6 QUESTIONS TO 4 EXPERTS ON SERVICE DISCOVERY

Brought to you by HighOps – https://highops.com

The topic of Service Discovery is not new (just think DNS!) but it's been moving fast over the last few years thanks in part to the explosion of services and microservices architectures and, of course, Docker.

As it often happens in these cases, we see a lot of different interpretations of its meaning, benefits, challenges, and adoption paths. We asked 4 experts who have been thinking, writing and implementing Service Discovery to share their experience by answering the following questions on the topic:

1) What does Service Discovery mean to you?
2) What are the key aspects Service Discovery should have and why?
3) What are the main benefits you see/care about?
4) What are the most solid options out there today?
5) Biggest adoption challenges that are not there yet in your opinion?
6) Starting from scratch is (relatively) easy. What about those with existing systems? any hints on how people could get started introducing Service Discovery in existing systems?

Marco Abis
July 5th, 2015

Experts Profiles

Sam Newman

My name is Sam Newman, a techie at ThoughtWorks. Aside from other things I've committed sporadically to open source projects, spoke at more than a few conferences, and wrote some things including the book Building Microservices for O'Reilly. http://samnewman.io - https://twitter.com/samnewman

Nitesh Kant

Nitesh Kant is an engineer in the Cloud & Platform engineering team at Netflix, where he has been leading the evolution of the Inter Process Communication (IPC) stack to follow the reactive programming paradigm. He is the core contributor of the reactive IPC library RxNetty which forms the heart of this new stack. He's also a contributor to eureka: Netflix's Service Discovery system and has ideated and designed the 2.0 version. Eureka is the backbone of Netflix's microservices architecture handling thousands of application instances in the Netflix ecosystem.
https://twitter.com/niteshkant

Jeff Lindsay

Jeff Lindsay is the author of Dokku and many Docker related open source projects. He is the co-founder and principal of Glider Labs, a DevOps consultancy specializing in modern, Docker-oriented system architectures. Jeff was involved in the early stages of Docker development, and has worked with many other organizations such as Twilio, Digital Ocean, and NASA Ames on distributed systems and developer platforms.
http://progrium.com/ - https://twitter.com/progrium

Eberhard Wolf

Eberhard Wolff works as a Fellow for innoQ. His technological focus is on modern architectures - often involving Cloud, Continuous Delivery, DevOps, Microservices and NoSQL Eberhard is a regular speaker at international conferences and author of over 100 articles and several books - most in German.
https://twitter.com/ewolff

Highlights

The full transcript of all answers is at the end and we recommend you read them all since they cover a lot of ground. Here are just some highlights:

1) What does Service Discovery mean to you?

It keeps track of all the services in a (large scale) distributed system so that they can be found by both people and other services. Think of DNS as a simple example but on steroids: complex systems need features like storing metadata about a service, health monitoring, varying query capabilities, realtime updates, etc.

It differs depending on the context, e.g. network device discovery, rendezvous discovery, SOA discovery but in all cases it's a coordination mechanism for services to announce themselves and find others without configuration.

2) What are the key aspects Service Discovery should have and why?

As the backbone of any large-scale service oriented architecture Service Discovery needs the be highly available and cover 3 main aspects: Registration, Directory and Lookup. Having just the Directory is not enough.

As mentioned the ability to store metadata is key since complex services provide multiple service interfaces and ports and often require non-trivial deployment environments. And once you have lots of metadata you need powerful querying capabilities, including health/status.

3) What are the main benefits you see/care about?

The main benefit is what in network discovery referred to as "zero configuration": rather than hardcode addresses we specify a service name (and sometimes not even that!). In modern architectures nodes come and go and you need to decouple individual service instances from the knowledge of the deployment topology of your architecture.

Having a way to get insights into the deployment status of all the services and to control the available instances from a centralized place becomes key, especially in complex system s which warrant something more than just DNS.

4) What are the most solid options out there today?

Service Discovery solutions are a plenty today in the industry.

As mentioned DNS has been used for a long time and is probably the largest Service Discovery system out there. For small-scale setups start with DNS but once you start provisioning nodes more dynamically, DNS starts becoming problematic due to the propagation time.

Arguably, Zookeeper is the most mature of the config stores used for discovery since it has been around for quite some time and is a comprehensive solution including configuration management, leader election, distributed locking etc. This makes it a very compelling general-purpose solution although it's often more complex than it could be.

etcd & doozerd are the new age cousins of Zookeeper, built with similar architectures and features sets and hence can be used interchangeably in place of Zookeeper

Consul is a newer solution in this space that provides configuration management and a generic key-value store apart from Service Discovery. It also has killer features of health checking of nodes and supporting DNS SRV for improved interop with other systems. A big differentiator from Zookeeper is the HTTP & DNS APIs that can be used to interact with consul vis-à-vis a Zookeeper client.

If you lean more towards AP systems Eureka is a great choice and is battle tested in Netflix and it prefers Availability over Consistency in the wake of network partitions.

5) Biggest adoption challenges that are not there yet in your opinion?

It's more complicated than you realize: it's an extension of the distributed systems problem.

You might roll out configuration files with service names, IPs and ports but when the system becomes very dynamic you need to migrate to a "real" Service Discovery solution and that migration is usually not as easy as you think. One of the biggest challenges is the inability to understand how intrusive the choice of a Service Discovery system is: once chosen it is very difficult to change it and hence it is critical to do it right.

Most systems implement some form of distributed consensus algorithms, designed to be resilient in the face of node outages, but these algorithms are notoriously hard to get right and understanding failure modes is both key and difficult and failing to analyse them correctly usually takes you to make the wrong choices.

6) Starting from scratch is (relatively) easy. What about those with existing systems? Any hints on how people could get started introducing Service Discovery in existing systems?

The first step of starting to use a Service Discovery system is for clients to stop baking knowledge about the dependent service deployment environments. Once the clients stop baking that knowledge in code they are forced to store that knowledge externally. Starting as simple as storing this knowledge in properties, they can slowly graduate to looking up this knowledge dynamically from an external Service Discovery system.

Once you pick a mechanism (network or service) and understand what you need to optimize for (and that's maybe the hardest part), it's just a matter of introducing integration points for registration and lookup.

Full Answers

1) What does Service Discovery mean to you?

Eberhard: Service Discovery is a basic concept in distributed systems and has a long history. However, it didn’t seem interesting to the general public until recently. The new interest indicates that more people are now creating truly distributed systems and need this kind of infrastructure.

Service Discovery is pretty fundamental for a distributed system. It keeps track of all the services. Such a centralized registry is a great foundation to build others infrastructure e.g. for monitoring, load balancing or routing.

Nitesh: Service Discovery is for a large-scale system, what an address book is for your email client.

Just like an address book helps you find people you want to send an email to, Service Discovery helps you find a dependent service in a large scale, service oriented system. Simplest example of a Service Discovery system is DNS (Domain Name System), where every service has a name and the DNS system converts these names into a set of IP addresses.
Complex service oriented architectures, however, require advanced form of Service Discovery which include features like storing metadata about the service, health of service instances, varying query capabilities, realtime updates, etc.

Sam: A means by which both a machine and a human being can find a given instance of a specific service.

Jeff: In general it's a coordination mechanism for services to announce themselves and find others without configuration. There are maybe three different contexts I think about Service Discovery:

1. Network device discovery - This is where most Service Discovery protocols exist like mDNS and Bonjour, where we talk about "zero configuration networking". This is usually more about services provided by devices on a local network, like a printer. In this context, Service Discovery is more network oriented.

2. Rendezvous discovery - This is using a known stable endpoint on the Internet to bootstrap a connection between several components. Etcd's cluster discovery URL, Hashicorp's Atlas service for bootstrapping Consul clusters, and even STUN servers fall into this category. This context is more bootstrap oriented.

3. SOA discovery - This is discovery designed for application services to discover each other. It's an evolution of network device discovery, but more application level than network level. It typically utilizes a directory service like etcd or Consul, which is probably the most significant differentiator from network device discovery. Some are able to use network discovery protocols for SOA discovery, but this context is more application oriented.

I've heard Service Discovery used to refer to all three of these, but I haven't heard anybody differentiate these before. I think it's useful to make these distinctions. That said, these days I think most people are talking about SOA or application-level discovery.

2) What are the key aspects Service Discovery should have and why?

Eberhard: I think Service Discovery systems should be highly available. If other services can’t be discovered the whole system might be unavailable. This might for example be the case if load balancing is used and each call to a system goes to the Service Discovery system to get a different node.

According to the CAP theorem you need to compromise between consistency (C) and availability (A) if you want partition tolerance (P). Network partition can always occur so there is no way to compromise on P.

So you can choose availability or consistency. No consistency just means new nodes won't be discovered or discovered nodes might be down - which the client will notice. The consequences are not very severe. So Service Discovery should be more AP than CP. Service Discovery is different from databases or systems that store configurations. Such data must be the same across a set of servers. In that case consistency might be more important.

Also advanced options such as load balancing or monitoring might be integrated into Service Discovery. And of course the solutions need to scale to the number of services in the system.

Also the Service Discovery solution should fit into the infrastructure of the system: Which platform is it written for? Which programming language does it use? If all of your system runs on the JVM maybe the Service Discovery should, too. And client libraries for all used platforms must be available - or other solutions like sidecars.

Nitesh: Service Discovery forms the backbone of any large-scale service oriented architecture as every service in the ecosystem depends on Service Discovery to function correctly. Hence, the most critical aspect of any Service Discovery solution is availability, in absence of which, it can become an Achilles' heel of the application using it.

Another important aspect is the ability to store structured, metadata about the service instance. This is important as very rarely in a large-scale system, a service is as simple as exposing a single port for service.
Complex services provide multiple service interfaces and ports. They also have a complex deployment environment where an instance may belong to multiple categories under the same service and the usage patterns change based on those categories. It is thus important for the clients interacting with these services to have a comprehensive knowledge about the deployment environment of any instance and use it while interacting with it.

Since, any instance has an associated structured metadata, it becomes important for a Service Discovery system to provide powerful querying capabilities to query the instances of a service. Complex applications use these querying capabilities to narrow down on instances that can be used in different scenarios. e.g.: A service may have some canary instances deployed with a new behavior and these instances can be used for certain users, percentage of traffic, etc. By attaching these instances to a different category, it can be indicated to the clients that these instances are different and must only be used in certain scenarios.

Another feature that many Service Discovery systems provide is an insight into the health/status of an instance. This is a way to control the usage of the instance by the clients, as clients should not use an instance which is not healthy or available. Although simple but this feature becomes really powerful to isolate an instance for debugging or take an entire farm of instances out of service (unavailable) in the wake of deployment bugs. What makes this feature much more powerful is when a Service Discovery system provides a means to override the status published by an instance, without changing the instance at all i.e. from an administrative console, for example.

Sam: Service Registries need to support the ability for services to be registered, and looked up. These are the basics. A richer query API can also be beneficial, for example to allow me to find an instance running in the same rack or data centre, or based on some other form of meta data. These systems are key - they need to be highly available. Understanding how the underlying system trade-offs consistency during failure modes is a key consideration for selecting the right tool for your use case.

Jeff: This is going to be different depending on the context and use case. For example, application-level discovery is often done with a centralized service making it more stateful. Whereas network-level discovery is often done using a stateless broadcast mechanism with no central service.

If we focus on the centralized approaches, and by centralized I just mean there being a central directory, there's three parts to discovery:

1. Registration - How does a service announce itself? In this case, how does it register with the directory service. This can happen in-process or out of process. It can include other metadata, but typically focuses on a name and address.

2. Directory - Where and how is the service directory stored? Is it eventually consistent or strongly consistent? Highly available? Is it service health aware? Can you get real-time updates?

3. Lookup - How does an application use discovery to connect to another service? Does it require a client library and do it in-process? Is it lightweight and use only DNS? Or is a proxy involved? Does it address load balancing? Unresponsive services?

An application Service Discovery solution needs to have a good answer to all three, not just provide a directory. But as you can see, even each leg of the problem has a number of different properties. Also, the integration points, registration and lookup, can be very application specific. So depending on the "stack" you use, you get different answers and different levels of generalness or specific optimizations.

At the end of the day, you just need an answer for registration and lookup for discovery. Everything else is an implementation detail.

3) What are the main benefits you see/care about?

Eberhard: In modern architectures nodes come and go - not just because of hardware failures but also because of deployments or elastic scaling.
The only way to deal with this is a system for Service Discovery that supports such a dynamic environment. So I think at a certain level of complexity you will need Service Discovery in your system - in particular with Microservices systems counting lots of services running on a huge number of servers.

Nitesh: The biggest benefit of using a Service Discovery system is to decouple individual service instances from the knowledge of the deployment topology of your architecture. Any service instance does not need to bake in code; the knowledge about how to reach a dependent service instance and the dependent service instances can make, otherwise breaking changes, like the port they listen to, for example. In absence of a Service Discovery system, such changes would be impossible to do.

Other benefit would be the ability to have insights into the deployment status of all the services in a service-oriented system and the ability to control the available instances from a centralized place.

Sam: Using a dedicated Service Discovery tool is something I do when a system I am working with achieves a level of complexity that warrants something more than just DNS. At that point, I'm looking for ease of registration & lookup, and ease of management of the tool itself, especially in highly resilient setups

Jeff: The main benefit is what network discovery referred to as "zero configuration". Instead of hardcoding addresses, we can specify a service name. Sometimes not even that. So it eases configuration and lets you decouple the application and the network.

Since Service Discovery effectively takes care of its own configuration, it can also change. If a new service comes online or an existing service dies, your application can be aware of this and adjust accordingly.

Both of these are incredibly import for cloud environments, where hosts can come and go easily, and microservice architectures, where there can be many services you'd have to connect up manually otherwise.

4) What are the most solid options out there today?

Eberhard: DNS - we use it every day and it is probably the largest Service Discovery system out there. We often forget what a great job those well established technologies do. And DNS is definitely rock solid and well understood. Spotify for example is actually using pure DNS for Service Discovery.

As I said before I lean more towards AP systems. And the Service Discovery system should fit your platform. I'm a Java/Spring guy so I like Eureka: It is AP, is written in Java and integrates easily with Spring Cloud. Also it has been battle proven at Netflix. However, there are quite a few other options that have great track records, too.
As I said I think Service Discovery is different from storing configuration - which might in fact be a CP issue. So I would rather not use systems that are originally designed to store configuration data and can be made to support Service Discovery, too.

Nitesh: Service Discovery solutions are a plenty today in the industry.

Historically, people use Zookeeper as the solution of choice which is a much comprehensive solution including configuration management, leader election, distributed locking etc. This makes it a very compelling general-purpose solution, which can also be used for a variety of use-cases apart from Service Discovery.

Etcd & doozerd are the new age cousins of Zookeeper, built with similar architectures and features sets and hence can be used interchangeably in place of Zookeeper.

Consul is a newer solution in this space that provides configuration management and a generic key-value store apart from Service Discovery. A big differentiator from Zookeeper is the HTTP & DNS APIs that can be used to interact with consul vis-à-vis a Zookeeper client.

Netflix's eureka is a solution built specifically for Service Discovery and is different than all other solutions above, as it prefers availability over consistency in the wake of network partitions. It also provides an HTTP API that makes it easy to use.

Sam: Start with DNS. Seriously. For small-scale setups, it's fine. Clients just look up instances based on a hard-coded DNS value. Registration of services can be painful however.

Once you start provisioning nodes more dynamically, DNS starts becoming problematic due to the propagation time. It is also fairly restrictive if you want to start doing more complex querying, for example looking for given instances of nodes in a certain data centre or rack to implement client-side load balancing or resiliency through replication.

In AWS, you may get away with just using the AWS query APIs and tags on resources. But assuming you're actually looking for a dedicated service registry system, three of the most obvious candidates right now are Zookeeper, etcd consul.

Zookeeper is tried and tested, but is long in the tooth and some find it painful to manage and isn't as nice from a registration and lookup point of view as the others (I know at least one project where they are trying to create more field HTTP frontons on top of Zookeeper).

Etcd and consul are the new kids on the block. Etcd have recently made some fairly significant changes to their core APIs, which makes me more wary, but Google think it is good enough to be part of Kubernetes.

Consul however is my favourite. A nice dashboard, easy service registration, and the killer features of health checking of nodes and supporting DNS SRV for improved interop with other systems. Throw in consul-template (which updates config files dynamically based on updates to consul) and consul-env and you have a great ecosystem of tooling around it.

Jeff: Pretty much anything that says it helps with Service Discovery will work, it just depends on what properties you want or need. For example, maybe you don't want the hassle of a centralized directory, so if you can use a multicast approach based on mDNS, that might be ideal. But EC2, for example, doesn't let you do multicast. Or sometimes you want better consistency. So then you'd look at etcd or Consul.

Arguably, Zookeeper is the most mature of the config stores used for discovery, but I wouldn't recommend it. I've found etcd and Consul are just as effective and in many ways much simpler and better suited for discovery.

5) Biggest adoption challenges that are not there yet in your opinion?

Eberhard: If you build a distributed system you will need to make the services find each other. However, you might roll out configuration files with service names, IPs and ports. This might work - until the system becomes too dynamic or there are just too many services. At that point you would need to migrate real to a Service Discovery solution - but that migration might be not easy. So existing solutions that are too simplistic for the system are a adoption challenge.

Nowadays the need for Service Discovery seems to be well understood. People are using it right from the start. I guess before that adoption was a problem of education: People did not know about these systems and were not facing the problems theses systems solve.

Nitesh: In my opinion, the biggest challenge is the inability of people to understand how intrusive the choice of a Service Discovery system is, in their architecture.

Once chosen, it is very difficult to change it and hence it is critical to do it right. In order to properly understand this intrusive nature, one should understand the various failure modes a Service Discovery system brings to the fore and how does your system behave in wake of such failures. Failure to do these analysis, leads to wrong choices from not using a proper Service Discovery system to not using Service Discovery at all.

Sam: Production case studies, especially for etcd and Consul. Both of these systems implement some form of distributed consensus algorithms, designed to be resilient in the face of node outages. These algorithms are notoriously hard to get right. Kyle Kingsbury (aka @aphyr), who has made a career recently of testing the claims made by systems like these, did find flaws with both etcd and Consul early on (https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul), although these were fixed fairly quickly. The failure modes associated with the implementations of these often complex algorithms can be hard to spot though, and may only occur infrequently enough that outside of rigorous testing like that performed by Kyle you may only encounter issues where you least want them - in production.

Jeff: It's more complicated than you realize. It's an extension of the distributed systems problem. All those questions I mentioned above. It's just making you think about a lot of stuff you normally might not think about, especially as a developer. At the same time, if it's set up and taken care of, it actually makes your life easier as a developer.

Also, until recently a lot of solutions were more "invasive" and required you to modify your application. I've spent a lot of time thinking about how to generalize and simplify the problem. For example, SDN simplifies discovery a little, but at the expense of more complexity elsewhere in your stack.

6) Starting from scratch is (relatively) easy. What about those with existing systems? Any hints on how people could get started introducing Service Discovery in existing systems?

Eberhard: The first question should be what you expect from the using Service Discovery in your systems. It might be that you outgrew your old approach as mentioned above. In that case you will need to introduce Service Discovery. You might want to do that migration big bang if your system is small enough. There are other ways, though. For example you could add the Service Discovery to a new part of your system to just take care of a few services and then let it grow from there. There are also other motivations and therefore different ways of adoption. Service Discovery gives you location transparency.

The clients don't know where a system is running - they rely on the Service Discovery to provide that information. So to decouple clients from the services you might introduce Service Discovery and then split systems or deploy them on different nodes. These changes are transparent to client because they just rely on the Service Discovery to return the current servers. Introducing Service Discovery might therefore be a good way to tear a monolith apart.

Nitesh: The first step of starting to use a Service Discovery system is for clients to stop baking knowledge about the dependent service deployment environments.

Clients should not be shipped with the knowledge of what network interfaces and ports a service instance exposes, how many instances of the service are available, etc.

Once, the clients stop baking that knowledge in code, they are forced to store that knowledge externally. Starting as simple as storing this knowledge in properties, they can slowly graduate to looking up this knowledge dynamically from an external Service Discovery system.

Sam: Start simple, start with DNS. Then, if you need a more sophisticated solution, I'd pick consul due to its support for DNS SRV. This will allow you to gradually shift over clients to Consul's REST API as and when you need the additional power, while older clients can still use DNS.

Jeff: Once you pick a mechanism (network or service) and understand what you need to optimize for (and that's maybe the hardest part), it's just a matter of introducing integration points for registration and lookup.

So I guess there's no easy answer. I mean, it's not "hard" it just requires thought. Learn more about it, figure out what you need and where your own pain points will be, and try the simplest thing first. Iterate from there.