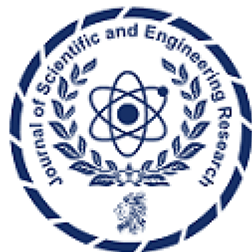
Available online www.jsaer.com

Journal of Scientific and Engineering Research, 2019, 6(12):318-323



Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Service Discovery: Enabling microservices for Mutual Discovery and Communication

Pallavi Priya Patharlagadda

Pallavipriya527.p@gmail.com
United States of America)

Abstract: The tiny, loosely connected distributed services are called microservices. Microservices design emerged as a response to the scalability, independently deployable, and innovation difficulties of monolithic architecture. It enables us to take a large program and divide it into effectively manageable little components, each with a set of assigned duties. It is regarded as the fundamental component of contemporary applications. Let's first examine the necessity of service discovery in microservices and then learn on how to do this in detail.

Keywords: microservices, service discovery in microservices

Problem Statement

When working with microservices, communication plays a crucial role. Each Microservice performs a small task and provides the result. when all these results are combined, we get the final result. So, It is important to know on how the client knows which microservice to contact and get the result. How does each microservice know about the other microservice location or address? To answer this problem, a technique that lets one service use another without knowing its precise location becomes necessary. We'll examine the idea of service discovery in this article**.**

Introduction

Assume that you are creating microservices. Your organization has implemented the microservices architecture; you have Address Service, Employee Service, Course Service, Student Service, and so on. There are hundreds or perhaps thousands of spring boot programs that you may have, all of which have been installed on various servers. As things stand right now, how will these servers communicate with one another when the Course Service, for example, needs to connect to the Address Service and the Student Service wants to connect to the Course Service and obtain some course-related data? All these servers will talk to one another using the REST API; all we must do is make a REST call. The very difficult thing is that, for a server to establish a connection with another server, it must first determine the IP address and port number of the server that is hosting the specific application.

Managing something where there are thousands and thousands of apps will not be an easy task. How are you going to handle the server IP? How are you going to keep their port number? Every server must establish a connection with another server to retrieve data from it. Therefore, managing all IP addresses and server ports becomes extremely important when managing thousands of servers and dividing a single application into thousands of modules that are deployed across various servers. Do you not believe that controlling the IP and the server URL will be essential?

Before we do a deep dive into what service discovery is all about, let us learn about monolithic & Microservice architecture.

Monolithic Architecture: The classic model of a software program is a monolithic architecture, which is created as a single, cohesive entity that is a separate program. It is common to associate the word "monolith" with something massive and glacial, which is not too unlike the reality of a monolithic software design architecture. A monolithic architecture unites all business concerns into a single, vast computer network with a single code base. This kind of application involves upgrading the complete stack to make changes, which includes generating and deploying an updated service-side interface and gaining access to the code base. Updates become cumbersome and limited as a result. Monoliths can be useful early in a project's lifecycle because they simplify code maintenance, cognitive burden, and deployment. This makes it possible for the monolith to release everything at once.

Advantages of a monolithic architecture:

● Simple deployment: It is simpler to deploy when there is only one executable file or directory.

● Development: It's easier to create an application with a single code base.

● Performance: Using microservices, several APIs may do the same task that a single API can do in a centralized code base and repository.

● Simplified testing: End-to-end testing for monolithic applications may be completed more quickly than for dispersed applications since they consist of a single, centralized unit.

● Simple debugging: It's simpler to track a request and identify a problem when all the code is in one location.

Disadvantages of a monolithic architecture:

● Decreased development speed – The complexity and speed of development increase with a large, monolithic application.

● Scalability: Individual components are not scalable.

● Dependability: If a module contains a bug, it may impact the availability of the program.

● Adoption barrier for technology: Modifications to the language or framework have an impact on the entire program, which makes them costly and time-consuming.

●Lack of Flexibility: The technologies included in a monolith already place restrictions on it.

● Deployment: Redeploying a monolithic program necessitates doing so after making minor adjustments.

Microservices Architecture:

The formal expression of microservices' capabilities through business-oriented APIs sets them apart from monolithic architectures. They include a fundamental business function, and because the interface is limited to business terminology, the execution of the service—which can entail linkages with record systems—is concealed from view. Services are inadvertently positioned as flexible resources that may be used in a variety of settings by being viewed as corporate assets. It is possible to reuse the same service across several business channels, digital touchpoints, and business processes. The loose coupling approach is used to reduce dependency between services and their customers. using standardization on contracts communicated using business-oriented APIs, service implementation changes do not affect customers. This makes it possible for service providers to alter the systems of record or service compositions that may be located behind the interface, change their implementation, and replace them without affecting other services.

Advantages of Microservices Architecture:

● Agility: Encourage small teams that deploy regularly to operate in an agile manner.

● Flexible scaling: Enables the quick deployment of new instances of a microservice to the corresponding cluster to alleviate pressure if it hits its load capacity. With clients dispersed over several instances, we are now multi-tenant and stateless. We can now handle significantly bigger instance sizes.

● Continuous deployment: Our release cycles have accelerated to become more frequent. We can now provide updates around twice or three times a day instead of just once a week as we used to.

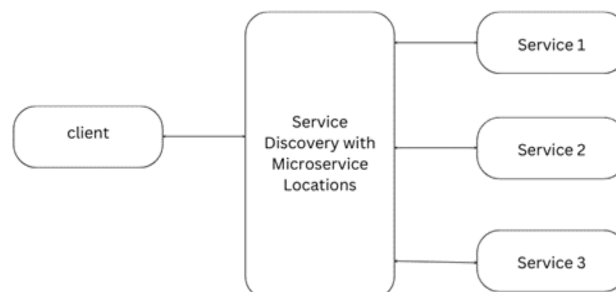
● Extremely testable and maintainable: Teams can try out new features and go back if anything doesn't work. This speeds up the time to market for new features and simplifies code updates. Furthermore, it is simple to identify and correct errors and problems in certain services.

● Independently deployable: Since microservices are discrete components, deploying individual functionalities on their own is quick and simple.

● Technological adaptability - Teams may freely choose the technologies they want thanks to microservice designs.

● High reliability: You don't have to worry about taking down the entire application to implement modifications for a particular service.
● Teams that use microservices are happier –because they can create and deploy independently without having to wait weeks for a pull request to be accepted, teams using Atlassian microservices are far more autonomous.
Disadvantages of microservices:
● Development sprawl: Because microservices are more dispersed and multi-team-created than monolithic architectures, they are more complicated. Poor operational performance and slower development pace are the outcomes of improperly managed development sprawl.
● Exponential infrastructure costs: Test suites, deployment plans, hosting infrastructure, monitoring tools, and other expenses may be incurred separately for every new microservice.
● Increased organizational overhead: To manage changes and interfaces, teams must collaborate and communicate at an even higher level.
● Difficulties in debugging: Since every microservice has a unique collection of logs, debugging becomes more difficult. To make debugging even more difficult, a single business process might operate on several computers.
● Absence of standardization: A multitude of languages, logging conventions, and monitoring systems may arise in the absence of a single platform.
● Absence of distinct ownership –As more services are made available, so do the teams that manage them. Knowing what resources are available to a team and who to ask for help might get harder with time.

Service Discovery
The process of automatically determining which service instances to use to answer a certain query is known as service discovery. Because Service Discovery can find a network on its own, a laborious configuration setup procedure is not necessary. Through the use of a shared language on the network, service discovery enables automatic device or service connections. (For example, service discovery on AWS and Kubernetes).
Why do you need service discovery?
Working with service instances that have changeable locations/ip addresses is a necessary part of developing a microservices application. Runtime modifications to these instances may also be required in response to failures, scaling, and service updates. The services or the end users that rely on these instances in this case have to know.
Assume you are developing code that uses a REST API to call a service. To submit a request, the code requires the service instance's IP address and port. It is no longer necessary to look for these locations every time if your application is hardware-based.
In a microservice design, on the other hand, the number of instances will differ and their locations won't be specified in a configuration file. As a result, it is challenging to determine the total number of services at any one time. To find service instances in dynamically allocated network locations in a cloud-based microservices location, you require service discovery.
● It facilitates load adaptation and appropriate distribution for all instances. There are three parts to service discovery.
● A service provider is a company that offers services via a network.
● A service registry refers to a database that holds the locations of every service instance that is currently accessible.
● A service consumer is the person who connects with the service instance and gets the location of the service provider from the service registry.



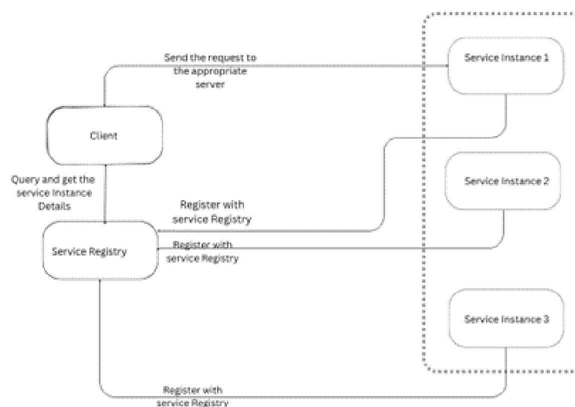
Types of Service Discovery:

Service discovery comes in two forms: client-side and server-side.

Client-side service discovery:

In this kind of service discovery, finding a service provider requires the service client or consumer to look through the service register. Next, using a load balancing mechanism, the client chooses an appropriate and low-cost service instance to submit a request for.

According to this design, as soon as the service launches, the location of the service instance is registered with the service registry. Following the termination of the service instance, the location data is removed. This update happens regularly using a heartbeat mechanism.



As long as the user is aware of which service instances are accessible and able to handle the demand, you may make informed judgments about load balancing with this kind. It is simple to comprehend and provides the customer with the ability to locate accessible service instances on the network.

The first inquiry from the client is sent to a Discovery Server, a central server that serves as a directory for all of the accessible service instances. Additionally, it gives the service instances an additional degree of abstraction.

An API gateway may be used to hide the service in client-side service discovery. If not, the client is in charge of putting features like cross-cutting, balancing, and authentication into practice.

Netflix OSS, whose service registry is Netflix Eureka, is a well-known illustration of client-side discovery. It offers a REST API for managing available instance queries and service instance integration. For load balancing purposes, Netflix Ribbon uses Eureka as an IPC client to handle requests to available service instances.

Advantages of Client-Side Service Discovery:

Using the client-side service discovery approach has several benefits. Other than the service registry, it is easy to use and has no moving components. Additionally, it gives the client authority over load-balancing choices.

- The client is capable of making deft choices per specifications.
- Hashing is one example of an application-specific load-balancing technique.
- It is easier to manage and less complex.
- It is not required to use a load balancer or router to redirect the client application or request.
- It saves the superfluous stages in the middle and produces a speedy output.

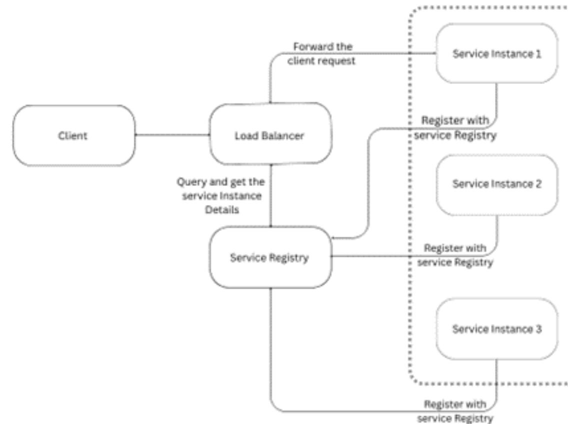
Disadvantages of Client-Side Service Discovery:

- Because it ties the client with the service registry, each programming language and framework that the service clients use requires the implementation of client-side discovery logic.
- Because the microservices design uses a variety of frameworks, tools, and technologies across several languages, application administration becomes more difficult.
- Reaching the desired microservice requires the client to make two calls.

Server-Side Service Discovery:

The consumer or client does not need to be aware of the service register to discover services. A router receives the requests and uses the service registry to do its search. The router passes the request and completes the task when it discovers a legitimate service instance that is accessible.

The client does not need to be concerned with load balancing or locating an appropriate service instance while using this pattern. Rather, the API gateway does this task by choosing the appropriate endpoint for a request that originates from the client.



In essence, the task of correctly accepting and forwarding the client server's request rests with the server side. The process does this without requiring any manual involvement from the client by keeping track of service locations and locating the required service. The load balancer processes each client request identically. When the service launches, the service instances register with the service registry, just like client-side discovery does. Deregistration of the service instance occurs upon termination of the service.

A well-known instance of server-side service discovery is the Elastic Load Balancer (ELB) offered by Amazon Web Services (AWS). Virtual private cloud (VPC) traffic and internal traffic destined for the internet are both balanced by the ELB.

Using its DNS name, the client sends a request through the ELB. TCP or HTTP requests are both possible. The traffic between EC2 instances or EC2 container service (ECS) containers is then load-balanced by the ELB. Without the need for a different service registry, the ELB immediately registers the EC2 instances and ECS containers.

Certain deployment technologies, such as Marathon and Kubernetes, require that a proxy be executed on every computer in the cluster. Using the host's IP address and the port designated for the service, the proxy routes the request as a server-side load balancer. Next, a service instance that is executing in the cluster and is available receives the request.

Advantages of Server-Side Service Discovery:

- The customer is not involved in delving into the specifics of locating an available service instance.
- Between the client side and server side, it establishes an abstraction.
- Because of this, the service clients may stop implementing the discovery logic independently for every language and framework they use.
- With some deployment environments, it is free to use.
- To request services, the customer simply needs to provide one call and is not required to search for instances that are accessible.

Disadvantages of Server-Side Service Discovery:

- If it isn't already offered by the deployment environment, you must put it up and maintain it.
- For the central server, you must put the load balancing system into place.
- The client is unable to select an appropriate service instance.

The Service Registry:

As was previously indicated, a crucial element of service discovery is the service registry. It is a database that has all of the service instances' locations. As a result, ongoing maintenance and updates are required.

Users may cache network locations from the service registry for later use, but they shouldn't rely on it too much since in this dynamic environment, the information soon ages and makes it impossible for the user to find service instances.

A cluster of servers that employ replication protocol to maintain consistency makes up a service registry. Consider HashiCorp's Consul as an example. Consul is mostly a utility for finding and setting up services. Clients may register and find service instances using its API, and it also does frequent health checks to guarantee service availability.

Consul's consolidated service registry facilitates the discovery of services by consolidating location data into a single register and including features like:

- Regulating the dispersed data plane to offer a dependable and scalable service mesh.
- Supplying a current list of all active services.
- Reducing human interaction by automating the configuration of centralized network middleware.
- Implementing automated certificate lifecycle management for third-party Certificate Authorities.
- Allowing services and their health status to be visible.
- Provide consistent support for various runtime systems and workload kinds

Conclusion

A cluster of dynamically changing service instances is present in a microservice application. These situations require a system to find or communicate since their network locations are changeable. Thus, to submit a request, a client needs service discovery.

Together with the administration API and query API, the service registry offers a database of accessible service instances. This allows for the discovery, registration, and deregistration of services according to use. The sort of discovery you want to utilize will depend on whether you want to automate or build up the environment: client-side or server-side.

References

[1]. https://microservices.io/patterns/server-side-discovery.html
[2]. https://www.hashicorp.com/resources/modern-service-networking-cloud-microservices
[3]. https://hackernoon.com/understand-service-discovery-in-microservice-c323e78f47fd
[4]. https://spring.io/blog/2015/01/20/microservice-registration-and-discovery-with-spring-cloud-and-netflix-s-eureka
[5]. https://medium.com/@jamesemyn/service-discovery-in-microsrvice-cbd54afb94f3
[6]. https://walkingtree.tech/service-discovery-microservices/
[7]. https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/
[8]. https://dzone.com/articles/getting-started-with-microservices-2
[9]. https://www.linkedin.com/pulse/service-discovery-micro-service-architecture-ananda-verma/
[10]. https://medium.com/@maxy_ermayank/service-registration-and-discovery-configuration-management-dffb15fc08a7
[11]. https://pusher.com/tutorials/service-discovery-microservices/#building-the-nodejs-service
[12]. https://callistaenterprise.se/blogg/teknik/2017/04/24/go-blog-series-part7/
[13]. https://www.codeprimers.com/service-discovery-in-microservice-architecture/
[14]. https://www.codeproject.com/Articles/1248381/Microservices-Service-Discovery
[15]. https://dzone.com/articles/microservices-architectures-what-is-service-discov
[16]. https://www.incentergy.de/blog/2018/09/10/microservices-service-discovery-and-configuration-with-dns/
[17]. https://blogs.infomentum.com/microservices-everything-you-need-to-know-part-4