BLOG TECH Chris Richardson · October 12, 2015

Service Discovery in a Microservices Architecture

microservices, *service discovery*, *etcd*, *DevOps*, *Netflix*, *Amazon Web Services (AWS)*, *ZooKeeper*, *Consul*

Editor *– This seven-part series of articles is now complete:*

1. *Introduction to Microservices*
2. *Building Microservices: Using an API Gateway*
3. *Building Microservices: Inter-Process Communication in a Microservices Architecture*
4. *Service Discovery in a Microservices Architecture (this article)*
5. *Event-Driven Data Management for Microservices*
6. *Choosing a Microservices Deployment Strategy*
7. *Refactoring a Monolith into Microservices*

You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook – Microservices: From Design to Deployment.

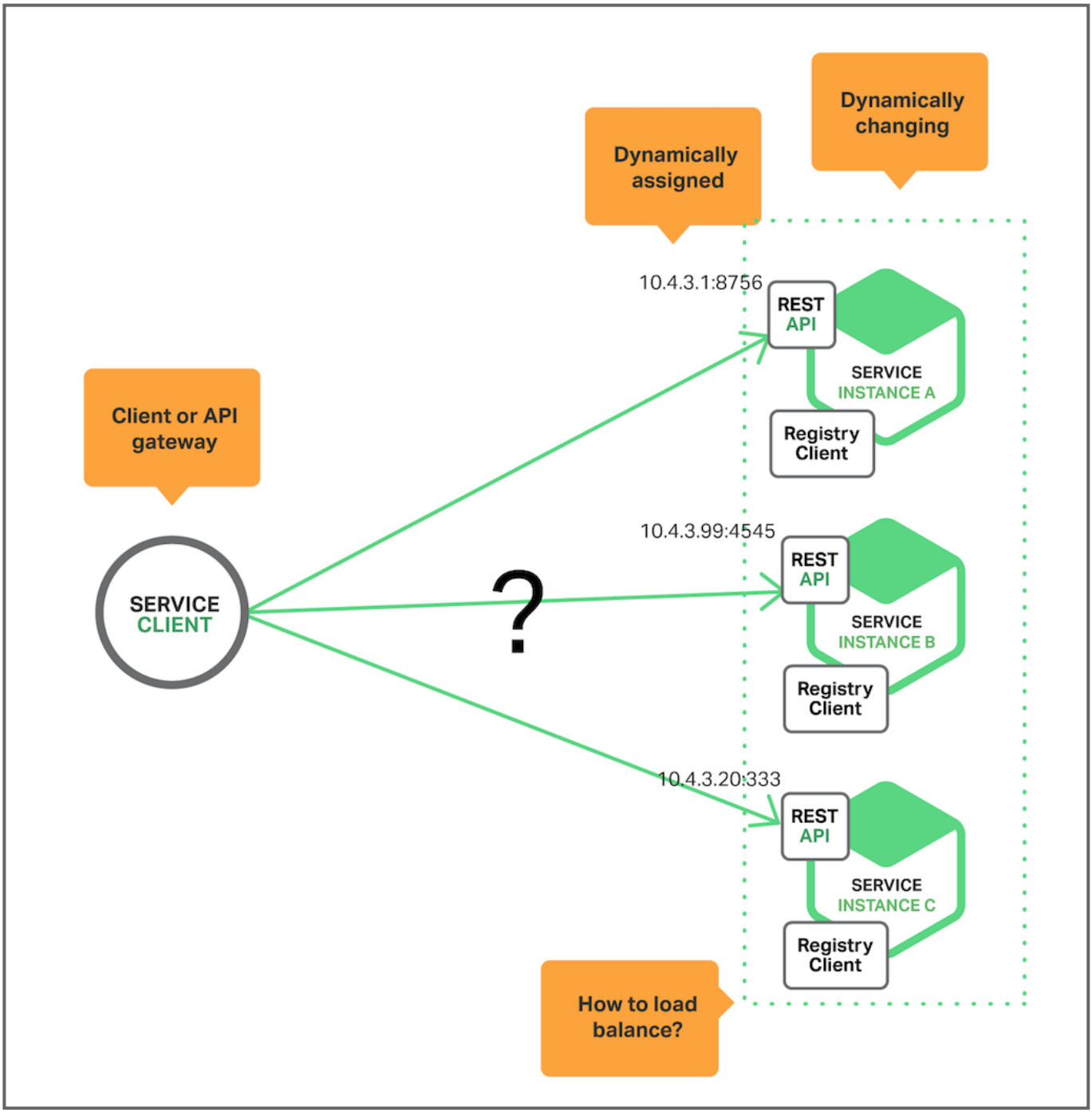
This is the fourth article in our series about building applications with microservices. The first article introduces the Microservices Architecture pattern and discussed the benefits and drawbacks of using microservices. The second and third articles in the series describe different aspects of communication within a microservices architecture. In this article, we explore the closely related problem of service discovery.

Why Use Service Discovery?

Let's imagine that you are writing some code that invokes a service that has a REST API or Thrift API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network

locations of service instances are relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated.

In a modern, cloud-based microservices application, however, this is a much more difficult problem to solve as shown in the following diagram.



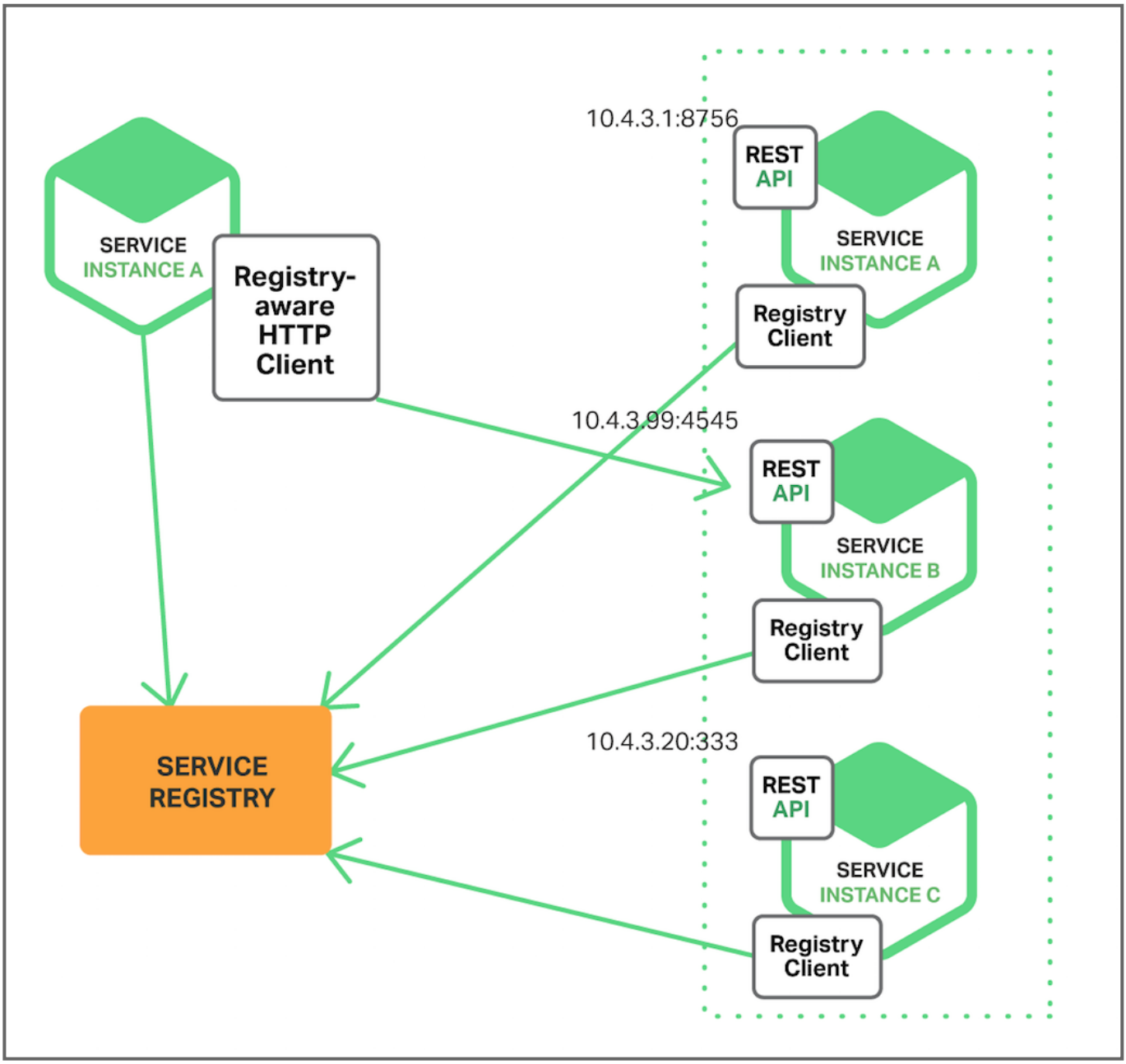
Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism.

There are two main service discovery patterns: client-side discovery and server-side discovery. Let's first look at client-side discovery.

The Client-Side Discovery Pattern

When using client-side discovery, the client is responsible for determining the network locations of available service instances 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.

The following diagram shows the structure of this pattern.



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 terminates. The service instance's registration is typically refreshed periodically using a heartbeat mechanism.

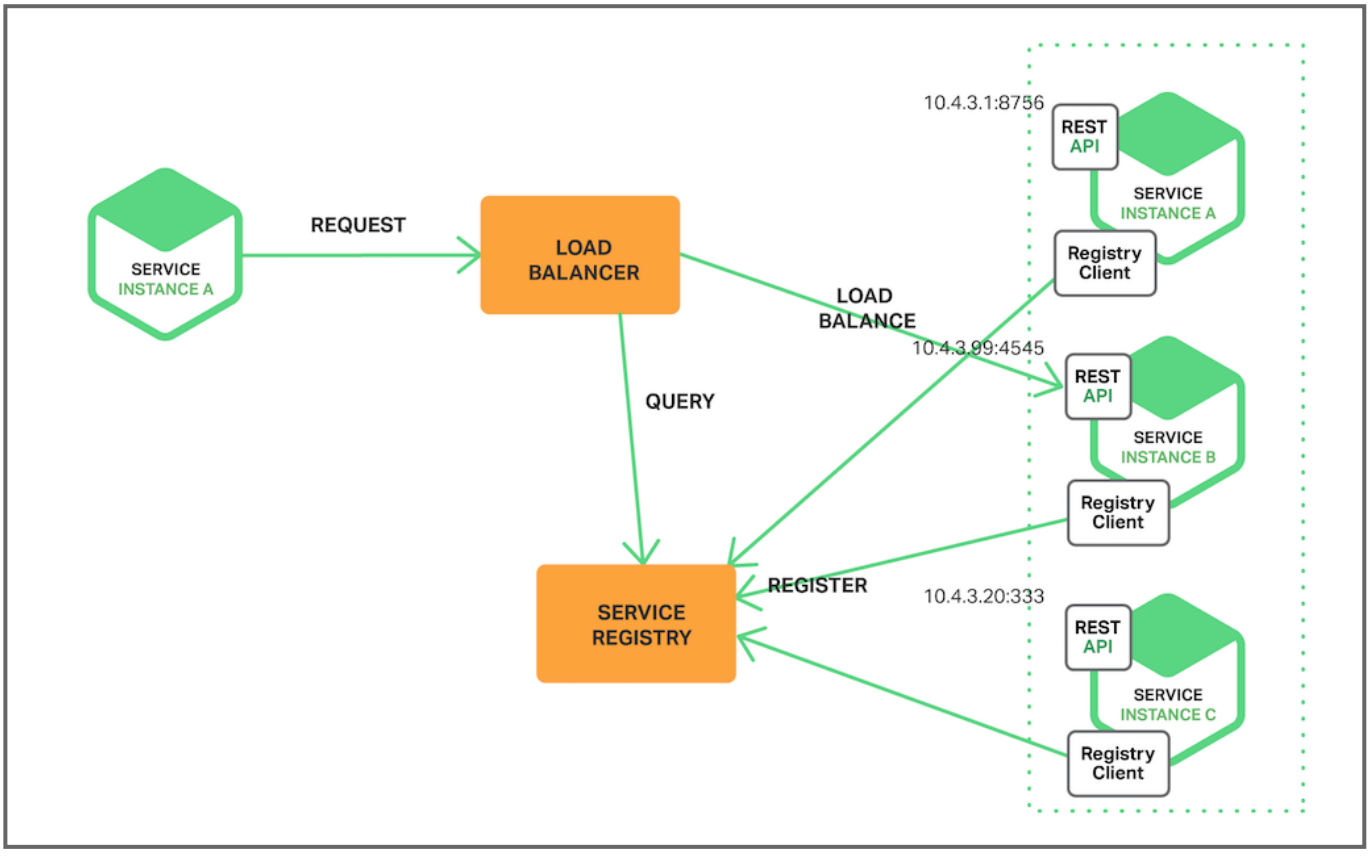
Netflix OSS provides a great example of the client-side discovery pattern. Netflix Eureka is a service registry. It provides a REST API for managing service-instance registration and for querying available instances. Netflix Ribbon is an IPC client that works with Eureka to load balance requests across the available service instances. We will discuss Eureka in more depth later in this article.

The client-side discovery pattern has a variety of benefits and drawbacks. This pattern is relatively straightforward and, except for the service registry, there are no other moving parts. Also, since the client knows about the available services instances, it can make intelligent, application-specific load-balancing decisions such as using hashing consistently. One significant drawback of this pattern is that it couples the client with the service registry. You must implement client-side service discovery logic for each programming language and framework used by your service clients.

Now that we have looked at client-side discovery, let's take a look at server-side discovery.

The Server-Side Discovery Pattern

The other approach to service discovery is the server-side discovery pattern. The following diagram shows the structure of this pattern.



The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

The AWS Elastic Load Balancer (ELB) is an example of a server-side discovery router. An ELB is commonly used to load balance external traffic from the Internet. However, you can also use an ELB to load balance traffic that is internal to a virtual private cloud (VPC). A client makes requests (HTTP or TCP) via the ELB using its DNS name. The ELB load balances the traffic among a set of registered Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers. There isn't a separate service registry. Instead, EC2 instances and ECS containers are registered with the ELB itself.

HTTP servers and load balancers such as NGINX Plus and NGINX can also be used as a server-side discovery load balancer. For example, this blog post describes using Consul Template to dynamically reconfigure NGINX reverse proxying. Consul Template is a tool that periodically regenerates arbitrary configuration files from configuration data stored in the Consul service registry. It runs an arbitrary shell command whenever the files change. In the example described by the blog post, Consul Template generates an **nginx.conf** file, which configures the reverse proxying, and then runs a command that tells NGINX to reload the configuration. A more sophisticated implementation could dynamically reconfigure NGINX Plus using either its HTTP API or DNS.

Some deployment environments such as Kubernetes and Marathon run a proxy on each host in the cluster. The proxy plays the role of a server-side discovery load balancer. In order to make a request to a service, a client routes the request via the proxy using the host's IP address and the service's assigned port. The proxy then transparently forwards the request to an available service instance running somewhere in the cluster.

The server-side discovery pattern has several benefits and drawbacks. One great benefit of this pattern is that details of discovery are abstracted away from the client. Clients simply make requests to the load balancer. This eliminates the need to implement discovery logic for each programming language and framework used by your service clients. Also, as mentioned above, some deployment environments provide this functionality for free. This pattern also has some drawbacks, however. Unless the load balancer is provided by the deployment environment, it is yet another highly available system component that you need to set up and manage.

The Service Registry

The service registry is a key part of service discovery. It is a database containing the network locations of service instances. A service registry needs to be highly available and up to date. Clients can cache network locations obtained from the service registry. However, that information eventually becomes out of date and clients become unable to discover service instances. Consequently, a service registry consists of a cluster of servers that use a replication protocol to maintain consistency.

As mentioned earlier, Netflix Eureka is good example of a service registry. It provides a REST API for registering and querying service instances. A service instance registers its network location using a `POST` request. Every 30 seconds it must refresh its registration using a `PUT` request. A registration is removed by either using an HTTP `DELETE` request or by the instance registration timing out. As you might expect, a client can retrieve the registered service instances by using an HTTP `GET` request.

Netflix achieves high availability by running one or more Eureka servers in each Amazon EC2 availability zone. Each Eureka server runs on an EC2 instance that has an Elastic IP address. DNS `TEXT` records are used to store the Eureka cluster configuration, which is a map from availability zones to a list of the network locations of Eureka servers. When a Eureka server starts up, it queries DNS to retrieve the Eureka cluster configuration, locates its peers, and assigns itself an unused Elastic IP address.

Eureka clients – services and service clients – query DNS to discover the network locations of Eureka servers. Clients prefer to use a Eureka server in the same availability zone. However, if none is available, the client uses a Eureka server in another availability zone.

Other examples of service registries include:

- etcd – A highly available, distributed, consistent, key-value store that is used for shared configuration and service discovery. Two notable projects that use etcd are Kubernetes and Cloud Foundry.
- consul – A tool for discovering and configuring services. It provides an API that allows clients to register and discover services. Consul can perform health checks to determine service availability.
- Apache Zookeeper – A widely used, high-performance coordination service for distributed applications. Apache Zookeeper was originally a subproject of Hadoop but is

now a top-level project.

Also, as noted previously, some systems such as Kubernetes, Marathon, and AWS do not have an explicit service registry. Instead, the service registry is just a built-in part of the infrastructure.

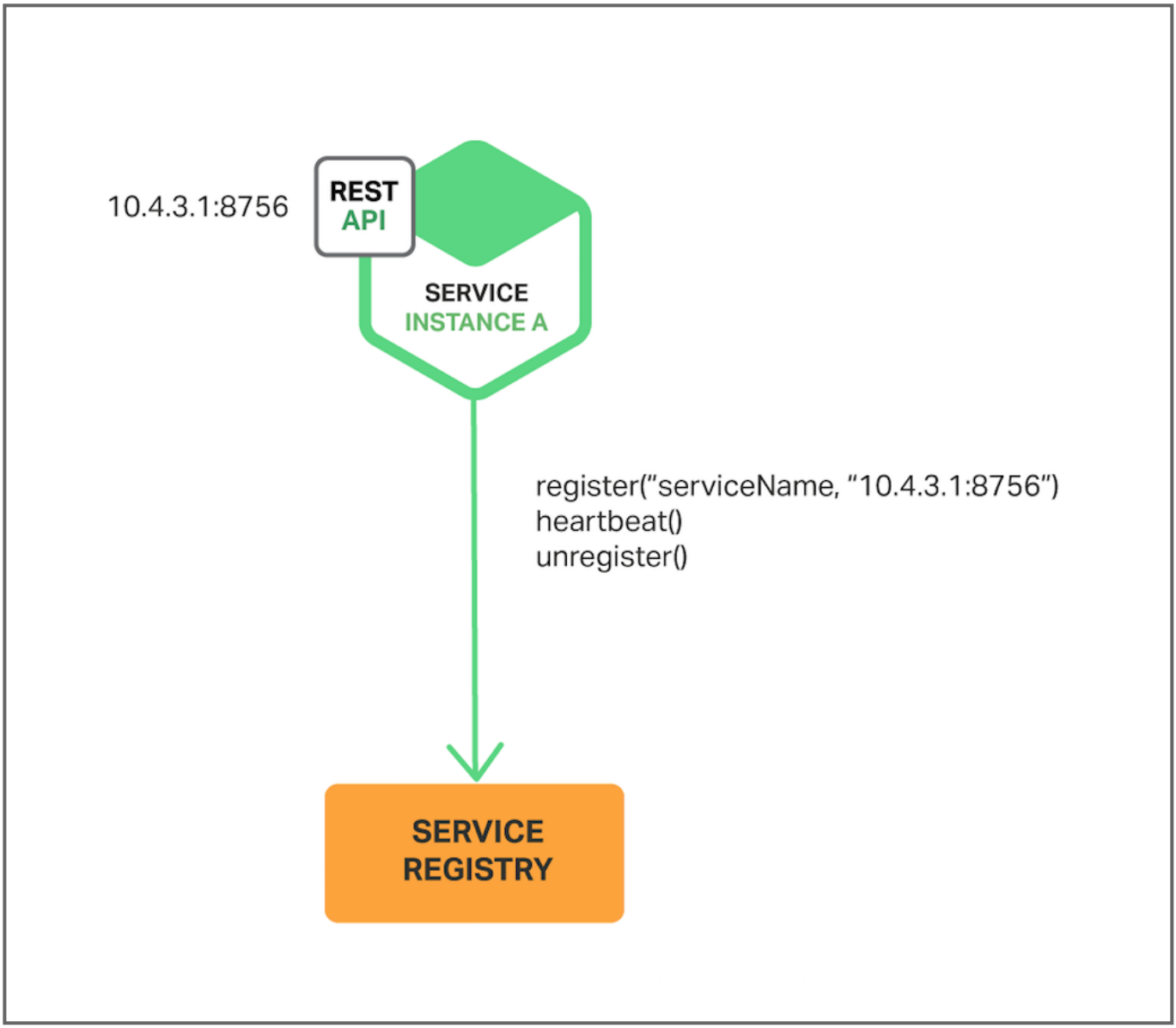
Now that we have looked at the concept of a service registry, let's look at how service instances are registered with the service registry.

Service Registration Options

As previously mentioned, service instances must be registered with and deregistered from the service registry. There are a couple of different ways to handle the registration and deregistration. One option is for service instances to register themselves, the self-registration pattern. The other option is for some other system component to manage the registration of service instances, the third-party registration pattern. Let's first look at the self-registration pattern.

The Self-Registration Pattern

When using the self-registration pattern, a service instance is responsible for registering and deregistering itself with the service registry. Also, if required, a service instance sends heartbeat requests to prevent its registration from expiring. The following diagram shows the structure of this pattern.



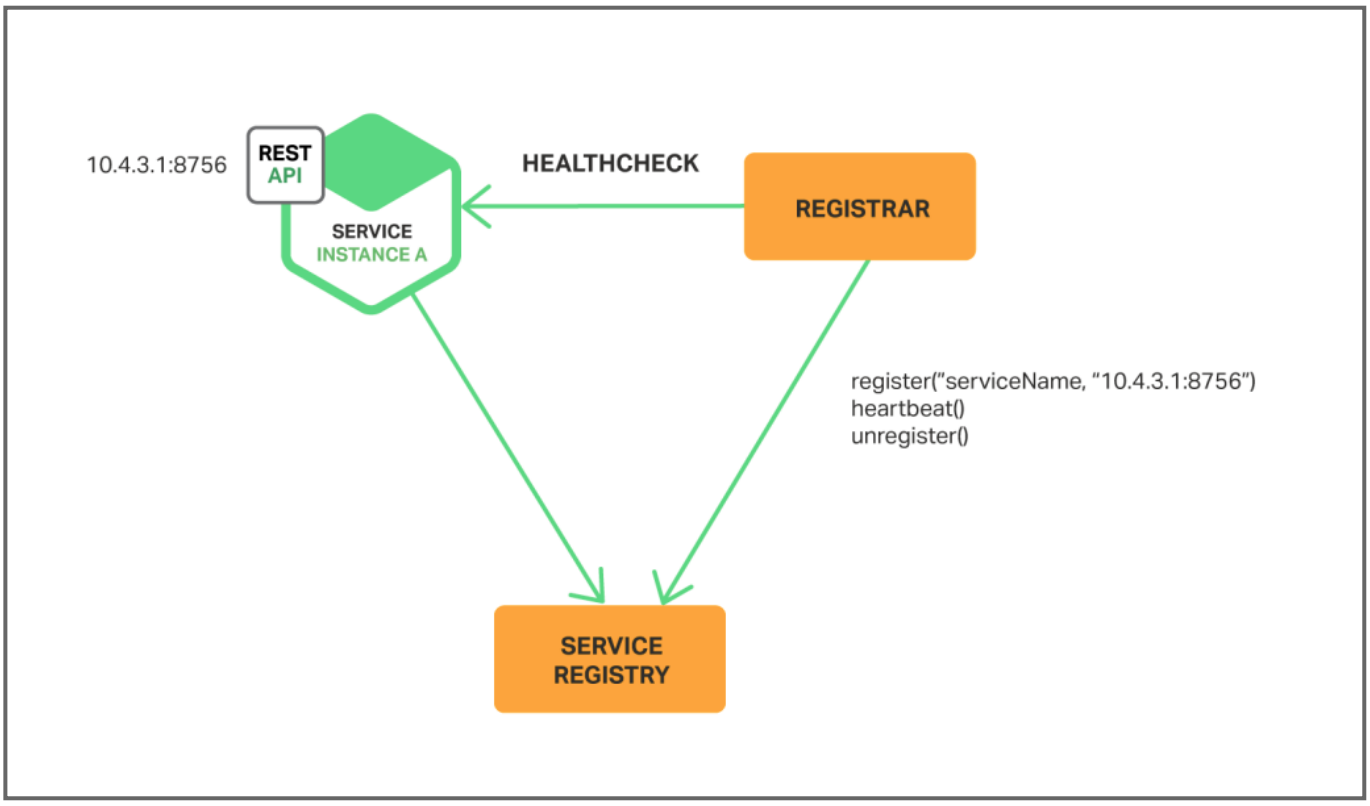
A good example of this approach is the Netflix OSS Eureka client. The Eureka client handles all aspects of service instance registration and deregistration. The Spring Cloud project, which implements various patterns including service discovery, makes it easy to automatically register a service instance with Eureka. You simply annotate your Java Configuration class with an `@EnableEurekaClient` annotation.

The self-registration pattern has various benefits and drawbacks. One benefit is that it is relatively simple and doesn't require any other system components. However, a major drawback is that it couples the service instances to the service registry. You must implement the registration code in each programming language and framework used by your services.

The alternative approach, which decouples services from the service registry, is the third-party registration pattern.

The Third-Party Registration Pattern

When using the third-party registration pattern, service instances aren't responsible for registering themselves with the service registry. Instead, another system component known as the *service registrar* handles the registration. The service registrar tracks changes to the set of running instances by either polling the deployment environment or subscribing to events. When it notices a newly available service instance it registers the instance with the service registry. The service registrar also deregisters terminated service instances. The following diagram shows the structure of this pattern.



One example of a service registrar is the open source Registrator project. It automatically registers and deregisters service instances that are deployed as Docker containers. Registrator supports several service registries, including etcd and Consul.

Another example of a service registrar is NetflixOSS Prana. Primarily intended for services written in non-JVM languages, it is a sidecar application that runs side by side with a service instance. Prana registers and deregisters the service instance with Netflix Eureka.

The service registrar is a built-in component of deployment environments. The EC2 instances created by an Autoscaling Group can be automatically registered with an ELB. Kubernetes services are automatically registered and made available for discovery.

The third-party registration pattern has various benefits and drawbacks. A major benefit is that services are decoupled from the service registry. You don't need to implement

service-registration logic for each programming language and framework used by your developers. Instead, service instance registration is handled in a centralized manner within a dedicated service.

One drawback of this pattern is that unless it's built into the deployment environment, it is yet another highly available system component that you need to set up and manage.

Summary

In a microservices application, the set of running service instances changes dynamically. Instances have dynamically assigned network locations. Consequently, in order for a client to make a request to a service it must use a service-discovery mechanism.

A key part of service discovery is the service registry. The service registry is a database of available service instances. The service registry provides a management API and a query API. Service instances are registered with and deregistered from the service registry using the management API. The query API is used by system components to discover available service instances.

There are two main service-discovery patterns: client-side discovery and service-side discovery. In systems that use client-side service discovery, clients query the service registry, select an available instance, and make a request. In systems that use server-side discovery, clients make requests via a router, which queries the service registry and forwards the request to an available instance.

There are two main ways that service instances are registered with and deregistered from the service registry. One option is for service instances to register themselves with the service registry, the self-registration pattern. The other option is for some other system component to handle the registration and deregistration on behalf of the service, the third-party registration pattern.

In some deployment environments you need to set up your own service-discovery infrastructure using a service registry such as Netflix Eureka, etcd, or Apache Zookeeper. In other deployment environments, service discovery is built in. For example, Kubernetes and Marathon handle service instance registration and deregistration. They also run a proxy on each cluster host that plays the role of server-side discovery router.

An HTTP reverse proxy and load balancer such as NGINX can also be used as a server-side discovery load balancer. The service registry can push the routing information to NGINX and invoke a graceful configuration update; for example, you can use Consul Template. NGINX Plus supports additional dynamic reconfiguration mechanisms – it can pull information about service instances from the registry using DNS, and it provides an API for remote reconfiguration.

In future blog posts, we'll continue to dive into other aspects of microservices. Sign up to the NGINX mailing list (form is below) to be notified of the release of future articles in the series.

Editor *– This seven-part series of articles is now complete:*

1. *Introduction to Microservices*
2. *Building Microservices: Using an API Gateway*
3. *Building Microservices: Inter-Process Communication in a Microservices Architecture*
4. *Service Discovery in a Microservices Architecture (this article)*
5. *Event-Driven Data Management for Microservices*
6. *Choosing a Microservices Deployment Strategy*
7. *Refactoring a Monolith into Microservices*

You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook – Microservices: From Design to Deployment.

Guest blogger Chris Richardson is the founder of the original CloudFoundry.com, an early Java PaaS (Platform as a Service) for Amazon EC2. He now consults with organizations to improve how they develop and deploy applications. He also blogs regularly about microservices at http://microservices.io.