ISTIO SERVICE MESH DEEP-DIVE

Table of Contents

Introduction

Enterprises nowadays are keen on adopting a microservices architecture, given its agility and flexibility. Containers and the rise of Kubernetes — the go-to container orchestration tool — made the transformation from monolith to microservices easier for them.

However, a new set of challenges emerged while using microservices architecture at scale:

- It became hard for DevOps and architects to manage traffic between services.
- As microservices are deployed into multiple clusters and clouds, data goes out of the (firewall) perimeter and is vulnerable; security becomes a big issue.
- Getting overall visibility into the network topology became a nightmare for SREs.

Implementing new security tools, or tuning existing API gateway or Ingress controllers, is just a patchwork and not a complete solution to solve the above problems. What architects need is a radical implementation of their infrastructure to deal with their growing network, security, and observability challenges. And that is where the concept of service mesh comes in.

What is a service mesh?

A service mesh decouples the communication between services from the application layer to the infrastructure layer. The abstraction at the infrastructure level happens by proxying the traffic between services (see Fig. A).

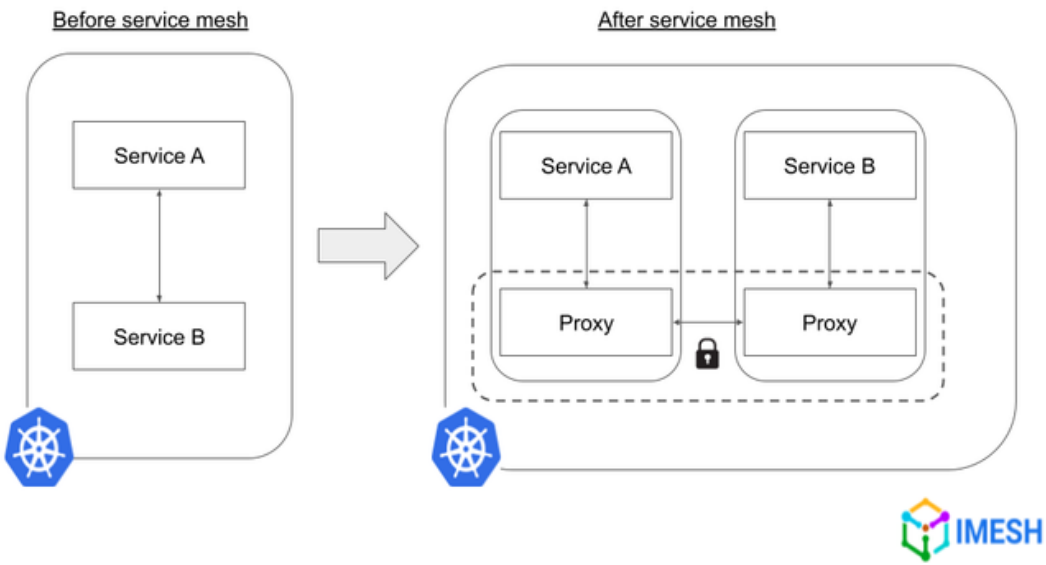

Fig A – Service-to-service communication before and after service mesh implementation

The proxy is deployed alongside the application as a sidecar container. The traffic that goes in and out of the service is intercepted by the proxy and it provides advanced traffic management and security features. On top of it, service mesh provides observability into the overall network topology.

In a service mesh architecture, the mesh of proxies is called the data plane, and the controller responsible for configuring and managing the data plane proxies is called the control plane.

Why do you need a service mesh for Kubernetes?

While starting off, most DevOps only have a handful of services to deal with. As the applications scale and the number of services increases, managing the network and security becomes complex.

Tedious security compliance

Applications deployed in multiple clusters from different cloud vendors talk to each other over the network. It is essential for such traffic to comply with certain standards to keep out intruders and to ensure secure communication.

The problem is that security policies are typically cluster-local and do not work across cluster boundaries. This points to a need for a solution that can enforce consistent security policies across clusters.

Chaotic network management

DevOps engineers would often need to control the traffic flow to services — to perform canary deployments, for example. And they also would want to test the resiliency and reliability of the system by injecting faults and implementing circuit breakers.

Achieving such kinds of granular controls over the network requires DevOps engineers to create a lot of configurations and scripting in Kubernetes and the cloud environment.

Lack of visualization over the network

With applications distributed over a network and communications happening between them, it becomes hard for SREs to keep track of the health and performance of the network infrastructure. This severely impedes their ability to identify and troubleshoot network issues.

Implementing service mesh solves the above problems by providing features, which make managing applications deployed to Kubernetes painless.

What is Istio service mesh?

Istio is an open-source service mesh platform that simplifies and secures traffic between microservices. Istio provides a dedicated infrastructure for traffic management, security, and observability, to help developers handle the network of microservices in Kubernetes and multiple clouds, at scale.

Istio works by deploying Envoy proxy- a L4 and L7 layer proxy- alongside each microservice. The proxy intercepts and handles service-to-service traffic, and thus abstracts communication logic from the service/application layer into a dedicated infrastructure layer (refer to fig. B).

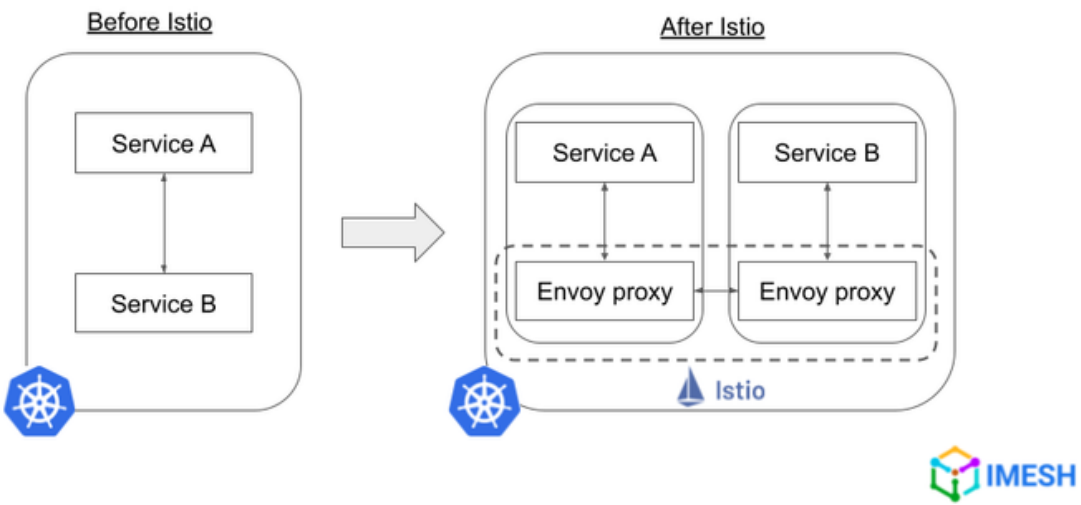

Fig. B – Service-to-service communication before and after deploying Istio service mesh in a Kubernetes cluster

Istio components and sidecar architecture

Istio has two main components that control and manage the entire network infrastructure layer (refer to Fig. C):

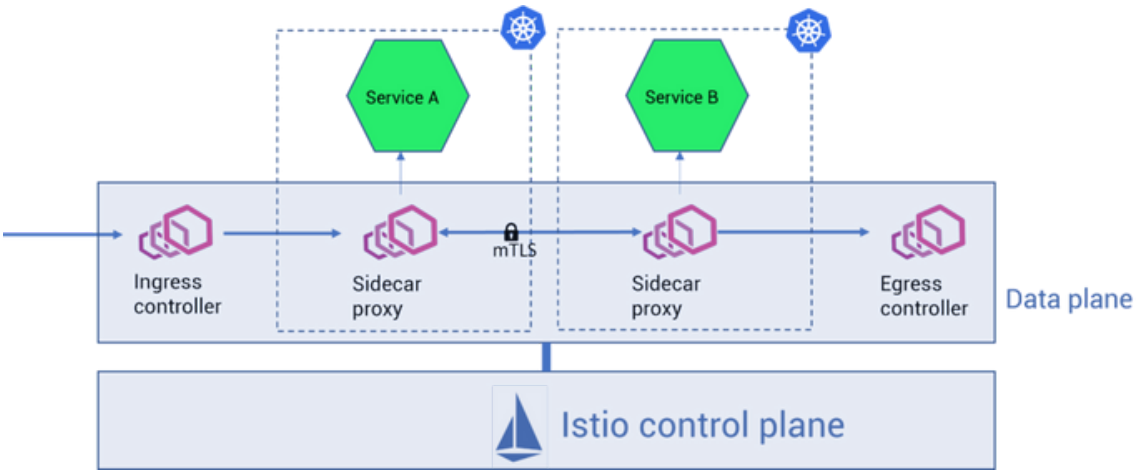

Fig. C – Istio sidecar architecture

- **Data plane:** It is a network of Envoy proxies that handle the communication between services in the mesh. An Envoy is a lightweight proxy deployed as a sidecar alongside each service in a mesh, intercepting the traffic between that particular service and other services.

  Envoy proxy data plane routes and controls the request flow between services, apart from providing service discovery, security (mTLS), network resiliency (retries, timeouts, circuit breakers, fault injection), and observability (logs, traces, and metrics) features.

Istio components and sidecar architecture

- **Control plane:** The control plane interacts with the data plane and provides a centralized management and configuration layer for data plane proxies. It converts high-level routing rules that define traffic control behaviour into Envoy-specific configurations.

Earlier control plane components were divided into Pilot, Galley, Citadel, and Mixer.

Fig E: The image highlights the containers inside a Kubernetes pod which is Istio-enabled.

Now, control plane functionalities are consolidated into a single binary called istiod. Istiod handles service discovery, configuration, and certificate management for service communication in the mesh.

Features of Istio service mesh

Istio provides several features for various IT teams in an organization such as ***traffic management***, ***security***, ***observability***, and ***extensibility***.

Traffic management

Istio can perform L7 and L4 routing (we will understand in the later chapters). Istio automatically detects all the endpoints of respective services in the mesh and stores them in its internal service registry. It helps Istio to manage traffic by load balancing between replica pods and apply fine-grained control over traffic splitting between services, like a canary deployment.

In a nutshell, these are the traffic management and network reliability features that Istio provides:

- **Load balancing:** Istio provides L3/L4 and L7 features on HTTP calls (HTTP/2, gRPC) to implement advanced traffic management features.
- **Timeouts:** It is the timeframe within which a service call succeeds or fails. Timeouts are helpful so that services do not hang due to the Envoy proxy waiting for replies forever.
- **Retries:** It is the number of times Envoy should try to connect to a service when the initial connection attempt fails.
- **Circuit breakers:** It is the threshold for calls to specific hosts within a service. Once the threshold has been reached, further connections to the host are prevented.
- **Fault injection:** It is a testing method for systems where errors are deliberately introduced to verify their ability to recover from error conditions.
- **Support for protocols:** Istio by default, supports many network protocols out of the box, such as HTTP, HTTPS, gRPC, TCP, UDP, etc.

Features of Istio service mesh

Security

Istio helps secure the microservices network by facilitating granular authentication, authorization, and access control policies.

- **Authentication**: Istio verifies the identity of users (humans or machines) by allowing peer-to-peer and request authentication policies.
    - **Peer authentication** involves mTLS implementation, where communication between services is encrypted and authenticated using certificates issued for both the client and the server.
    - **Request authentication** involves server-side verification, where the client has to attach JWT (JSON Web Token) to the request.
- **Authorization:** Verifying whether the authenticated user can access a server and perform specific actions is done using authorization policies in Istio. Authorization policies can be set to allow, deny, or execute custom actions against an incoming request based on different parameters.
- **Access control:** Istio controls the access of authorized users to resources by implementing the least privilege policy and role-based access control (RBAC). Istio supports RBAC policies to be set on method, service, and namespace levels.

Istio helps secure the network by automating key and certificate rotation at scale.

Observability

Istio offers observability and real-time visibility into applications' performance and behavior by providing detailed telemetry for traffic flow between services in the mesh.

Istio generates the following types of telemetry:

Features of Istio service mesh

- **Metrics:** Istio generates service metrics for real-time performance monitoring of services. They are based on the four “golden signals” of monitoring: latency, traffic, errors, and saturation.

- **Distributed traces:** Istio collects activity traces across multiple services in a mesh to better understand service dependency and traffic flow.

- **Access logs:** Istio can produce a complete record of communication between services in the mesh, making it easier to understand the behavior of each workload.

In addition to providing add-ons for tools such as Prometheus, Grafana, Kiali, and Jaeger, Istio can be integrated with Apache Skywalking for unified observability.

Extensibility

Istio provides the ability to extend proxy functionality using WebAssembly (Wasm), which is a sandboxing technology that can be used to extend the Istio proxy (Envoy). Istio does it by replacing the primary extension mechanism in it called Mixer.

With Wasm, users can build support for new protocols, custom metrics, loggers, and other filters. And these Wasm modules can be distributed dynamically at runtime.

The following are some of the open-source projects that have emerged over Istio using WASM:

- Slime – an intelligent service manager to uses Istio and Envoy
- MOSN – provides cloud-native edge gateways and agents
- Aeraki – includes support for all the L7 protocols apart from HTTPs and gRPC

How does Envoy proxy or sidecar work?

In the earlier section, we discussed the data plane and control plane components of Istio. The data plane consists of a set of Envoy proxies deployed as sidecars. These proxies are responsible to manage and control all communication between microservices.

The sidecar is automatically injected into the application whenever we implement Istio (or make a service or namespace Istio enabled). And traffic is directed to and from the application services from these sidecars without developers worrying about it. In this section, we will discuss how sidecar injection works in an application container and how an Envoy proxy deals with traffic management.

Sidecar injection

Sidecar injection is a process of configuring additional containers for the pod template, which are istio-init and istio-proxy (Envoy Proxy). Sidecars can be injected automatically or manually. When the injection is complete, you will see that Istio has injected an initContainer (istio-init) and sidecar proxy-related configurations into the original pod template.

Fig E: The image highlights the containers inside a Kubernetes pod which is Istio-enabled.

How does Envoy proxy or sidecar work?

istio-init

This is an init-container used to manipulate iptables rules (refer to Fig. D) to manage inbound/outbound traffic through the side-car proxy. The istio-init container will run before the application container and will run to completion.

Multiple Init containers can be specified in a Pod; if more than one is selected, the Init containers will run sequentially. The next Init container can only be run if the previous Init container must run successfully. Kubernetes only initializes the Pod and runs the application container when all the Init containers have been run.

The Init container starts sequentially during Pod startup after the network and data volumes are initialized. Each container must be successfully exited before the next container can be started. If exiting due to an error will result in a container startup failure, it will retry according to the policy specified in the Pod's restartPolicy. However, if the Pod's restartPolicy is set to Always, the restartPolicy is used when the Init container failed.

The Pod will not become Ready until all Init containers are successful. The ports of the Init containers will not be aggregated in the Service. The Pod that is being initialized is in the Pending state but should set the Initializing state to true. The Init container will automatically terminate once it is run.

To provide more context about the IP table manipulation, you can refer to the Fig. D which highlights the istio-init container setting iptables for handling the traffic. All the inbound and outbound traffic now goes through certain rules: ISTIO_INBOUND, ISTIO_IN_REDIRECT, ISTIO_OUTBOUND, ISTIO_REDIRECT in the iptables firewall before entering into the sidecar container. The sidecar container which is a Envoy proxy is responsible for handling the traffic after intercepting the traffic through the iptables manipulation, and sending it to the app containers or receiving a response from it. Envoy proxy has inbound and outbound handlers to manage and control the traffic. In the next section, we will understand various components in Envoy proxies which are responsible for handling the traffic.

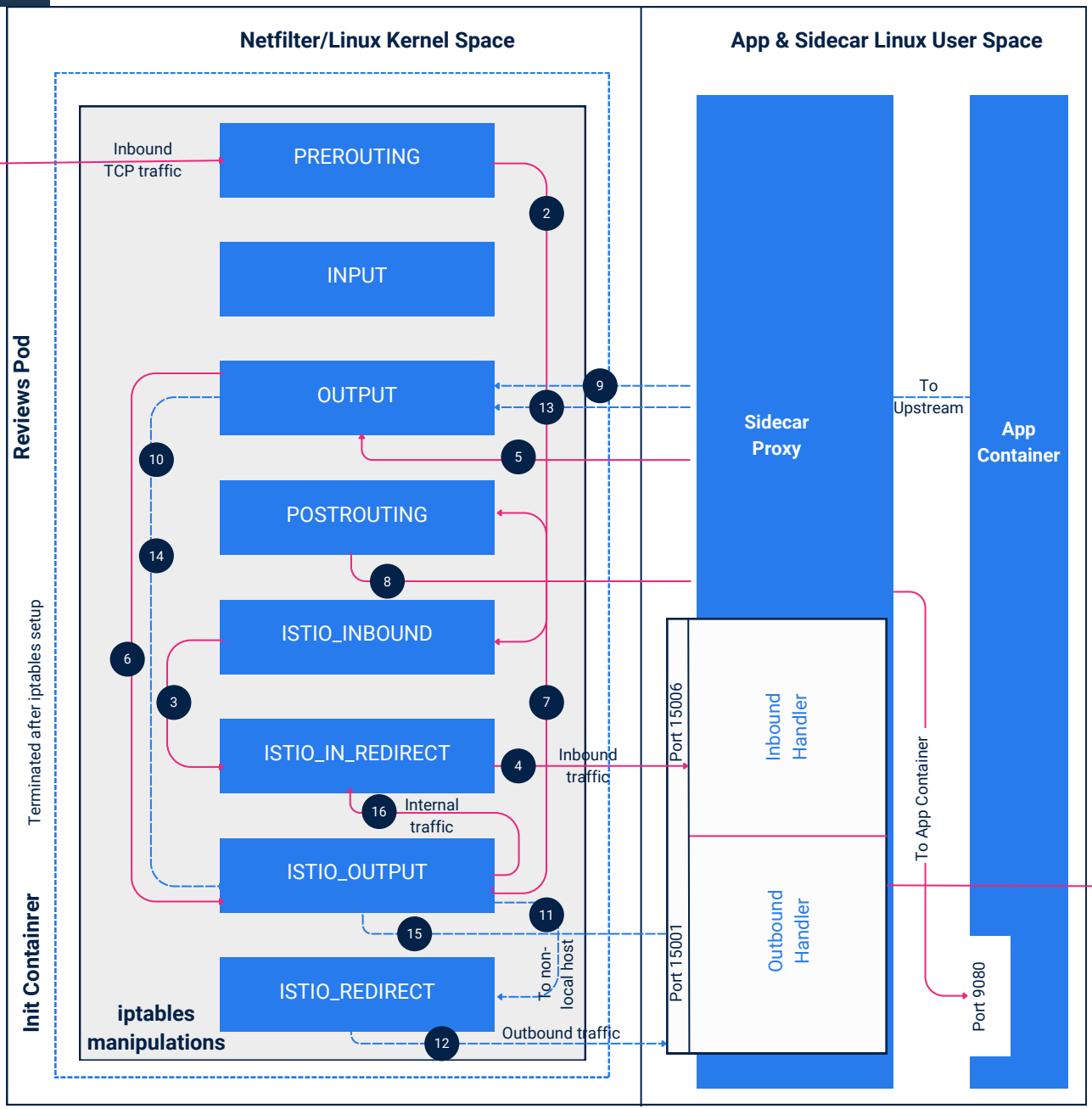

Fig F: The image represents iptable manipulation done by the init container with new rules: ISTIO_INBOUND, ISTIO_IN_REDIRECT, ISTIO_OUTBOUND, ISTIO_REDIRECT

(Sources: Istio.io and jimmysong.io)

How does Envoy proxy or sidecar work?

Envoy proxy or sidecar architecture

Whenever an Envoy proxy receives a connection request from a client (downstream), it will try to parse it and then open a new connection with the server (upstream).
Envoy has 3 modules to handle the traffic and they are Listener, Filter, and Cluster.

1. **Listener:** Module responsible for listening to an external request. The listener binds to a port and awaits the connection. When the request appears, the listener analyzes the connection and decides which filter chain fits the conditions. The most simple case means that each request will be sent through the same set of filters.
2. **Filter:** The filter module allows users to apply network rules on the traffic listened. Examples of a few filters are:
   a. TCP proxy filter: an Envoy filter that allows us to specify the target cluster for the traffic.
   b. HTTP connection manager: This is a special filter that allows us to specify a set of subfilters designed specifically for working with HTTP requests/responses. It is the most crucial filter for proxying REST API applications.
   c. Custom filter: This option allows users to write custom logic, create their own filters, and apply them in the filter pipeline just like others.

There can be more than one filter in a filter chain which can be applied sequentially on a traffic. Usually the last filter in the filter chain is a proxy filter or HTTP manager filter (as it contains routing options). An important thing to remember is that the response received from UPSTREAM will be processed in reverse order. Filters can be configured to work in only one direction.

3.Cluster: Cluster is a module that provides an abstraction for back-end services or UPSTREAM. This describes the target location for the client's connections. If the location consists of multiple addresses user can define the load balancing algorithm, which will be used by Envoy to resolve where exactly the data should be sent.

How load-balancing works for traffic splitting for HTTP and gRPC

This section will delve into how the Istio load-balancing feature works. We will first look at load balancing in the Kubernetes cluster before using the Istio service mesh.

Load management without Istio

In a Kubernetes cluster, services balance the loads into pods (replicas). If service A (4 replicas) is trying to communicate with service B (4 replicas), then a Service A pod will hit the service B endpoint, and service B will load balance the request among its pods. Also, Kubernetes only supports traffic distribution based on instance scaling, which quickly becomes complex if you want granular traffic distribution. Please refer to the image for reference.

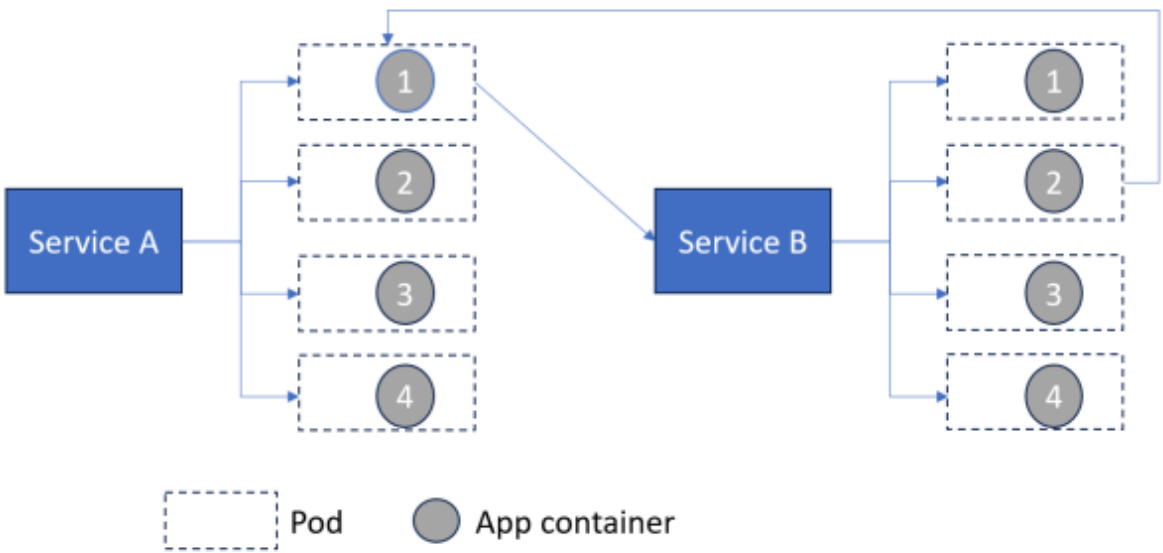

Fig G: The image represents the working of the service to service communication without sidecar proxy

How load-balancing works for traffic splitting for HTTP and gRPC

Load management with Istio and Envoy sidecar

Once you bring any services under the Istio service mesh, Istio's traffic routing rules let you easily control the flow of traffic and API calls between services (refer to the image below). All the service-to-service (including gateway-to-service) communication will be done by Envoy proxies that are deployed along with your services. All traffic that your mesh services send and receive (data plane traffic) is proxied through Envoy, making it easy to direct and control traffic around your mesh without making any changes to your services. Now your DevOps team can implement granular controls such as circuit breakers, timeouts, retries, and traffic splitting for canary rollouts and A/B testing.

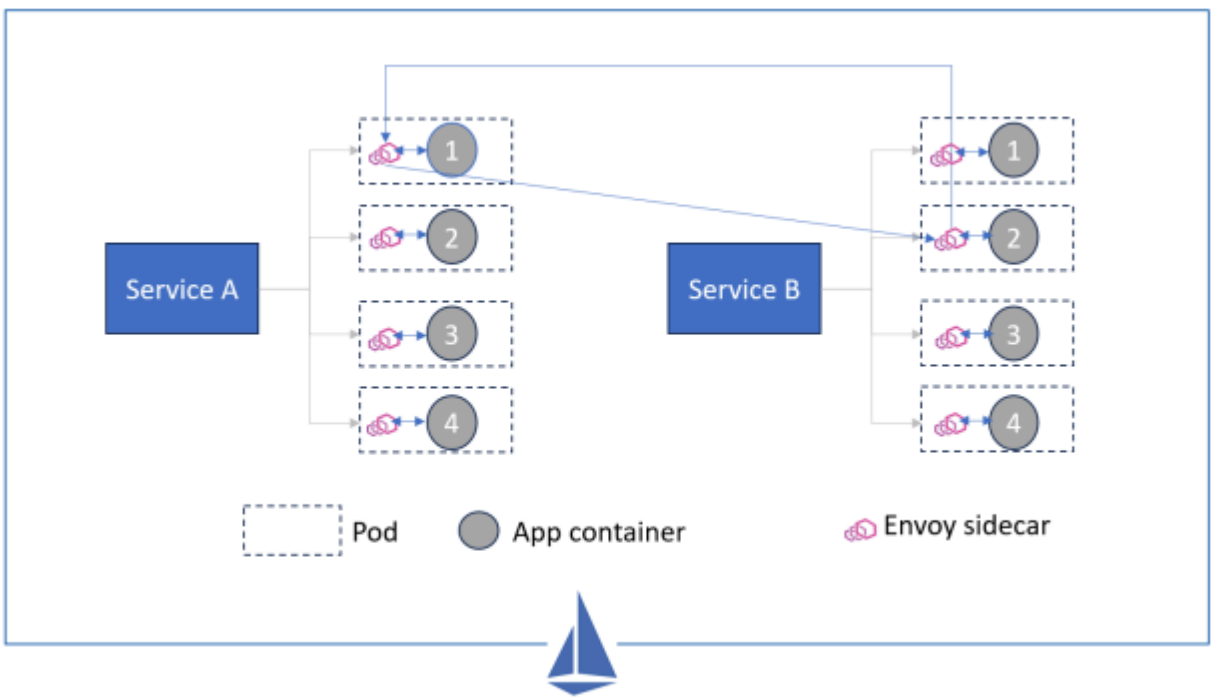

Fig H: The image represents the working of the service to service communication with Envoy proxy

How load-balancing works for traffic splitting for HTTP and gRPC

In order to direct traffic within your mesh, Istio needs to know where all your endpoints are, and which services they belong to. To populate its own service registry, Istio connects to a service discovery system. For example, if you've installed Istio on a Kubernetes cluster, then Istio automatically detects the services and endpoints in that cluster.

Using this service registry, the Envoy proxies can then direct traffic to the relevant services. Most services will have multiple replicas or pods to handle service traffic. By default, the Envoy proxies distribute traffic across each service's load balancing pool using a **least requests model**, where each request is routed to the host with fewer active requests from a random selection of two hosts from the pool; in this way the most heavily loaded host will not receive requests until it is no more loaded than any other host.

While Istio's basic service discovery and load balancing gives you a working service mesh, it's far from all that Istio can do. In many cases you might want more fine-grained control over what happens to your mesh traffic. You might want to direct a particular percentage of traffic to a new version of a service as part of A/B testing, or apply a different load balancing policy to traffic for a particular subset of service instances. You might also want to apply special rules to traffic coming into or out of your mesh, or add an external dependency of your mesh to the service registry. You can do all this and more by adding your own traffic configuration to Istio using Istio's traffic management resources.

How load-balancing works for traffic splitting for HTTP and gRPC

HTTP and gRPC Routing with Istio

Istio can process application-level protocols such as HTTP (HTTP/1.1, HTTP/2, and gRPC), and TLS (including HTTPS). With HTTP, you can route individual HTTP requests, rather than just connections. In addition, several rich attributes are available, such as host, path, headers, query parameters, etc. While TCP and TLS traffic generally behave the same with or without Istio (assuming no configuration has been applied to customize the routing), HTTP has significant differences.

- Istio will load balance individual requests. In general, this is highly desirable, especially in scenarios with long-lived connections such as gRPC and HTTP/2, where connection level load balancing is ineffective.

Requests are routed based on the port and Host header rather than port and IP. This means the destination IP address is effectively ignored. For example, curl *8.8.8.8 -H "Host: review.default.svc.cluster.local"*, would be routed to the *review* Service.

Benefits of Istio service mesh

HTTP and gRPC Routing with Istio

Istio is considered to be resource-intensive, and it has a bit of a learning curve. But the benefits Istio provides outweigh them all. Below are some significant benefits of implementing Istio:

- **5X increased developer experience:** Istio abstracts the network, security, and observability from the core application. Using Istio, developers can work on business logic rather than writing complex networks or AuthN/Z rules. This ensures better productivity, fosters innovation, and improves developer experience.

- **100% zero trust network security:** Istio allows DevOps engineers and security managers to set granular security policies for enforcing strict authentication and authorization for communication between services. Coupled with mTLS-based traffic encryption, these significantly improve the infrastructure's security posture. The centralized way of enforcing policies Istio provides makes compliance easier for security teams, and the policies work across cluster boundaries. This allows them to seamlessly implement a zero-trust network (ZTN) for microservices.

- **Zero-hassle progressive delivery:** With its ability to perform flexible and fine-grained traffic splitting between services, Istio helps DevOps engineers to perform progressive delivery canary and blue-green releases without any hassle.

- **10X faster audit:** Access logs provided by Istio help Auditors analyze the performance of the network over a period of time. The telemetry data helps them quickly identify performance bottlenecks and suggest improvements.

Benefits of Istio service mesh

- **4X faster MTTR from network failures:** Istio helps SREs and Ops teams to have observability and real-time visibility of microservices networks in the cloud. The telemetry data provided by Istio is helpful for them to diagnose and troubleshoot any errors, and restore services as early as possible. The data provides SREs and Ops with end-to-end visibility into the request flow and dependencies between services, enabling them to have multicluster and multicloud visibility to analyze the performance and behavior of applications.

- **99.99% Resilient infrastructure:** With advanced load balancing and traffic management capabilities, cloud architects can ensure a highly available and high-performance network infrastructure for production.