# System Design: What is Service Discovery?

How services find each other

ASHISH PRATAP SINGH

DEC 05, 2024



Back when applications ran on a **single server**, life was simple.

Today's modern applications are far more complex, consisting of **dozens or even hundreds of services**, each with multiple instances that scale up and down dynamically.

This makes it harder for services to efficiently **find and communicate** with each other across networks.

That's where **Service Discovery** comes into play.

In this article, we'll dive into what service discovery is, why it's important, how it works, the different types (client and server side discovery), and best practices for

implementing it effectively.

---

# 1. What is Service Discovery?

**Service discovery** is a mechanism that allows services in a distributed system to **find and communicate** with each other dynamically.

It hides the complex details of where services are located, so they can interact without knowing each other's exact network spots.

Service discovery registers and maintains a record of all your services in a **service registry**. This service registry acts as a single source of truth that allows your services to query and communicate with each other.

**Example service registry record of a service:**


A service registry typically stores:

- **Basic Details:** Service name, IP, port, and status.
- **Metadata:** Version, environment, region, tags, etc.
- **Health Information:** Health status, last health check.
- **Load Balancing Info:** Weights, priorities.
- **Secure Communication:** Protocols, certificates.

This abstraction is important in environments where services are constantly being added, removed, or scaled.

# 2. Why is Service Discovery Important?

Think about a massive system like **Netflix**, with hundreds of microservices working together. Hardcoding the locations of these services isn't scalable. If a service moves to a new server or scales dynamically, it could break the entire system.

**Service discovery** solves this by dynamically and reliably enabling services to locate and communicate with one another.

Here are its key benefits:

- **Reduced Manual Configuration:** Services can automatically discover and connect to each other, eliminating the need for manual configuration and hardcoding of network locations.
- **Improved Scalability:** As new service instances are added or removed, service discovery ensures that other services can seamlessly adapt to the changing environment.
- **Fault Tolerance:** Service discovery often include health checks, enabling systems to automatically reroute traffic away from failing service instances.
- **Simplified Management:** Having a central registry of services makes it easier to monitor, manage, and troubleshoot the entire system.

# 3. Service Registration Options

Service registration is the process where a service announces its availability to a **service registry**, making it discoverable by other services.

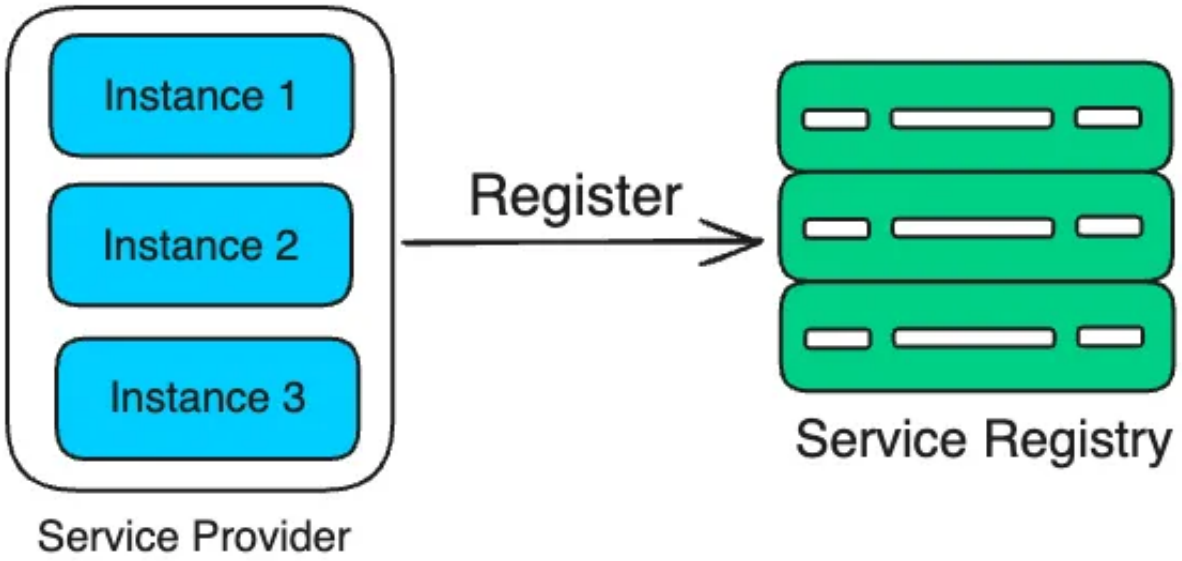
The method of registration can vary depending on the architecture, tools, and deployment environment.

Here are the most common **service registration options**:

## 3.1. Manual Registration

In manual registration, service details are added to the registry manually by a developer or operator. This approach is simple but not suitable for dynamic systems where services scale or move frequently.

## 3.2. Self-Registration




Visualized using Multiplayer

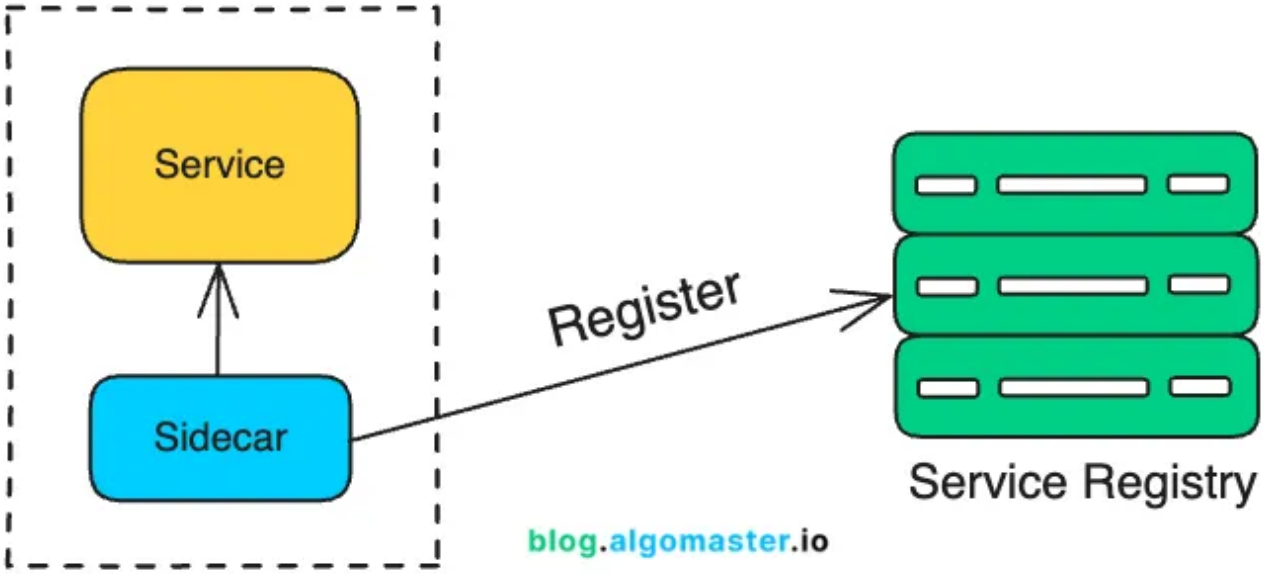
In self-registration, the service is responsible for registering itself with the service registry when it starts. The service includes logic to interact with the registry, such as

sending API requests to register its details.

**How it works:**

1. When a service or an instance starts, it retrieves its own network information (e.g., IP address, port).
2. It sends a registration request to the service registry (e.g., via HTTP or gRPC).
3. To ensure the registry has up-to-date information, the service may periodically send heartbeat signals to confirm it is active and healthy.

## 3.3. Third-Party Registration (Sidecar Pattern)




Visualized using Multiplayer

In third-party registration, an external agent or "sidecar" process handles service registration. The service itself does not directly interact with the registry. Instead, the sidecar detects the service and registers it on its behalf.

**How it works:**

1. The sidecar runs alongside the service (e.g., in the same container or on the same host).
2. The sidecar detects when the service starts and gathers its network details.

3. It sends the registration request to the service registry.

## 3.4. Automatic Registration by Orchestrators

In modern orchestrated environments like **Kubernetes**, service registration happens automatically. The orchestration platform manages the lifecycle of services and updates the service registry as services start, stop, or scale.

**How it works:**

1. The orchestrator (e.g., Kubernetes) detects when a service or container is deployed.
2. It assigns the service an IP address and port.
3. It registers the service automatically with its built-in service discovery mechanism (e.g., Kubernetes DNS).

## 3.5. Configuration Management Systems

Some systems use configuration management tools (e.g., Chef, Puppet, Ansible) to register services. These tools manage the service lifecycle and update the service registry whenever services are added or removed.

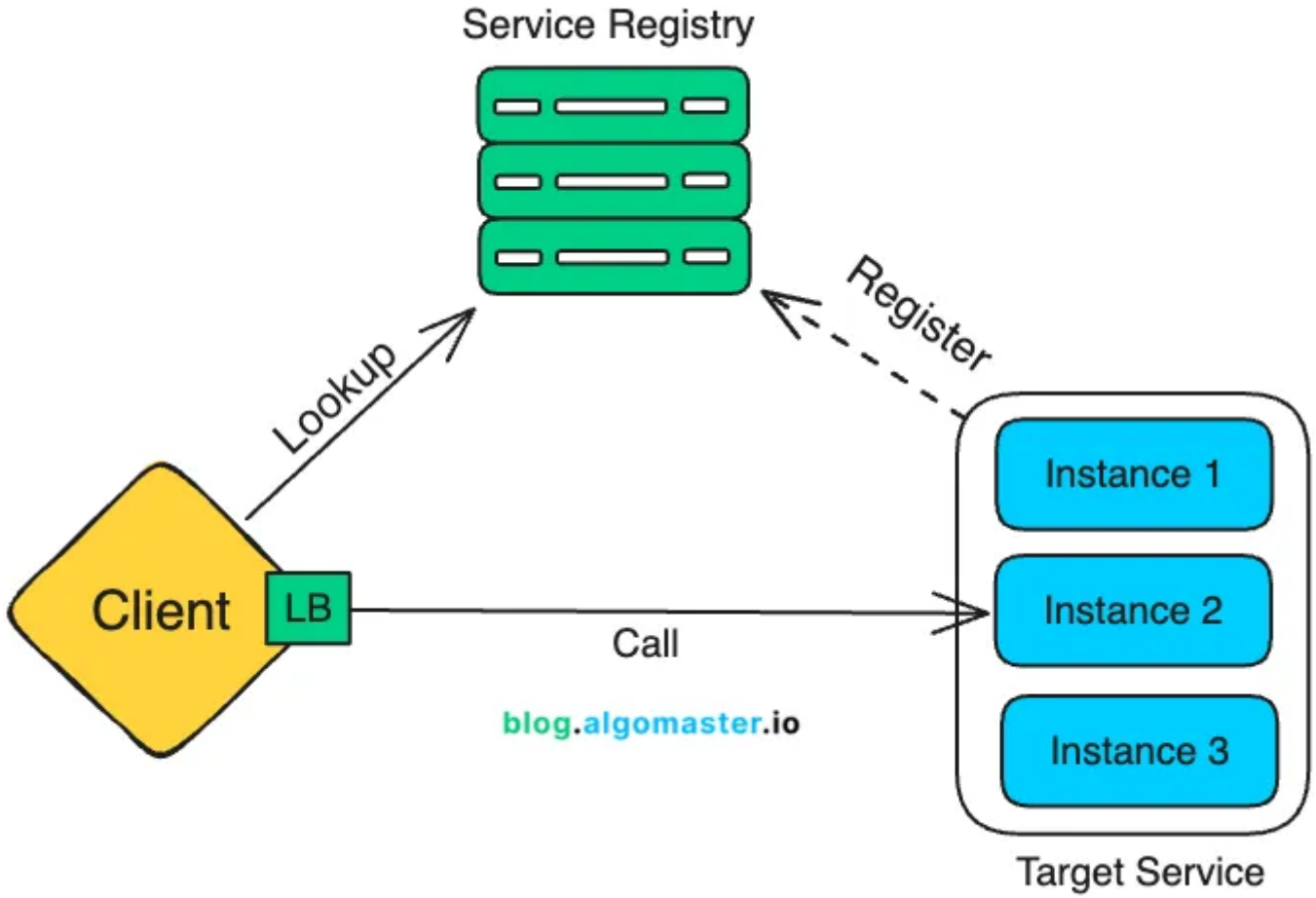
# 4. Types of Service Discovery

There are two primary types of service discovery: client-side discovery and server-side discovery.

## 4.1. Client-Side Discovery

In this model, the responsibility for discovering and connecting to a service lies entirely with the client.

### How it works:




Visualized using Multiplayer

1. **Service Registration:** Services (e.g., `UserService`, `PaymentService`) register themselves with a centralized **service registry**.
    a. They provide their network details (IP address and port) along with metadata

like service health or version.

2. **Client Queries the Registry:** The client (a microservice or API gateway) sends a request to the **service registry** to find the instances of a target service (e.g., `PaymentService`).
    a. The registry responds with a list of available instances, including their IP addresses and ports.
3. **Client Routes the Request:** Based on the information retrieved, the client selects one of the service instances (often using a load balancing algorithm) and connects directly to it.

The client maintains control over how requests are routed, such as distributing traffic evenly across instances or prioritizing the closest instance.

### Example Workflow

Let's consider a real-world example of a food delivery app:

- A **Payment Service** has three instances running on different servers.
- When the **Order Service** needs to process a payment, it queries the **service registry** for the location of the `Payment Service`.
- The service registry responds with a list of available instances (e.g., `IP1:Port1`,

`IP2:Port2`, `IP3:Port3`).

- The **Order Service** chooses an instance (e.g., `IP1:Port1`) and sends the payment request directly to it.

### Advantages:

- Simple to implement and understand.
- Reduces the load on a central load balancer.

### Disadvantages:

- Consumers need to implement discovery logic.
- Changes in the registry protocol require changes in clients.

> **Example:** Netflix's open-source library, **Eureka**, is a popular tool for client-side service discovery.

## 4.2. Server-Side Discovery

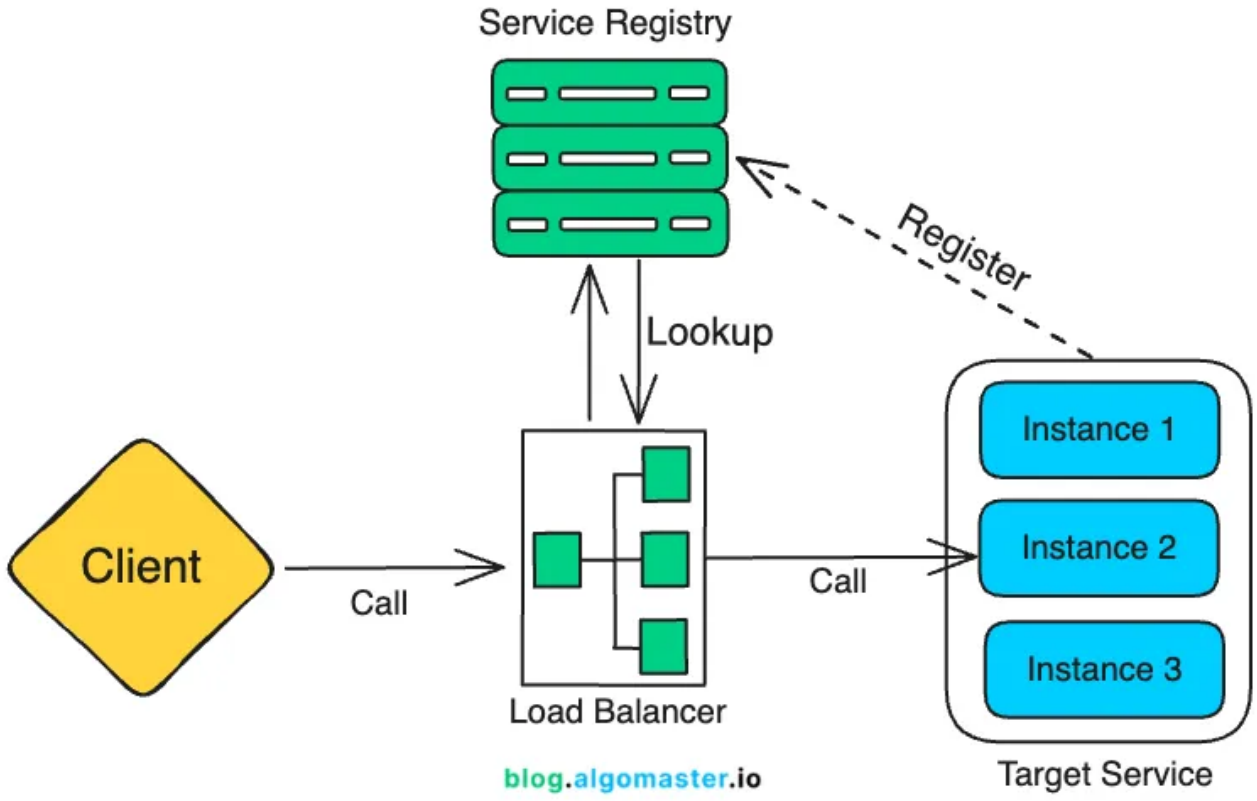
In this model, the client delegates the responsibility of discovering and routing requests to a specific service instance to a **centralized server or load balancer**.

Unlike client-side discovery, the client does not need to query the service registry

directly or perform any load balancing itself.

Instead, the client simply sends a request to a central server (load balancer or api gateway), which handles the rest.

### How it works:




Visualized using Multiplayer

1. **Service Registration:** Services register themselves with a centralized **service registry**, similar to client-side discovery.
    - The service registry keeps track of all service instances, their IP addresses, ports, and metadata.
2. **Client Sends Request:** The client sends a request to a **load balancer** or **API gateway**, specifying the service it wants to communicate with (e.g., `payment-service`).
    - The client does not query the service registry or know the specific location of the service instances.
3. **Server Queries the Service Registry:** The load balancer or gateway queries the service registry to find available instances of the requested service.
4. **Routing:** The load balancer selects a suitable service instance (based on factors like load, proximity, or health) and routes the client's request to that instance.
5. **Response:** The service instance processes the request and sends the response back to the client via the load balancer or gateway.

## Example Workflow

Let's take an example of an e-commerce platform with microservices for "Order Management" and "Payment Processing."

1. **Registration:** The `PaymentService` registers two instances with the service registry:
    - Instance 1: `IP1:8080`
    - Instance 2: `IP2:8081`
2. **Client Request:** The `OrderService` sends a request to the **load balancer** or API gateway, specifying the `PaymentService`.
3. **Discovery and Routing:** The load balancer queries the service registry and retrieves the list of available `PaymentService` instances.
    - It selects one instance (e.g., `IP1:8080`) and routes the request to it.
4. **Processing and Response:** The selected instance of `PaymentService` processes the request and sends the response back to the `OrderService` via the load balancer.

### Advantages:

- Centralizes discovery logic, reducing the complexity for consumers.
- Easier to manage and update discovery protocols.

### Disadvantages:

- Introduces an additional network hop.
- The load balancer can become a single point of failure.

> **Example**: AWS Elastic Load Balancer (ELB) integrates with the AWS service registry for server-side discovery.

# 5. Best Practices for Implementing Service Discovery

1. **Choose the Right Model:** Use client-side discovery for custom load balancing and server-side for centralized routing.
2. **Ensure High Availability:** Replicate the service registry and test failover scenarios to prevent downtime. Deploy multiple instances of the service registry to avoid single points of failure.
3. **Automate Registration:** Use self-registration, sidecars, or orchestration tools for dynamic environments. Ensure proper deregistration of stale services.
4. **Use Health Checks:** Monitor service health and remove failing instances automatically.
5. **Follow Naming Conventions:** Use clear, unique service names with versioning to

avoid conflicts (e.g., `payment-service-v1`).

6. **Caching**: Use caching mechanisms to reduce the load on the service registry and improve discovery performance.
7. **Scalability**: Ensure that the service discovery system can scale with the growth of your services.

# 6. Conclusion

Service discovery may not be the most glamorous aspect of distributed systems, but it is undoubtedly one of the most essential.

Think of service discovery as the address book of your microservices architecture. Without it, scaling and maintaining distributed systems would be chaotic.

It serves as the backbone that enables the seamless communication and coordination between services, allowing complex applications to function reliably and efficiently.

---

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

If you have any questions or suggestions, leave a comment.

This post is public so feel free to share it.

---

**P.S.** If you're enjoying this newsletter and want to get even more value, consider becoming a **paid subscriber**.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all **premium courses** on **algomaster.io**.

**There are group discounts, gift options, and referral bonuses available.**

---

Checkout my **Youtube channel** for more in-depth content.

Follow me on **LinkedIn**, **X** and **Medium** to stay updated.

Checkout my **GitHub repositories** for free interview preparation resources.

I hope you have a lovely day!

See you soon,
Ashish

---

131 Likes • 4 Restacks

← Previous

Next →

## Discussion about this post

Comments Restacks

Write a comment...

Ashwin 5 Dec 2024

Can zookeeper be considered a tool for service registry as well?

LIKE (2) REPLY SHARE