Zero Configuration Service Mesh with On-Demand Cluster Discovery

https://netflixtechblog.com/zero-configuration-service-mesh-with-on-demand-cluster-discovery-ac6483b52a51

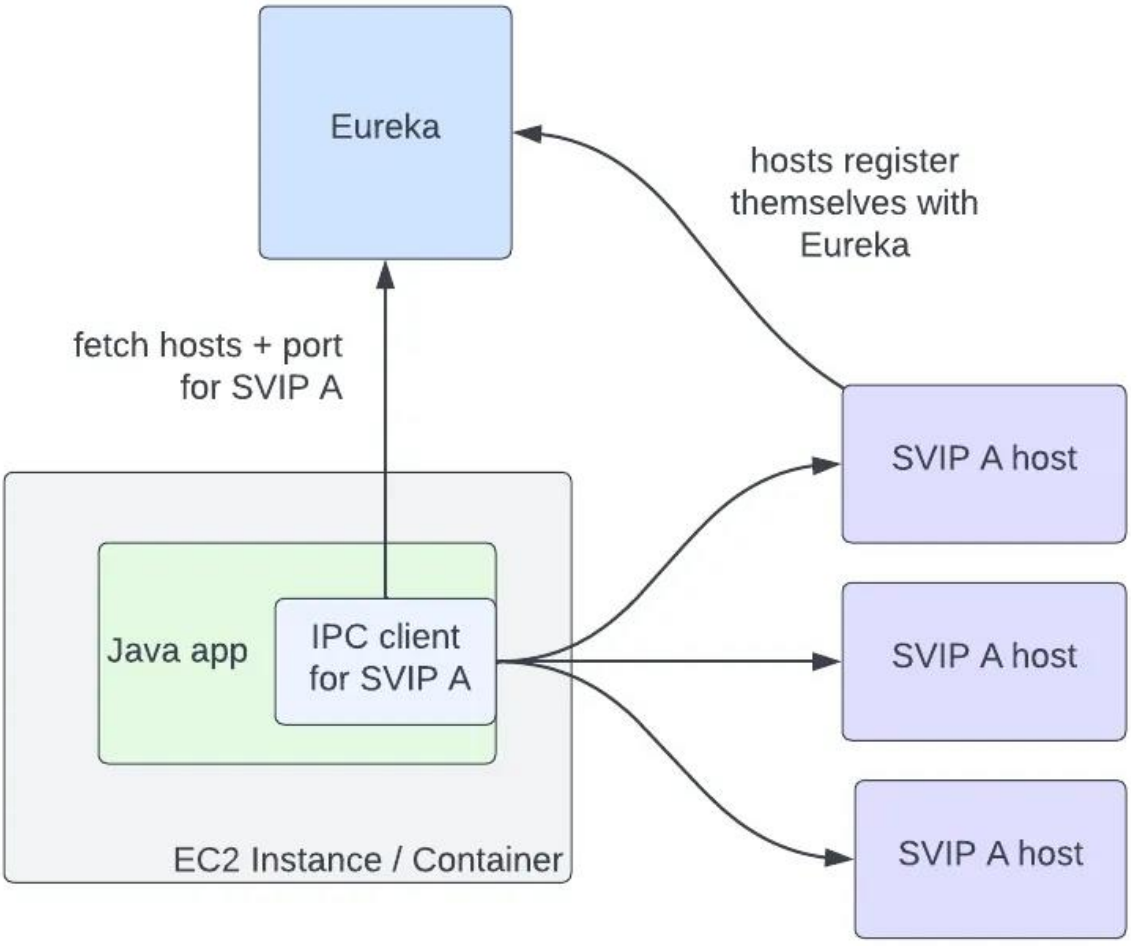
A brief history of IPC at Netflix

For Inter-Process Communication (IPC) between services, we needed the rich feature set that a mid-tier load balancer typically provides. We also needed a solution that addressed the reality of working in the cloud: a highly dynamic environment where nodes are coming up and down, and services need to quickly react to changes and route around failures. To improve availability, we designed systems where components could fail separately and avoid single points of failure.

During these early years in the cloud, we built Eureka for Service Discovery and Ribbon (internally known as NIWS) for IPC. Eureka solved the problem of how services discover what instances to talk to, and Ribbon provided the client-side logic for load-balancing, as well as many other resiliency features.

Eureka and Ribbon presented a simple but powerful interface, which made adopting them easy. In order for a service to talk to another, it needs to know two things: the name of the destination service, and whether or not the traffic should be secure. The abstractions that

Eureka provides for this are Virtual IPs (VIPs) for insecure communication, and Secure VIPs (SVIPs) for secure. A service advertises a VIP name and port to Eureka (eg: *myservice*, port *8080*), or an SVIP name and port (eg: *myservice-secure*, port 8443), or both. IPC clients are instantiated targeting that VIP or SVIP, and the Eureka client code handles the translation of that VIP to a set of IP and port pairs by fetching them from the Eureka server. The client can also optionally enable IPC features like retries or circuit breaking, or stick with a set of reasonable defaults.



The downside is that Eureka is a new single point of failure as the source of truth for what hosts are registered for VIPs. However, if

Eureka goes down, services can continue to communicate with each other, though their host information will become stale over time as instances for a VIP come up and down. The ability to run in a degraded but available state during an outage is still a marked improvement over completely stopping traffic flow.

Why mesh?

we've continued to add more functionality to our IPC clients: features such as adaptive concurrency limiting, circuit breaking, hedging, and fault injection have become standard tools that our engineers reach for to make our system more reliable. Compared to a decade ago, we now support more features, in more languages, in more clients. Keeping feature parity between all of these implementations and ensuring that they all behave the same way is challenging: what we want is a single, well-tested implementation of all of this functionality, so we can make changes and fix bugs in one place.

This is where service mesh comes in: we can centralize IPC features in a single implementation, and keep per-language clients as simple as possible: they only need to know how to talk to the local proxy. Envoy is a great fit for us as the proxy: it's a battle-tested OSS product at use in high scale in the industry, with many critical resiliency features, and good extension points for when we need to extend its functionality. The ability to configure proxies via a central control plane is a killer feature: this allows us to dynamically configure client-side load balancing as if it was a central load balancer, but still avoids a load

balancer as a single point of failure in the service to service request path.

What is a service mesh?

A service mesh is a software layer that handles all communication between services in applications. This layer is composed of containerized microservices. As applications scale and the number of microservices increases, it becomes challenging to monitor the performance of the services. To manage connections between services, a service mesh provides new features like monitoring, logging, tracing, and traffic control. It's independent of each service's code, which allows it to work across network boundaries and with multiple service management systems.

What are the benefits of a service mesh?

A service mesh provides a centralized, dedicated infrastructure layer that handles the intricacies of service-to-service communication within a distributed application. Next, we give several service mesh benefits.

What are the benefits of a service mesh?

A service mesh provides a centralized, dedicated infrastructure layer that handles the intricacies of service-to-service communication within a distributed application. Next, we give several service mesh benefits.

Service discovery

Service meshes provide automated service discovery, which reduces the operational load of managing service endpoints. They use a service registry to dynamically discover and keep track of all services within the mesh. Services can find and communicate with each other seamlessly, regardless of their location or underlying infrastructure. You can quickly scale by deploying new services as required.

Load balancing

Service meshes use various algorithms—such as round-robin, least connections, or weighted load balancing—to distribute requests across multiple service instances intelligently. Load balancing improves resource utilization and ensures high availability and scalability. You can optimize performance and prevent network communication bottlenecks.

Traffic management

Service meshes offer advanced traffic management features, which provide fine-grained control over request routing and traffic behavior. Here are a few examples.

Traffic splitting

You can divide incoming traffic between different service versions or configurations. The mesh directs some traffic to the updated version, which allows for a controlled and gradual rollout of changes. This provides a smooth transition and minimizes the impact of changes.

Request mirroring

You can duplicate traffic to a test or monitoring service for analysis without impacting the primary request flow. When you mirror requests, you gain insights into how the service handles particular requests without affecting the production traffic.

Monitoring

Service meshes offer comprehensive monitoring and observability features to gain insights into your services' health, performance, and behavior. Monitoring also supports troubleshooting and performance optimization. Here are examples of monitoring features you can use:

- Collect metrics like latency, error rates, and resource utilization to analyze overall system performance
- Perform distributed tracing to see requests' complete path and timing across multiple services
- Capture service events in logs for auditing, debugging, and compliance purpose

How does a service mesh work?

A service mesh removes the logic governing service-to-service communication from individual services and abstracts communication to its own infrastructure layer. It uses several network proxies to route and track communication between services.

A proxy acts as an intermediary gateway between your organization’s network and the microservice. All traffic to and from the service is routed through the proxy server. Individual proxies are sometimes called *sidecars*, because they run separately but are logically next to each service. Taken together, the proxies form the service mesh layer.

Data plane

The data plane is the data handling component of a service mesh. It includes all the sidecar proxies and their functions. When a service wants to communicate with another service, the sidecar proxy takes these actions:

1. The sidecar intercepts the request
2. It encapsulates the request in a separate network connection
3. It establishes a secure and encrypted channel between the source and destination proxies

The sidecar proxies handle low-level messaging between services. They also implement features, like circuit breaking and request retries, to enhance resiliency and prevent service degradation. Service mesh functionality—like load balancing, service discovery, and traffic routing—is implemented in the data plane.

Control plane

The control plane acts as the central management and configuration layer of the service mesh.

With the control plane, administrators can define and configure the services within the mesh. For example, they can specify parameters like service endpoints, routing rules, load balancing policies, and security settings. Once the configuration is defined, the control plane distributes the necessary information to the service mesh's data plane.

The proxies use the configuration information to decide how to handle incoming requests. They can also receive configuration changes and adapt their behavior dynamically. You can make real-time changes to the service mesh configuration without service restarts or disruptions.

Service mesh implementations typically include the following capabilities in the control plane:

- Service registry that keeps track of all services within the mesh
- Automatic discovery of new services and removal of inactive services
- Collection and aggregation of telemetry data like metrics, logs, and distributed tracing information

A service mesh is built into an app as an array of network proxies. Proxies are a familiar concept—if you're accessing this webpage from a work computer, there's a good chance you just used one. Here's how it works:

1. As your request for this page went out, it was received by your company's web proxy.
2. After passing the proxy's security measure, it was sent to the server that hosts this page.
3. Next, this page was returned to the proxy and checked against its security measures again.
4. And then it was sent from the proxy to you.

In a service mesh, each service instance is paired with a "sidecar" proxy that runs alongside each service and intercepts all inbound and outbound network traffic. Each sidecar proxy sits alongside a microservice and routes requests to and from other proxies. The proxy handles tasks like traffic routing, load balancing, enforcing security policies, and collecting telemetry data. Instead of communicating directly with one another, services send requests through their sidecar. The sidecars handle inter-service communication. All of this comprises the data plane.

The control plane manages the configuration and policy distribution across the data plane. The control plane also distributes traffic routing rules, manages security certificates between services, configures components to enforce policies, and collects telemetry.

Without a service mesh, each microservice needs to be coded with logic to govern service-to-service communication. This makes communication failures harder to diagnose because the logic that governs interservice communication is hidden within each service.