Exploring Linkerd as a Service Mesh for Observability, Reliability, and Security

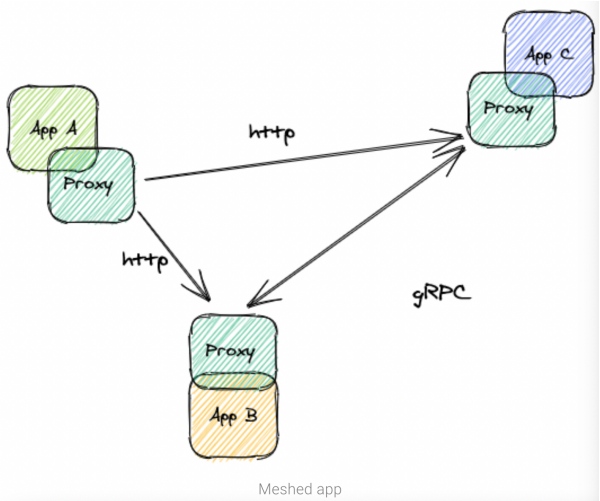
Ayushman Khazanchi

DD2482
May 2022

Introduction to service mesh

Service Mesh is a tool for adding observability, reliability, and security features to applications by inserting these features at the platform layer of the deployment as opposed to the more traditional application layer. It acts as a dedicated infrastructure layer for your network. A service mesh is made up of at least three components: Proxy, Data Plane, and Control Plane.

Modern applications are often deployed in a microservices-based architecture where one service might need to interact with many other services to complete its own task. This creates an environment where the network calls could create a map of thousands of interlinking between different services. All three components of the service mesh together offer insight into the network traffic at a much more granular level and also offer a way to control how different parts of an application interact with each other. Doing the same with service to service communications built into the applications themselves can become incredibly complex at scale and hard to observe since the network, having been embedded into the application layer, is no longer fully transparent.



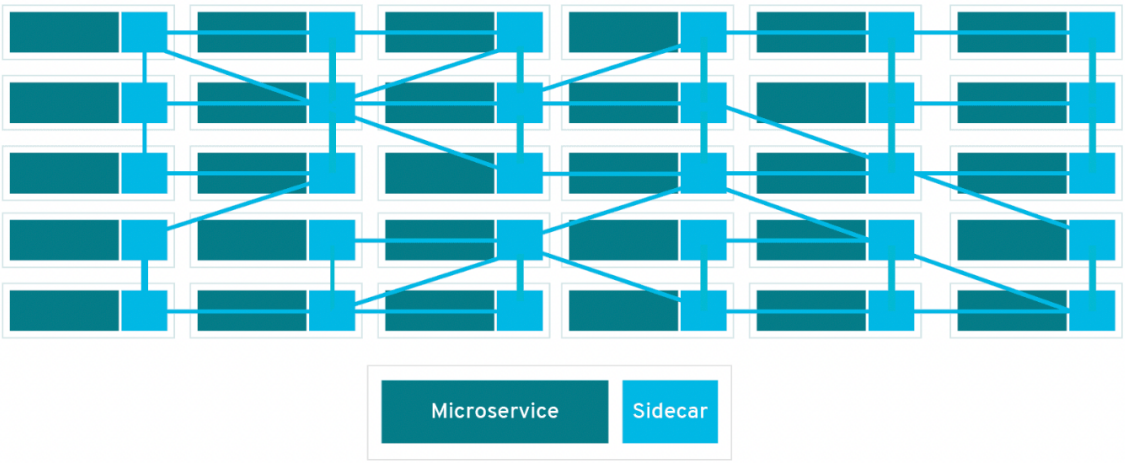
Meshed app

By abstracting the networking layer to another infrastructure layer a service mesh can offer not just network transparency but also higher security and better load performance by routing requests away from unavailable nodes and towards those that have lesser load.

Service Mesh Architecture

Proxy

At its core, a service mesh is built on the idea of "service linking". It is not a mesh of services but instead a mesh of proxies that are invoked by services running locally. These proxies are Layer 7 TCP proxies that act as both forward and reverse proxies. In service mesh terms they are often referred to as "sidecar" because they sit alongside a service and not within them. The brand of proxy itself doesn't matter and can be different based on different tools.



Data Plane

These proxies relay calls to each other from their respective services thereby creating a mesh network. This mesh network is known as the data plane. With each proxy transparently intercepting calls to and from their service they're able to extract metrics and apply policy-level changes at the infrastructure layer. This helps in providing not just a more secure foundation for your deployments but also provides greater visibility into the network.

Control Plane

The control plane is a collection of APIs and tools that provide a centralized source for interacting with the data plane. They also provide a centralized source for user interaction and often contain a dashboard for the administrator to monitor and

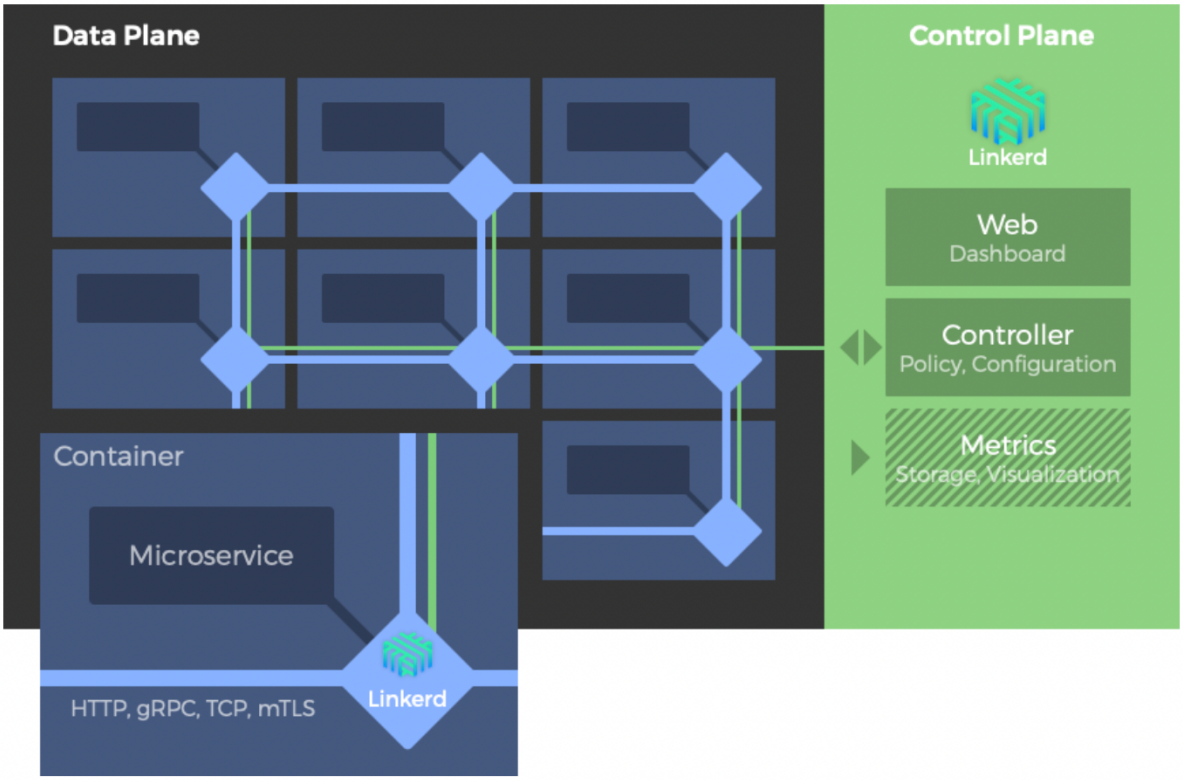
interact with. The control plane is also responsible for sending out commands (via APIs or gRPC calls) to the data plane to prepare and setup TLS certificates, metrics aggregation, and policies among other things.

Introduction to Linkerd

Linkerd is among the most popular open-source service mesh technologies available today. It is developed primarily by Buoyant. Linkerd works by installing a set of "ultralight, transparent proxies"[10] alongwith each application instance. The proxies handle all traffic between the services and because they're transparent they are able to access network metrics and telemetry data from across the network. This design lets Linkerd ensure they are able to "measure and manipulate traffic to and from your service without introducing excessive latency"[10].

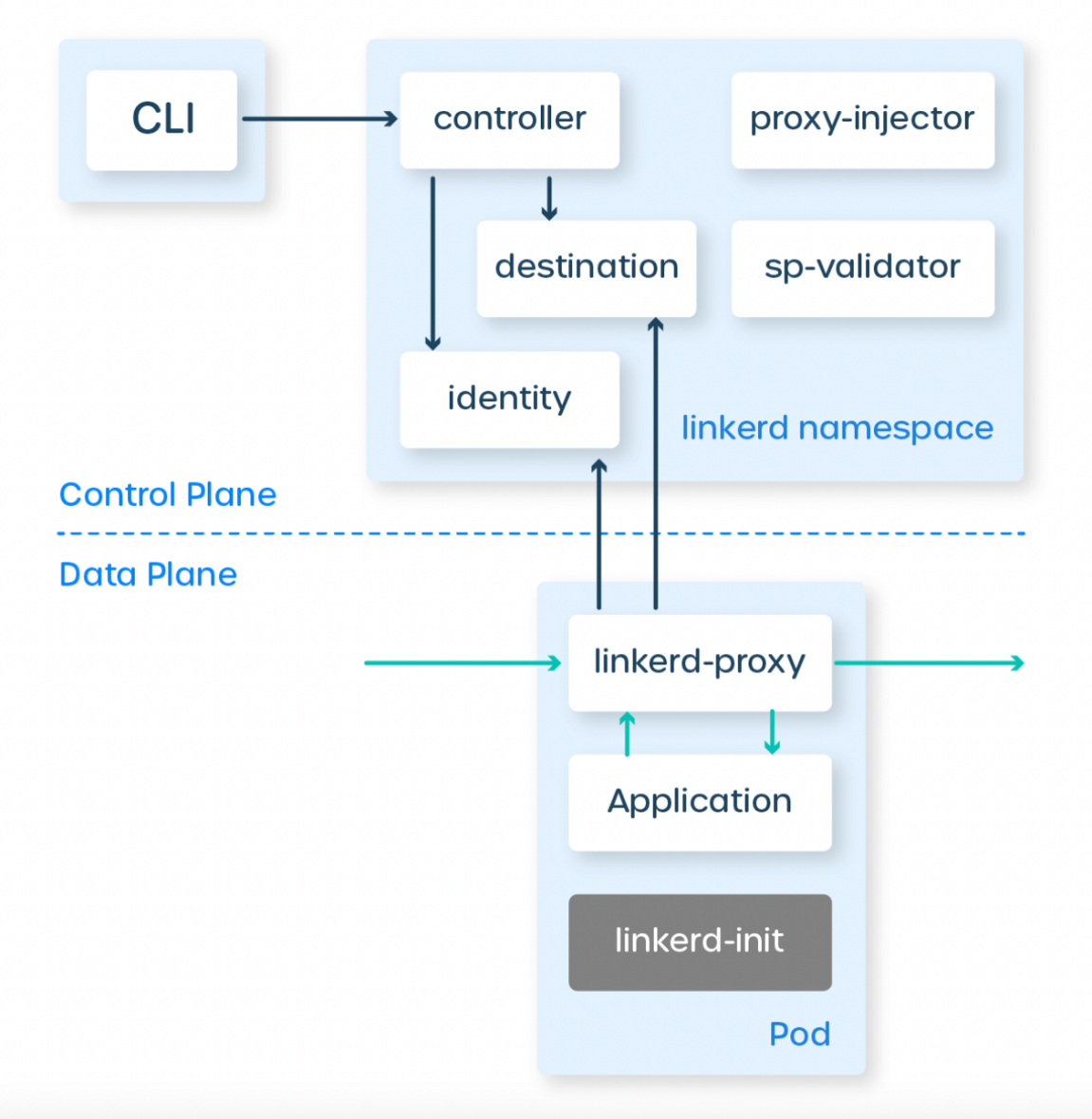
Architecture

On a high level the Linkerd architecture looks pretty much the same as any service mesh architecture. Linkerd deployed on Kubernetes – where it is most frequently used – sits inside its own cluster with its proxy deployed on each microservice pod. The proxies then communicate with each other to create a mesh network (data plane) and are informed or manipulated by the control plane as shown below.



A typical Linkerd deployment on Kubernetes. The control plane consists of a controller, a metrics and a web component. The data plane consists of Linkerd proxies.

In case of Linkerd the control plane includes a main controller component, a web component that provides access to the dashboard, and a metrics component containing instances of Prometheus and Grafana. The control plane also includes components for service discovery, identity (certificate authority), and public-api for web and CLI endpoints. With the help of these components, the control plane is able to manipulate the ultralight Linkerd2-proxy installations across the network for its needs. The control plane components are the ones responsible for implementing retries, adding mutual TLS (mTLS), and performing request load balancing. The data plane, by contrast, is a collection of all the individual linkerd-proxy next to each application instance. The blue boxes in the diagram represent the Kubernetes pod boundaries showing how the proxy runs inside the pod itself. This is a representative logical diagram. When deployed in a real-world environment the control plane may expand to three or five instances and the data plane to hundreds or thousands.



In the figure above you can see the several different components of Linkerd's control plane along with a single Linkerd2-proxy sitting alongside a service instance. Many service mesh use Envoy as a proxy layer but Linkerd uses a small-scale micro-proxy written in Rust called Linkerd2-proxy which it has built from the ground up to be highly-performant and requiring low footprint. Linkerd2-proxy is designed to be secure, lightweight, and to consume a lot fewer resources than something like Envoy which can have a bigger footprint on resource usage.

Observability

Since Linkerd, and most service mesh, are designed to be used in conjunction with microservice architecture, they work best when deployed on something like Kubernetes as they can rely heavily on the underlying Kubernetes constructs. This helps Linkerd easily see the health of the app by surfacing metrics by namespace, deployments, pods, and even clusters. Linkerd is able to tap into inter-app communication using its transparent proxies and allows a visual look into the live request-response metadata of each network call. This offers increased observability at a level that is not possible when the service-to-service communication is baked into the application layer. Having abstracted network as a layer complex activities like tracing network calls and diagnosis of network issues becomes a trivial task.

Reliability

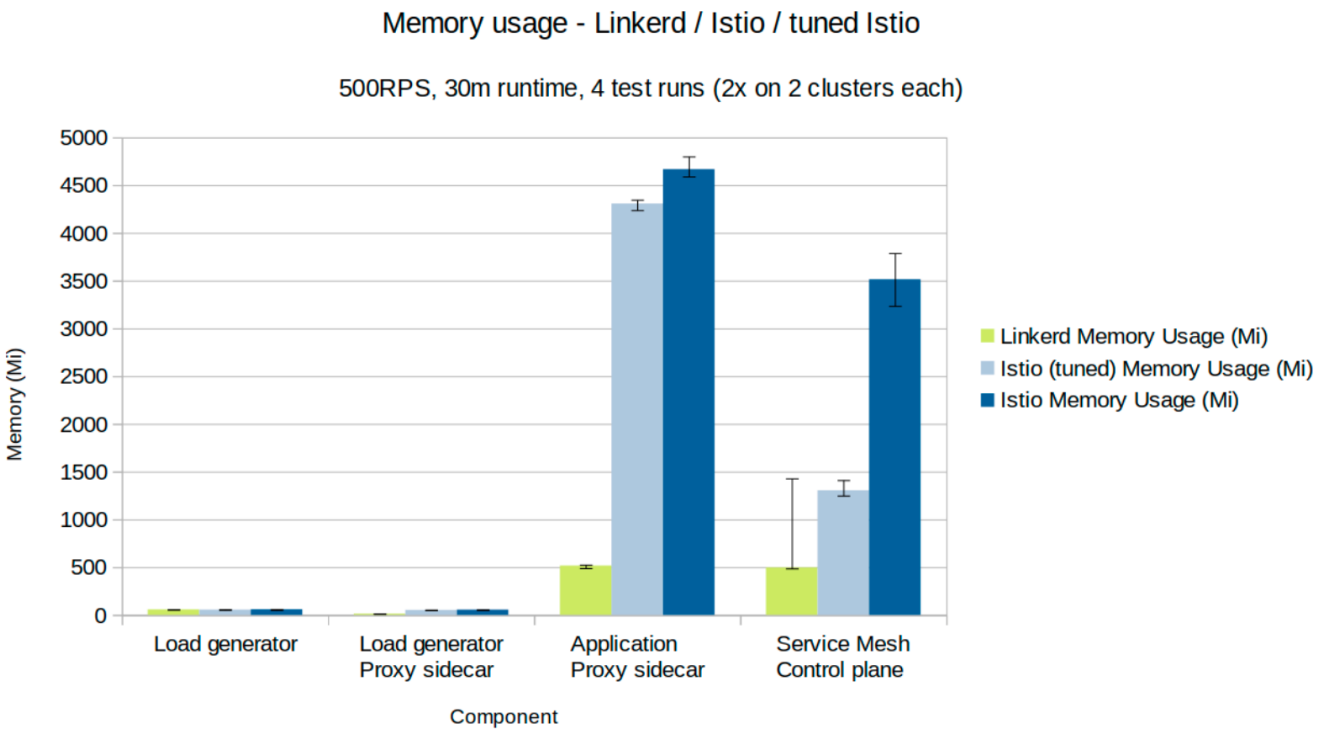
Linkerd also uses native Kubernetes constructs to improve reliability and performance benefits for services. Services can offload routing and DNS strategies to the proxy layers and benefit from smarter routing decisions so that requests don't end up at failed instances. Linkerd proxy also offers better performance by simply monitoring which nodes receive traffic frequently and route some requests away from those nodes towards less busy nodes using an implementation of an exponentially weighted moving average algorithm. Since the Linkerd2-proxy layer acts as both a forward and reverse proxy it can also be configured to allow retries timeouts and provide "latency-aware Layer 7 load balancing for HTTP traffic and Layer 4 load balancing for non-HTTP traffic" [5].

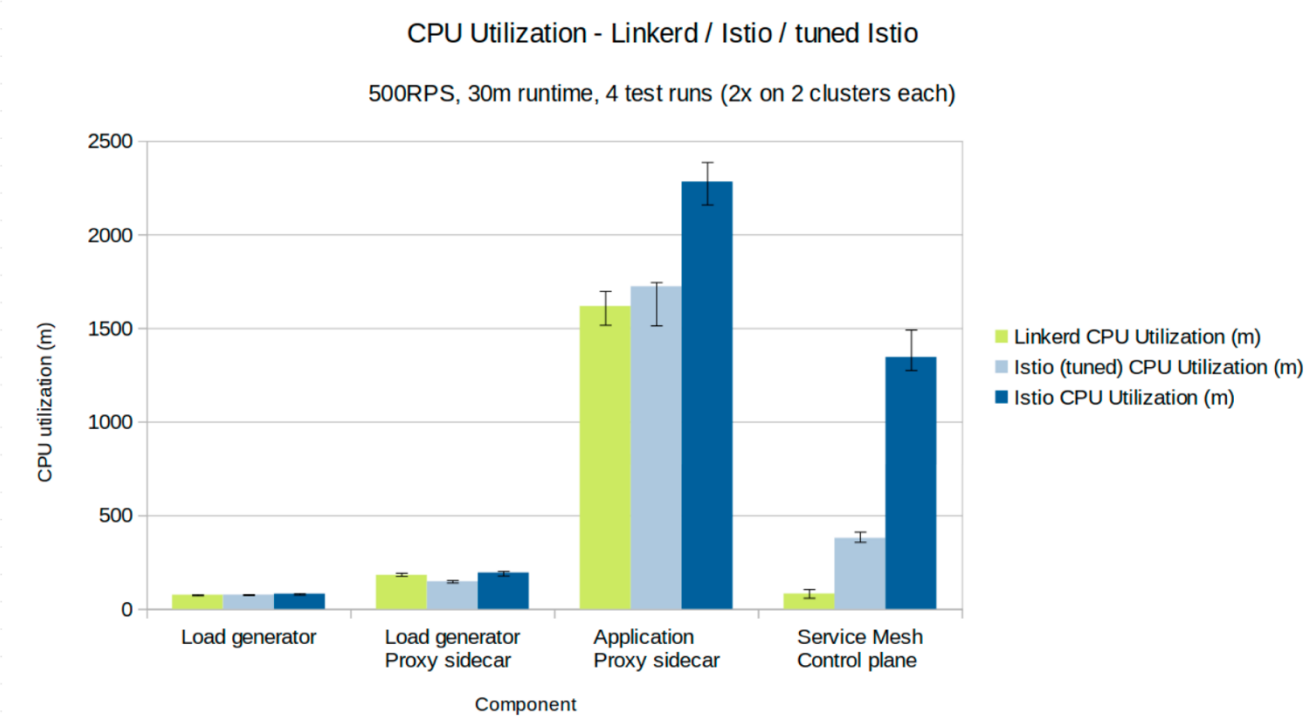
Security

Linkerd offers zero-trust protection on Kubernetes environments. It does so by simplifying security and transparently and automatically enabling mutual TLS (mTLS) between all services. The idea behind the design is to have always-on security by default and make it as minimal config as possible from the operator level. Since security is often made more resilient by being in the open for thorough testing, Linkerd has also participated in independent security audits so as to ensure growing trust in the service [2].

Competitive Advantage

These days, the service mesh space is buzzing with names. However, Linkerd emerges in many ways as not just a competitive candidate but a leading one at that. Unlike its competitors, Linkerd can be deployed as a sidecar on an individual host basis making it an economical and flexible choice as shown in the diagrams below [11]. Linkerd requires no configuration out of the box and is thus also relatively easier to setup compared to its competitors like Istio or Hashicorp Consul. Out of the box it comes with support for HTTP/2, gRPC, and other common protocols. It allows TLS setup throughout the environment and distributes traffic highly intelligently using modern load-balancing algorithms. It provides dynamic request routing and distributed tracing to allow for easier debugging of root cause of issues. Being an open-source software itself, it provides built-in integration with other open-source projects such as Prometheus and Grafana for visibility into telemetry and analytics. All these features combined ensure Linkerd provides a high level of resilience at a low resource footprint that offers it a slight competitive advantage over its peers. To add to that Linkerd takes up a smaller resource footprint than its more recognized competitor, Istio.



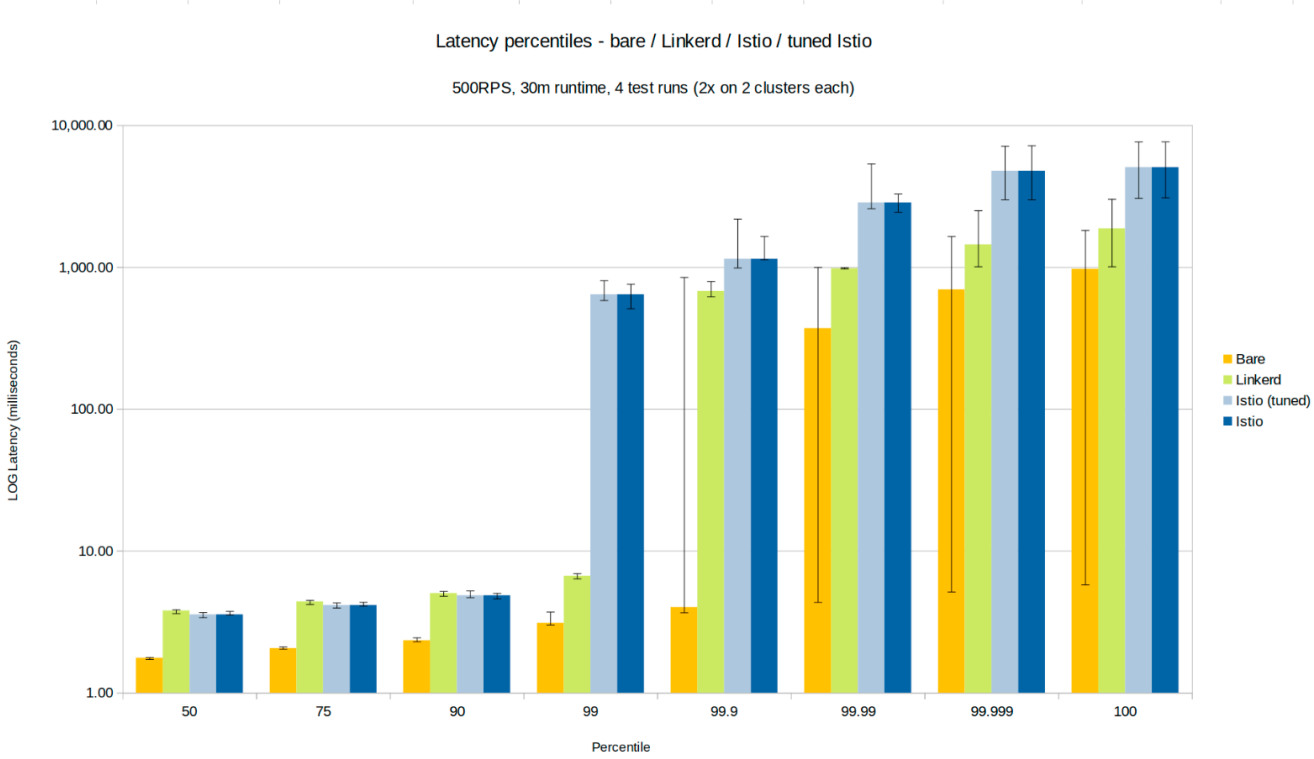


Drawbacks of Service Mesh

Service meshes are designed to operate in the most ideal way in a microservices-based architecture. Being built on the idea of service-to-service communication, a service mesh operates best in an environment where service-to-service calls are pre-optimized. Often this implies a Kubernetes environment where the underlying constructs are used widely in the implementation. Despite the strong coupling, a dedicated infrastructure layer that covers all network communication can seem an invasive implementation and thus requires a uniform environment that takes into account the granularity of design and a uniform tech stack. They can be a complex technology that is difficult to retrofit into environments that are not microservice or Kubernetes oriented and thus don't always make sense in those contexts. A monolith with some microservices can certainly work with a service mesh but a service mesh may not offer the same level of visibility and reliability in a monolith architecture as it would in a microservice one. As a result, it's not a viable solution in some contexts and is best applied to newly developed microservice based applications. Ideally ones that run on Kubernetes.

At the very best, service meshes include a proxy on each pod that contains a service. All requests to and from one service are routed through its proxy. As a result, service meshes are often criticized for their poor performance and lack of support of hybrid-cloud topologies. In application landscapes that are constantly evolving, a service mesh can offer an overhead that may be too much to handle. With apps and services changing at a fast pace keeping up with service mesh technology can seem like an

unnecessary exercise. Additionally, they can have a considerable impact on application performance as compared to direct calls across the network which can often be difficult to diagnose. If the debugger itself is failing, then the debugging can take a lot longer. Thanks however to its extensive focus on simplicity, minimal configuration, and lightweight proxy, Linkerd does well in trying to get close to this bare metal performance.



Logarithmic Latency (in milliseconds) for 500 requests per second

One of the biggest concerns against service mesh technologies is the added overall complexity to a deployment. If their resource footprint is high, they can also come with poor performance and gaps in hybrid, multi-cluster deployments. Since they come with certain built-in tooling, they offer opinionated solutions that may not exist beforehand in the environment thus leading to more integration and education overhead. For example, if a deployment doesn't use Prometheus, adding Linkerd into the environment introduces the overhead of maintaining a new tool. If a service mesh fails at any point, it creates additional overhead of debugging the debugger. For many companies the service mesh deployment is a fairly new exercise and time spent on understanding the tooling may take away time from more productive endeavors. As a result, a service mesh can often become difficult to justify for a hybrid, dynamic, and fast-evolving environment.

The architecture of the service mesh itself has some direct implications. Adding a proxy to every pod can quickly become a massive increase in resource requirement and only really makes sense if the environment is built with microservice-first architecture. Heavy use of proxies implies that the proxies should really be fast since at least two hops are added to each network call and that they need to be small and lightweight in order to not affect performance. Additionally, there needs to some sort of configuration and deployment management solution when deploying proxies at such a vast scale.

Benefits of Service Mesh

Despite their drawbacks, service meshes do offer a state-of-the-art solution to make service-to-service communication resilient. They offer high observability and security across the entire stack and configure the network as a dedicated infrastructure layer. This allows abstracting all network failures and allows for easy debugging in failure or performance issues. It allows for increased visibility into debugging the network layer that may otherwise have been bundled into the application.

As we've observed, service meshes are best used in stand-alone, microservice-based or Kubernetes-forward environments that allow for individual deployments and updates. Each new service that is added to such an environment can easily be incorporated into the mesh by simply adding a proxy along the service. Over time the data captured by the service mesh can be applied as policy or service network rules to make the system more efficient and reliable. By providing visibility at the network layer a service mesh can offer insight into routes or ports that may well be closed thereby increasing security of the system. These performance metrics can then offer other ways to further optimize communication channels between microservices.

Conclusion

We've understood that service meshes require a fair amount of Layer 7 uniformity. They offer tracing, intelligent routing, health checks, mTLS, identity, and reliability. It is best if a service mesh offers endpoints with similar weight and granularity and provide multi-tenant solutions. If the API endpoints are too non-uniform, then it can result in metrics that are less comparable across the board. As a result, observability may suffer, and the promise of visibility may be lost. Additionally, service meshes suffer if not all the services are part of the mesh. This is one of the biggest reasons they are not as easily adopted in multi-cluster and hybrid environments. Uniformity is also required as they work best in environments that benefit from having a self-contained microservice-based model with complex logic and interdependent communication calls that may be constructed as an independent network layer.

Many applications today are not written in the service-based communication style of independent microservices. Many applications are still monolithic behemoths that are slowly migrating towards a more micro-service-based style. As a result, it's possible that for these types of applications a service mesh can seem like a problem looking for a solution. In fact, using a service mesh in such an environment may not even be suitable. However, there are still immense benefits to adding solutions like Linkerd to the ecosystem as they allow a relatively low footprint entry into the world of service meshes. They can offer a faster and more uniform approach into the world of service-to-service communication between microservices.

References

1. Anjali Khatri; Vikram Khatri (2020). Mastering Service Mesh: Enhance, Secure, and Observe Cloud-native Applications with Istio, Linkerd, and Consul. Packt Publishing. p. 39. ISBN 9781789611946.

2. Cure53, Pentest-Report Linkerd2 and Linkerd2 Proxy 06.2019, https://github.com/linkerd/linkerd2/blob/main/SECURITY_AUDIT.pdf

3. William Morgan; The Service Mesh – What every software engineer needs to know about the world's most over-hyped technology; https://buoyant.io/service-mesh-manifesto

4. Jason Morgan; Introduction to the service mesh—the easy way; https://linkerd.io/2021/04/01/introduction-to-the-service-mesh

5. Tobias Kunze; A brief introduction to Linkerd; https://glasnostic.com/blog/an-introduction-to-what-is-linkerd-service-mesh

6. Redhat; What's a Service Mesh?; https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh

7. Tobias Kunze; Should I use a Service Mesh?; https://glasnostic.com/blog/should-i-use-a-service-mesh

8. Tobias Kunze; What is a Service Mesh?; https://glasnostic.com/blog/what-is-a-service-mesh-istio-linkerd-envoy-consul

9. William, Morgan; Why Linkerd doesn't use Envoy, https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy

10. Linkerd; Overview; https://linkerd.io/2.11/overview

11. Thilo Fromm; Performance Benchmark Analysis of Istio and Linkerd; https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd

12. Gursimran Singh; Linkerd Benefits and its Best Practices; xenonstack.com/insights/what-is-linkerd

13. William Morgan; Linkerd Benchmarks; https://linkerd.io/2019/05/18/linkerd-benchmarks/