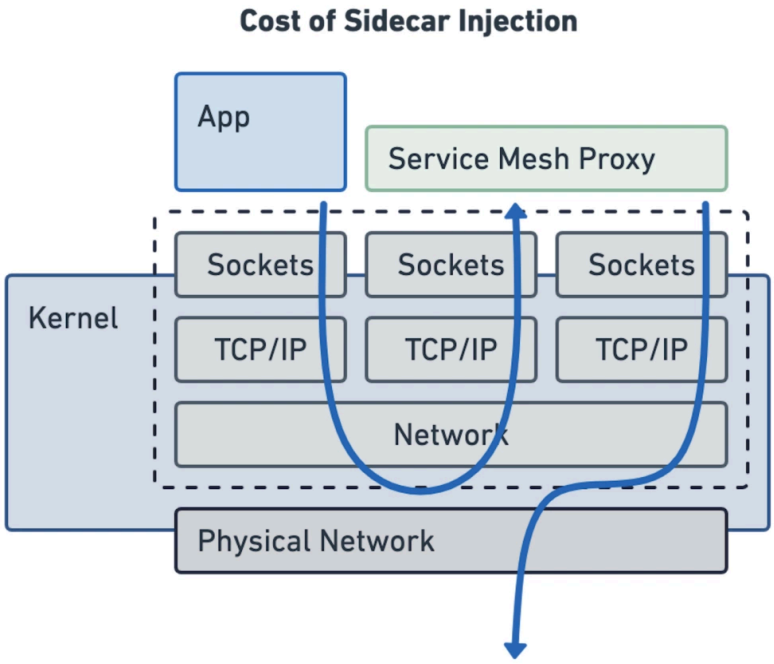
It's only logical that if we consider service mesh as a responsibility of the operating system, then it must conform to and integrate with the concept of namespaces and cgroups. This will look something like this:



Unsurprisingly, this looks very natural and is probably what most users expect from a simplicity perspective. Applications remain unchanged, they continue to use sockets to communicate as they always have. The desirable service mesh functionality is provided transparently as part of Linux. It's just there, like TCP is there today.

The Cost of Sidecar Injection

If we look closer into the sidecar model, we notice that it is actually trying to emulate this model. The application continues to use sockets and everything gets stuffed into a network namespace of the Linux kernel. However, it is more complex than it looks, many additional steps are required to transparently inject the sidecar proxy:



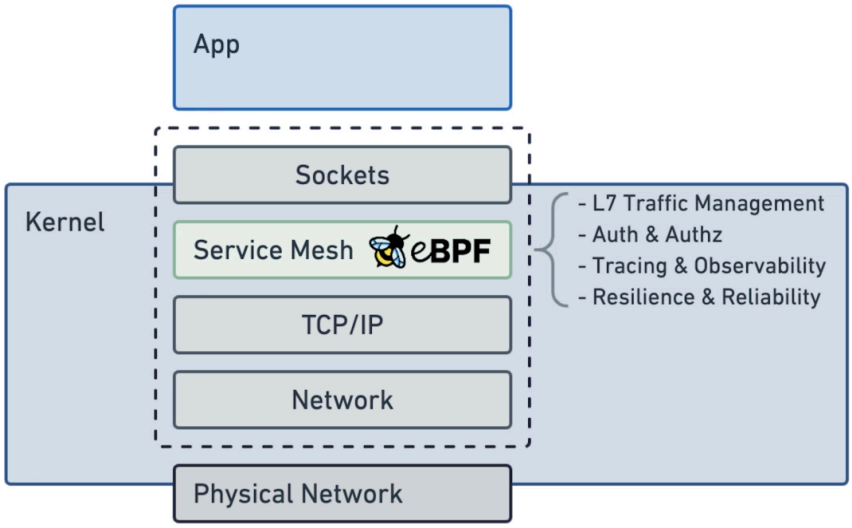
This additional complexity comes at a significant cost in terms of latency and additional resource consumption. Early benchmarks indicate that this can impact latency up to 3-4x and a significant amount of additional memory is required for all the proxies. We'll look into both later on in this post as we compare it to an eBPF-based model.

Unlocking the Kernel Service Mesh with eBPF

Why have we not created a service mesh in the kernel before? Some people have been semi-jokingly stating that kube-proxy is the original service mesh (See We've Made Quite A Mesh – Tim Hockin, Google). There is some truth to that. Kube-proxy is a good example of how close the Linux kernel can get to implementing a service mesh while relying on traditional network-based functionality implemented with iptables. However, it is not enough, the L7 context is missing. Kube-proxy operates

exclusively on the network packet level. L7 traffic management, tracing, authentication, and additional reliability guarantees are required for modern applications. Kube-proxy cannot provide this at the network level.

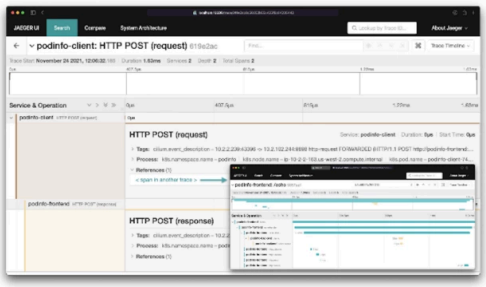
eBPF changes this equation. It allows to dynamically extend the functionality of the Linux kernel. We have been using eBPF for Cilium to build a highly efficient network, security, and observability datapath that embeds itself directly into the Linux kernel. Applying this same concept, we can solve service mesh requirements at the kernel level as well. In fact, Cilium already implements a variety of the required concepts such as identity-based security, L3-L7 observability & authorization, encryption, and load-balancing. The missing parts are now coming to Cilium. You will find details on how to join the Cilium service mesh beta program driven by the Cilium community at the end of this blog.



Some of you may wonder why the Linux kernel community is not addressing these requirements directly. eBPF has a massive advantage. eBPF code can be inserted at runtime into an existing Linux kernel similar to a Linux kernel module, but unlike a kernel module, it can be done in a secure and portable manner. This allows for eBPF implementation to continue to evolving with the service mesh community. New kernel versions would take years to make it into the hands of users. eBPF is the critical technology that allows the Linux kernel to keep up with the rapidly evolving cloud native technology stack.

eBPF-based L7 Tracing & Metrics without Sidecars

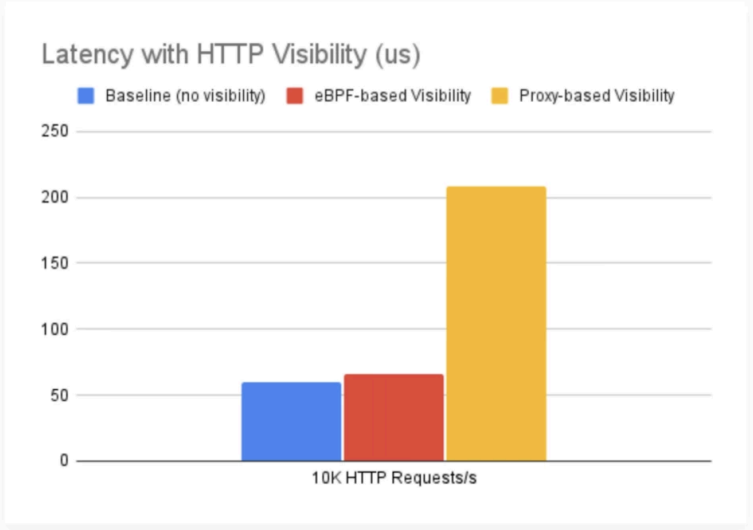
Let's look at L7 tracing and metrics observability as a concrete example of how an eBPF-based service mesh has a massive impact on preserving low latency and keeping the cost of observability low. Applications teams rely on application visibility and monitoring as a fundamental requirements these, this includes capabilities such as request tracing, HTTP response rates, and service latency information. However, this observability should come at no significant cost (latency, complexity, resources, ...).



eBPF-based OpenTelemetry with Jaeger

eBPF-based HTTP Golden Signals with Prometheus & Grafana

In the benchmark below, we see early measurements of how implementing HTTP visibility with eBPF or a sidecar approach affects latency. The setup is running a steady 10K HTTP requests per second over a fixed number of connections between two pods running on two different nodes and measures the mean latency for the requests.



We are deliberately not mentioning the specific proxy used in these measurements because it does not matter. The results are almost identical for all proxies we have tested. To be clear, this is not about whether Envoy, Linkerd, Nginx, or another proxy is faster. There are differences in the mentioned proxies but they are insignificant in comparison to the cost of injecting the proxy in the first place. Almost none of the overhead is coming from the logic in the proxy itself. The overhead is added by injecting the proxy, redirecting network traffic to it, terminating connections, and initiating new connections.

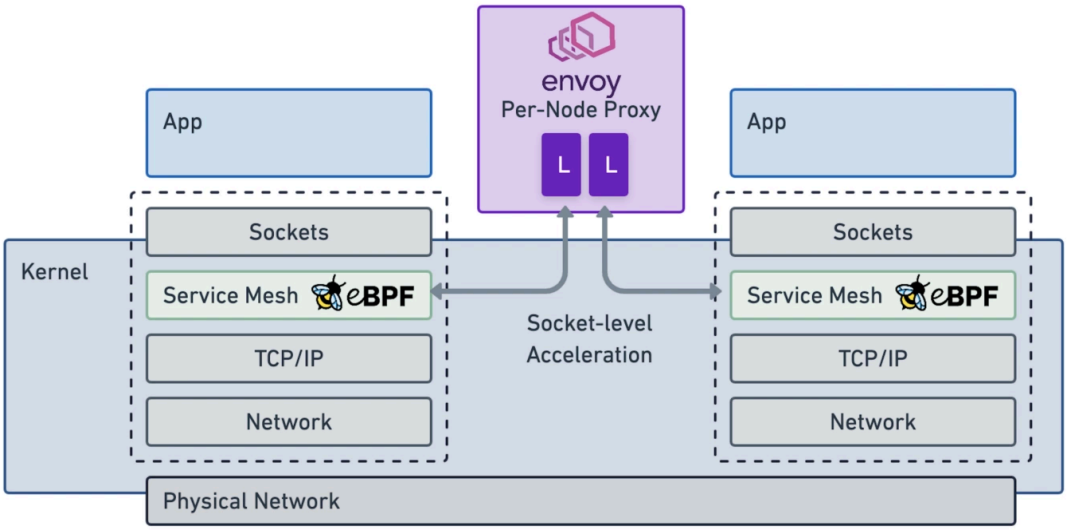
These early measurements make it obvious that an eBPF-based in-kernel approach is extremely promising and may deliver on the desire to implement fully transparent service mesh functionality at no significant overhead.

eBPF Accelerated Per-Node Proxy

More and more use-cases can be covered with this eBPF-only approach and thus completely void the L4 proxy. For some use cases, a proxy is still needed. For example when connections need to be spliced, when TLS termination is being performed, or for some forms of HTTP authorization.

Our eBPF service mesh efforts will continue to focus on areas where the most can be gained from a performance perspective. You may not mind terminating a connection with a proxy once as traffic flows into the cluster if you have to perform TLS termination anyway. However, you will care much more about the impact of injecting two proxies into the path of every single connection just to extract HTTP metrics and tracing data.

When a use case cannot be implemented with an eBPF-only approach, the mesh can fall back onto a per-node proxy model that directly integrates the proxy with the socket layer of the kernel.



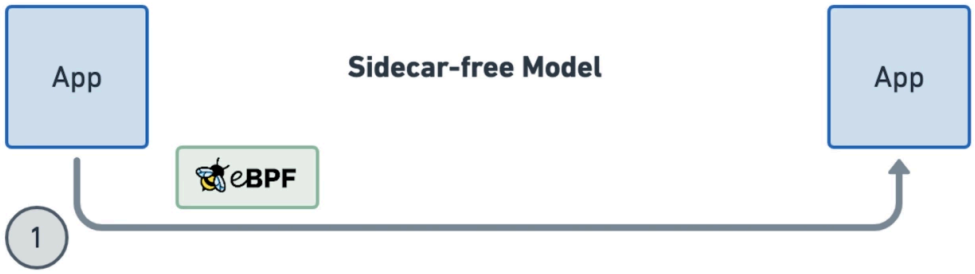
Instead of relying on network-level redirection, eBPF can inject the proxy directly at the socket level, keeping paths short. In the case of Cilium, Envoy is being used although from an architecture perspective, any proxy could be integrated into this model. Conceptually, this allows to extend the concept of a Linux kernel network namespace directly into the concept of an Envoy listener configuration and turn Envoy into a multi-tenant proxy.

Sidecar vs per-Node Proxy

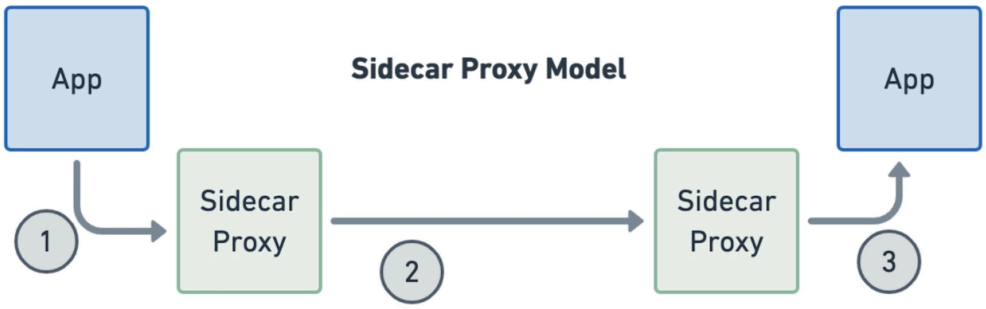
Even when a proxy is needed, the cost of proxy will vary depending on the architecture deployed. Let's look at a per-node proxy model compared to a sidecar model and see how they compare.

Proxies per Connection

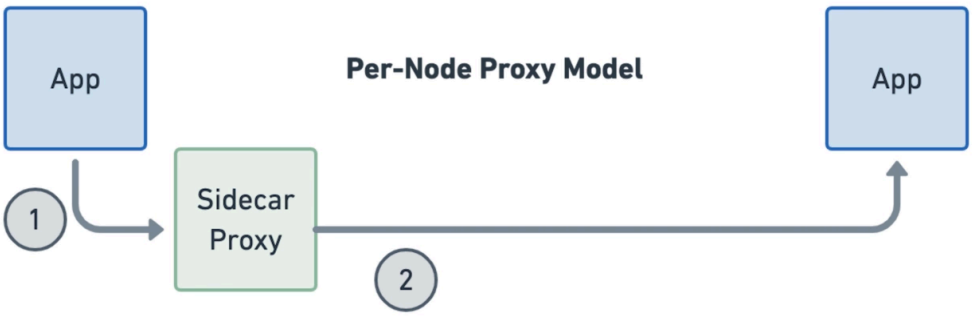
The number of network connections required will vary depending on whether proxies are in the picture. The simplest scenario is the sidecar-free model which implies no changes to the number of network connections. A single connection will serve the requests and eBPF will provide service mesh capabilities such as tracing or load-balancing on the existing connection.



Providing the same functionality with a sidecar model requires injecting a proxy twice into the connection which results in three connections that need to be maintained. This results in increased overhead and multiplication of required memory for all the additional socket buffers which manifest in higher service to service latency. This is the sidecar overhead we have seen earlier in the sidecarless L7 visibility section.



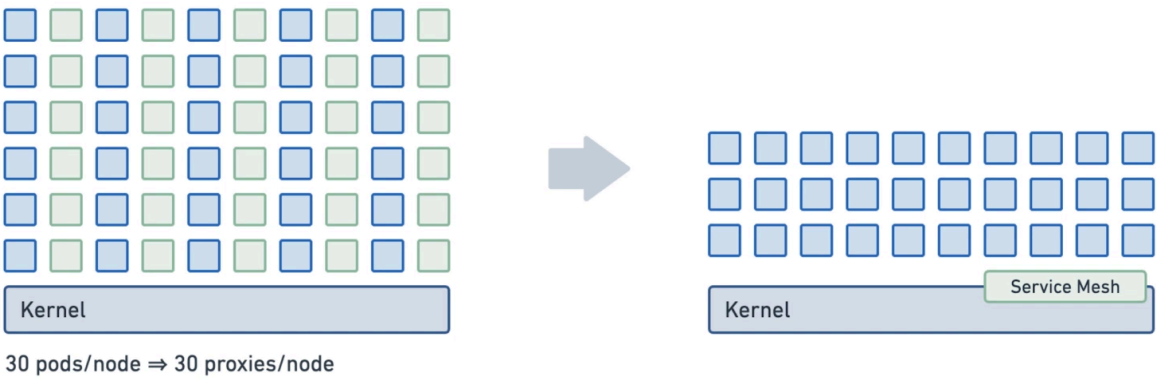
Switching to a per-node proxy model allows us to get rid of one of the proxies because we no longer rely on running a sidecar inside of each workload. Still less ideal than no additional required connections, but better than always requiring two additional connections.



Total number of proxies required

Running a sidecar in each workload can result in a large number of proxies. Even if each individual proxy instance is pretty optimized in its

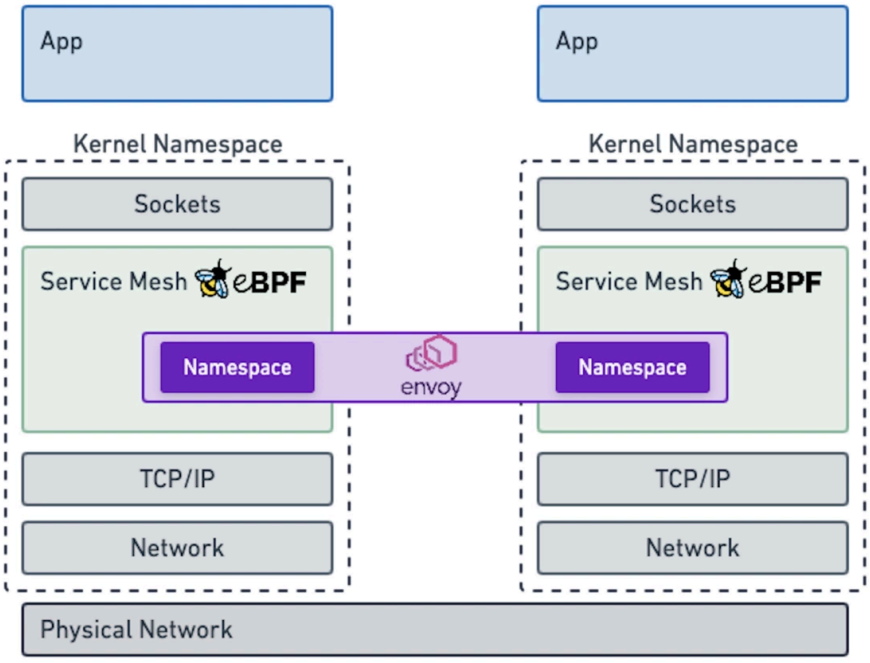
memory footprint, the sheer quantity of instances will result in a heavy total impact. Moreover, each proxy maintains data structures such as routing and endpoint tables which grow as the cluster grows, so the memory consumption per proxy will be higher the larger the cluster gets. Some service meshes today attempt to solve this by pushing partial routing tables to individual proxies, limiting where they can route to.



Let's assume 30 pods/node in a 500 node cluster, a sidecar based architecture will require to run 15K proxies. With 70MB of memory consumed per proxy (already assuming heavily optimized routing tables), this still results in 1.5TB of memory consumed by all sidecars in the cluster. In a per-node model, with the same assumed memory footprint per proxy, the 500 proxies will consume no more than 34GB of memory.

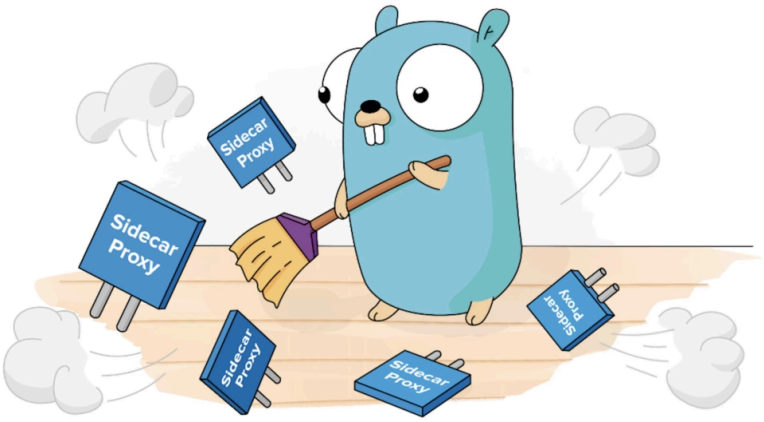
Multi-Tenancy

As we move from a sidecar model to a per-node model, the proxy will serve connections for multiple applications. The proxy has to become multi-tenant aware. This is exactly the same transition that has happened as we switched from using individual virtual machines to using containers instead. As we stopped using an entirely separate copy of the operating system running in each virtual machine and started sharing the operating system with multiple applications, Linux has to become multi-tenant aware. This is why namespaces and cgroups exist. Without them, a single container could consume all of the resources of a system and containers could access each other's filesystems in an uncontrolled manner.



Wouldn't it be great if this behaved exactly the same for network resources at the service mesh level? Envoy already has an initial concept of namespaces, they are called listeners. Listeners can carry individual configurations and operate independently. This will open entirely new possibilities: all of a sudden, we can easily control resource consumption and establish fair queueing rules, and distribute the available resources either equally to all applications or according to specified rules. This can and should look exactly the same as the way we define CPU and memory constraints of applications in Kubernetes today. If you want to learn more about this topic, I've spoken at EnvoyCon about this (Envoy Namespaces – Operating an Envoy-based Service Mesh at a Fraction of the Cost, Thomas Graf, EnvoyCon 2019).

Want to get Involved? – Join the Cilium Service Mesh Beta



Alongside the upcoming Cilium 1.11 release, the Cilium community is hosting a new Cilium Service Mesh beta program. It features a new build that will make the following functionality available:

- L7 Traffic Management & Load-balancing (HTTP, gRPC, ...)
- Topology Aware Routing across clusters, clouds, and premises
- TLS Termination
- Canary Rollouts, Retries, Rate Limiting, Circuit Breaking, etc, configured through Envoy
- Tracing with OpenTelemetry & Jaeger integration
- Built-in Kubernetes Ingress Support

All of the above features are available in a feature branch on github.com/cilium/cilium. The beta program allows Cilium maintainers to engage directly with users about their needs. To sign up, you can directly fill out this form or you can read more about the program in the announcement by the Cilium community.

Conclusion

eBPF is the answer to provide a native and highly efficient service mesh implementation. It will free us from the sidecar model and allows to integrate existing proxy technology into existing kernel namespacing concepts allowing them to become part of the beautiful container abstraction we already use every day. On top of that, eBPF will be able to

offload more and more of the functionality that is currently performed by proxies to reduce the overhead and complexity even further. By being able to integrate almost any existing proxy, the architecture also allows integrate with most existing service mesh control planes (Istio, SMI, Linkerd, ...). This will make the benefits of eBPF available to a wide set of end-users while decoupling the datapath efficiency and overhead discussion from control plane aspects.

If you are interested in exploring this space, we would love to hear from you. Feel free to get in contact by reaching out on Twitter or the eBPF & Cilium Slack.

Further Reading

- How eBPF Streamlines the Service Mesh, Liz Rice, The New Stack
- Cilium Service Mesh Beta Program, Cilium Community
- Learn more about Cilium

AUTHOR

Thomas Graf

CTO & Co-Founder Isovalent, Co-Creator Cilium, Chair eBPF Governing Board

Thomas Graf is one of the creators of Cilium and the CTO & Co-Founder of Isovalent, the company behind Cilium. Before that, Thomas spent 15 years as a kernel developer working on the Linux kernel in networking, security, and eventually eBPF.