Level 400

This document focuses on Microservice patterns for Service Discovery, different of patterns in service discovery

Problem Statement:

In case of Monolithic application services calls one another with language-level methods or procedure calls but in microservice architecture services need to call one another as REST or GPRC mechanism.

Microservice application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically. Virtual machines and containers are usually assigned dynamic IP addresses.

How does the services discover the location of another service – the API gateway?

One solution to this obstacle could be to hard code the URLs in to microservices but this approach quickly runs into three problems

1. Change in service URL require code updates, it is also time consuming and depending on the size of the application
2. If application is deployed to cloud, service providers will produce unique URLs that are unpredictable
3. If we have multiple environments, URLs may vary between local, staging and production. Hard cording URLs is not a flexible enough solution to work across the multiple environments your deployment will pass through

Given all the issues, hard coding URLs is not a viable way of microservice communication

Service Discovery is the essential component in a microservice architecture that makes dynamic communication between microservices possible. It is the process that automatically detects, registers, and shares the locations of services in a network.

In addition to helping services locate one another, this communication solution provides a way to perform vital health checks that verify your services and systems are up and running. Also, if a service becomes obsolete and goes offline, it can be deregistered via service discovery.

There are 2 main service discovery patterns to solve these issues

1. Client-side discovery
2. Server-side discovery

Each of these patterns has their own advantages and disadvantages

Client-side Discovery:

When using Client-Side discovery, the client is responsible for determining the network locations of available service instance and load balancing requests across them. The client queries a service registry, which is a database of available service instances. The client then uses a load‑balancing algorithm to select one of the available service instances and makes a request.

Diagram

Description automatically generated

The network location of a service instance is registered with the service registry when it starts up. It is removed from the service registry when the instance goes offline. The service instance’s registration is typically refreshed periodically using a heartbeat mechanism.

Example:

[Netflix OSS](https://netflix.github.io/) is an example of the client‑side discovery pattern, Eureka as their service discovery engine. Spring Cloud provides a Spring Boot implementation of the Eureka server and enables services to register with the Eureka server using an application ID. Each registered service is assigned an instance ID so that each registered service instance is identified based on application ID and instance ID. Eureka clients communicate with Eureka server using Ribbon — Netflix’s client-side load balancer, which keeps a local cache of services received from the service registry. Ribbon periodically updates its’ local cache to match that of the Eureka server.

Reference Implementation git repo: <https://github.com/spring-cloud/spring-cloud-netflix.git>

Client-side discovery has the following benefits:

* Fewer moving parts and network hops compared to Server-Side Discovery
* Since the client knows about the available service instances, it can make intelligent, application-specific load balancing decisions such as using hashing consistently

Client-side discovery also has the following drawbacks:

* This pattern couples the client to Service Registry
* Client-side service discovery logic must be implemented for each programming language/framework used by your application
* The client must make two calls to reach the target microservice

Server-Side Discovery:

The other design pattern to service discovery problem stated above is the server-side discovery pattern. In a server-side service discovery, client doesn’t have to be aware of service registry. The request is made through a router / load balancer. The router / load balancer queries the service registry and routes each request to an available service instance. As with the client-side discovery, service instances are registered and deregistered with Service Registry.

Diagram

Description automatically generated

Example:

In deployment environments, like Kubernetes and Marathon, a proxy is run on each host in the cluster. The proxy acts as a server-side load balancer and routes the request using the host’s IP address and the port assigned to the service. Then, the request is forwarded to an available service instance running in the cluster. Istio is one such example for Service-Side discovery

Istio:

Istio is an open-source platform to connect, manage, and secure microservices. Istio provides an easy way to create a network of deployed services with load balancing, service-to-service authentication, monitoring, and more, without requiring any changes in service code. You add Istio support to services by deploying a special sidecar proxy throughout your environment that intercepts all network communication between microservices, configured and managed using Istio’s control plane functionality.

Service Discovery and **Load Balancing in Istio:**

Service registration: Istio assumes the presence of a service registry to keep track of the pods/VMs of a service in the application. It also assumes that new instances of a service are automatically registered with the service registry and unhealthy instances are automatically removed. Platforms such as Kubernetes, Mesos already provide such functionality for container-based applications. A plethora of solutions exist for VM-based applications.

Service Discovery: Pilot consumes information from the service registry and provides a platform-agnostic service discovery interface. Envoy instances in the mesh perform service discovery and dynamically update their load balancing pools accordingly.

Diagram

Description automatically generated

As illustrated in the figure above, services in the mesh access each other using their DNS names. All HTTP traffic bound to a service is automatically re-routed through Envoy. Envoy distributes the traffic across instances in the load balancing pool. While Envoy supports several [sophisticated load balancing algorithms](https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/load_balancing), Istio currently allows three load balancing modes: round robin, random, and weighted least request.

In addition to load balancing, Envoy periodically checks the health of each instance in the pool. Envoy follows a circuit breaker style pattern to classify instances as unhealthy or healthy based on their failure rates for the health check API call. In other words, when the number of health check failures for a given instance exceeds a pre-specified threshold, it will be ejected from the load balancing pool. Similarly, when the number of health checks that pass exceed a pre-specified threshold, the instance will be added back into the load balancing pool. You can find out more about Envoy’s failure-handling features in [Handling Failures](https://istio.io/v0.5/docs/concepts/traffic-management/handling-failures.html).

Services can actively shed load by responding with a HTTP 503 to a health check. In such an event, the service instance will be immediately removed from the caller’s load balancing pool

Advantages of Server-Side Service Discovery

The server-side discovery type has many benefits.

* It does not involve the client in exploring the details of finding an available service instance.
* It creates an abstract between the client and server-side.
* It removes the need to implement the discovery logic separately for each language and framework that the service clients use.
* It is available for free, with some deployment environments.
* The client needs to make only one call to request services, without having to be involved in looking up available instances.

Disadvantages of Server-Side Service Discovery

The disadvantages of service side discovery can be the following.

* It needs to be set up and managed by you unless already provided by the deployment environment.
* You need to implement the load balancing mechanism for the central server.
* It does not let the client choose a suitable service instance.