Service Discovery Overview

By **Simplicity Itself Team**, www.simplicityitself.io
June 10th, 2015

When building microservices, you have to naturally distribute your application around a network. It is almost always the case that you are building in a cloud environment, and often using *immutable infrastructure*. Ironically, this means that your virtual machines or containers are created and destroyed much more often than normal, as this immutable nature means that you don't maintain them.

These properties together mean that your services need to be reconfigured with the location of the other services they need to connect to. This reconfiguration needs to be able to happen on the fly, so that when a new service instance is created, the rest of the network can automatically find it and start to communication with it. This process is called *Service Discovery*.

This concept is one of the key underpinnings of a Microservice architecture. Attempting to create Microservices without a service discovery system will lead to pain and misery as you will, in effect, be working as a manual replacement.

As well as the standalone solutions presented here, most platforms, whether full PaaS or the more minimal container managers, have some form of Service Discovery.

Which system to choose? There are currently several key contenders to choose from, ZooKeeper, Consul, Etcd, Eureka and RollYourOwn

ZooKeeper

http://zookeeper.apache.org/

ZooKeeper is an Apache project providing a distributed, eventually consistent hierarchical configuration store.

ZooKeeper originated out of the world of Hadoop, where it was built to help in the maintenance of the various components in a hadoop cluster. It is not a service discovery system per se, but is instead a distributed configuration store that provides notifications to registered clients. With this, it is possible to build a service discovery infrastructure, however every service must explicitly register with ZooKeeper, and the clients must then check in the configuration.

Netflix have invested a lot of time and resources into ZooKeeper, and so a significant amount of Netflix OSS projects have some ZooKeeper integration.

Simplicity Itself Recommends:

ZooKeeper is a well understood clustered system. It is a consistent configuration store, and so being well designed and built, a network partition will cause the smaller side of the partition to shut down. For that reason, you must choose whether consistency or availability is more important to you.

If you do choose a consistent system for service discovery, such as Zookeeper, then you need to understand the implications on your services. You have tied them to the lifecycle of the discovery system, and also exposed them to any failure conditions it may have. You should not assume that 'consistent' means free from failure. Zookeeper is among the older cluster managers, and consensus (pun intended) is that it's implementation of master selection is robust and well behaved.

If you do choose to use ZooKeeper, investigate the Netflix OSS projects, staring with Curator as a first point of call, and only use bare ZooKeeper if they don't fit your needs.

Since Zookeeper is mature and established, there is a large ecosystem of good quality (mostly!) clients and libraries to enrich your projects.

Consul

http://www.consul.io

Consul is a peer to peer, strongly consistent data store that uses a gossip protocol to communicate and form dynamic clusters. It is based on the Serf library.

It provides a hierarchical key/value store that you can place data in and register watches against to be notified when something changes within a particular key space. In this, it is similar to ZooKeeper.

As opposed to ZooKeeper and Etcd, however, Consul implements a full service discovery system in the library, and so you don't need to implement your own use use a third party library. This includes health checks on both nodes and services.

It implements a DNS server interface, allowing you to perform service lookups using the DNS protocol. It also allows 'clients' to run as independent processes and register/monitor services on their behalf. This removes the need to add explicit Consul support into your applications. This is similar in concept to the Netflix OSS *Sidecar* concept that allows services with no ZooKeeper support to be registered and be discoverable in ZooKeeper.

Deployment wise, Consul agents are deployed onto the systems that services are running on, and not in a centralised fashion.

Simplicity Itself Recommends:

This is a newer product, and one that we like a lot. Generally recommended if you are able to adopt it.

As with the other strongly consistent systems in the list, care must be taken that you understand the implications of adopting it, including understanding it's potential failure modes.

If you would like something similar that chooses availability rather than consistency, investigate the related Serf project. The Serf library serves as the basis of Consul, but has chosen different data guarantees guarantees. It is nowhere near as full featured, but can handily survive a split brain scenario and reform afterwards without any ill effects.

Etcd

http://www.etcd.io

Etcd is an HTTP accessible key/ value store. In that, it is similar in concept to ZooKeeper and the K/V portion of Consul. It functions as a distributed, hierarchical configuration system, and can be used to build a Service Discovery system.

It originally grew out of the CoreOS project, is maintained by them and recently achieved a stable major release

Simplicity Itself Recommends:

If you are primarily using HTTP as your communication mechanism, then Etcd can't be easily beaten. It provides a well distributed, fast HTTP based system, and has query and push notifications on change, via long polling.

Eureka

https://github.com/Netflix/eureka

Eureka is a 'mid tier load balancer' built by Netflix and released as open source. It is designed to allow services to be able to register with a Eureka server and then locate each other via that server.

Eureka has several capabilities beyond other solutions presented here. It contains a built in load balancer which, although fairly simple, certainly does it's job. Netflix state that they have a secondary load balancer implementation internally that uses Eureka as a data source and is much more full featured. This hasn't been released.

If you use Spring for your projects, then Spring Cloud is an interesting project to look into, in order to be able to automatically register and resolve services in Eureka.

Simplicity Itself Recommends:

Like all Netflix OSS projects, it was written to run on the AWS infrastructure first and foremost. While other Netflix OSS projects have been extended to run in other environments, Eureka does not appear to be moving in that direction. We very much like the relation between client and server in the Eureka design, as it leaves the clients with the ability to continue working if the service discovery infrastructure fails.

Server wise, Eureka has also chosen availability rather than consistency. You must also be aware of the implications of this choice as it directly affects your application. Primarily, this manifests as a potentially stale or partial view of the full data set. This is discussed in the Eureka documentation.

For Spring projects, Eureka is now the quickest to get started with due to the investment the Spring team has made in adopting the Netflix OSS components via the Spring Cloud sub-projects.

RollYourOwn

If you can't adapt to that, then you will have to create your own discovery solution within your existing infrastructure.

The basis of this will be :-

- Services must be able to notify each other of their availability and supply connection information
- Periodic updates to the records to strip out stale information
- Easy integration into your application infrastructure, often using a standard protocol such as HTTP or DNS
- Notifications on services starting and stopping.

Simplicity Itself Recommends:

Building your own discovery service should not be taken lightly. If you do need to then we recommend building a system that values availability rather the consistency. These are significantly easier to build, and more likely that you will build something that is functional.

The approach we would recommend would be to use some existing message infrastructure and broadcast notifications on service status. Each service caches the latest information from the broadcasts and uses that as a local set of service discovery data. This has the potential for being stale, but we've found this approach to scale reasonably well and is easy to implement.

If you do require consistency, then using some consistent data store could serve as the basis for a distributed configuration system that can be used to build service discovery. You will also want to emit notifications on status changes. You should realise, though, that building a consistent, distributed system is exceptionally hard to get right, and very easy to get subtly wrong.

Overall, really not recommended, but certainly possible.