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 has to 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.

Example:

A popular example of server-side service discovery is Amazon Web Services (AWS) Elastic Load Balancer (ELB). The ELB is used to balance the load of external traffic from the internet, as well as internal traffic directed to a virtual private cloud (VPC).

The client makes a request through the ELB using its DNS name. The request can be HTTP or TCP. The ELB then performs load balancing of the traffic among EC2 instances or EC2 container service (ECS) containers. The EC2 instances and ECS containers are registered directly with the ELB, without the existence of any separate service registry.

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.