James Phillips on Service Discovery

Charles Anderson

**SERVICE DISCOVERY** has become increasingly important in the construction of scalable, elastic, and always-on distributed systems. In this interview, James Phillips, who works on the open source service discovery tool Consul at HashiCorp, explains what service discovery is, provides use cases, and discusses available tools. You can listen to the entire interview at www.se-radio.net. Portions of the interview not included here for reasons of space discuss consensus, security, monitoring, durability, and bootstrapping.

Charles Anderson (CA): At a high level, what’s service discovery?

James Phillips (JP): A service can be a logical entity within your architecture. You might have a database or a pool of application servers. At the highest level, service discovery is a system that lets you ask, “Where is a service located?” The answer is usually an IP (Internet Protocol) address or another important number. A set of processes usually back a given service, and for most highly available or scalable things, more than one process makes up a service. Service discovery lets you find those things on a spectrum. You can have a fully statically configured service discovery system or a dynamic one. The techniques you use to do service discovery vary, depending mostly on how fresh the information is and how changes get put into your service discovery catalog.

CA: When you say “static configuration,” that makes me think of configuration files, which I might update occasionally. Then I could use a configuration tool such as Puppet or Chef. What’s wrong with that approach?

JP: That approach can work okay when you have a small handful of things to manage. But as soon as you get into the medium-to-large, large, to very large infrastructure, it becomes very hard to keep that catalog up to date by hand. The time involved to do a convergence run can take minutes. That’s often not fast enough to propagate a change. Also, having a human in the loop to check in changes to that catalog and to manage a list of IP addresses, and a system to push that out, becomes a burden when you want to run as automatically as possible. It becomes a maintenance and response time problem for infrastructure when it gets beyond trivial scale.

CA: And I can imagine that, even at a small scale, response time would be a problem, as you mentioned.

JP: Think about a master failover of your database. You need to point all
your clients somewhere else, and you want that change to go out quickly. You don't want that to take tens of minutes to propagate through your system.

The configuration management tools definitely have a role, even if you're using a service discovery system for bootstrapping your nodes or getting the service discovery system kicked off. But managing the ongoing state of the infrastructure becomes difficult to do with a static setup.

CA: What system scale is appropriate for service discovery—LAN, single datacenter, or across datacenters?

JP: My short answer to that is, yes, it can have a place at all of those different levels. You would not have it within an application process. You're not going to find things like components or plug-ins with a service discovery mechanism. But once you have a process trying to find another process, it's appropriate. For highly available or globally distributed infrastructures, it definitely makes sense for some service in datacenter A to reach out and find an instance of another service in datacenter B.

Usually there's a preferential scope. You want to use things that are close and nearby. Sometimes it's part of normal operations to find things remotely that are located in a much larger tier across the Internet. It definitely makes sense when you're talking about [geographic] redundancy and failover. Ideally, your service discovery system would let you cross those boundaries. Your application has to know how to reach out to all of these different things.

CA: On a local network, is service discovery comparable to zero-configuration networking—something such as Apple Bonjour?

JP: That is definitely the most basic example: “What's the IP address of an instance of this type of service?” We'll see later that, for managing other types of configuration or orchestration, that [approach] probably isn't ideal. [Finding IP addresses on a LAN] requires multicast. [However, that approach] is often a low-level piece of a bigger service discovery solution.

CA: You mentioned processes finding each other. Can you tell us more about scenarios in which service discovery would be appropriate? I can think of examples such as a service-oriented architecture or possibly microservices.

JP: Definitely. Even a monolithic application must deal with issues like “Where's the database?” and “How do I connect to it?” That makes sense in any type of architecture. But, as you move to service-oriented architecture or microservices, you're going to have a much more distributed set of pieces that need to find each other. Behavior becomes more dynamic as you have an architecture that's distributed across functional pieces, with different pieces of the architecture coming and going over time. In general [the difference is that] you'll have more than one instance to choose from. You want to choose a healthy one to talk to.

Service discovery really shines there. Under a resource scheduler, like Kubernetes, or Mesos, or Nomad, some [containers] are placed onto the machines in your cluster by an automated infrastructure. At that point, there is no realistic way to have humans editing config files and then pushing out the changes. You need something that, in real time, can manage where all your resources are, which ones are healthy, and where to find them.

CA: Are there applications or scenarios in which I wouldn't want to use service discovery?

JP: I don't think so. I think the basic hygiene of separating the configuration out of your application is a good thing to start with early on. You don't want to be hard-coding IP addresses in your source code, to give an extreme example. You will progressively use more features of service discovery as your application gets more sophisticated.

It's possible to start simple within an integration that's very, very lightweight, even zero impact on your application code, and then expand from there by adding more orchestration features—for example, electing a leader among many potential leaders in your pool of services.

If you build machine images, it's nice to separate out the service discovery piece into a separate layer to avoid the need to make a new machine image when it's some configuration changes. Having that layer in there can even have practical implications for your deployment pipeline and how you manage your images for deployment.

CA: You said something there that made me think about immutable infrastructure. If you're talking about baked-in Amazon Machine Images, extracting out this variable configuration and information would make sense, right?

JP: Having that layer there and then being able to dynamically push
changes out after something is deployed, without having to get into that image and change its configuration, is a super-big plus. That same machine image can connect to any database or any instance of your API server because it’s getting all of that configuration on the fly from the service discovery system.

CA: Let’s move into some more technical details. What data are typically stored in service discovery repositories? You mentioned host names, or IP addresses, and port numbers. Is that the extent of it? Or is there more?

JP: That’s the most fundamental data. When you are managing services and their configuration, you generally need a little bit more information—things like database user name, potentially credentials, tokens, and other information along with the basic ID and port information that’s already stored in the service discovery system.

It’s also a good place to put things like feature flags. Nearly every service discovery system supports a general bucket of key–value information, which can be used to capture the whole set of the configuration data that you need for interacting with other services.

CA: What are a service discovery tool’s typical components?

JP: A complete solution for service discovery comes down to four different pieces. There’s the core service discovery: So, where is this thing? What’s this IP address and port? How do I connect to it? And there is going to be a health-monitoring piece. It’s not useful to get an instance that’s no longer functioning when you’re looking for something to connect to. The health-monitoring component creates a fresh and live set of data that is easily managed.

You need some configuration storage. So that’s generally in the form of a key–value store. That can also be used for coordination. And then you need an orchestration piece. It’s often useful to be able to elect a leader from many possible instances or to make sure that an operation is done with no race conditions.

CA: Would load balancing be a function of service discovery? Or is that going to be something separate?

JP: It can be. That’s an architectural choice with different service discovery systems. Some systems make load balancing a first-class part of the architecture. In those cases you route all of your traffic at a load balancer that manages talking to the different healthy instances. It’s often the case, though, that you can avoid a load-balancing tier by using service discovery for that purpose. You can avoid that one tier and one potential point of failure. You can talk directly to a healthy instance by virtue of how your service discovery system works.

There are tradeoffs and there are choices. One common practice is within the datacenter not to use load balancers. But you’ll use your service discovery system to configure an external load balancer that your customers are using. Some systems run load balancers internally but route traffic over late hours. There’s a range of possible solutions.

CA: How do servers that want to offer services know where they’re going to register? And on the flip side, the clients that are looking for a service—how do they know? Do we need another layer of service discovery?

JP: It’s turtles all the way down. You generally need some kind of seed or some way to introduce a new entity into the service discovery system. Some systems do that with well-known DNS records for a root service. There are different architectures in terms of how you access the service discovery system. Consul, for example, runs an agent on every node. Your applications only ever talk to their local agent, and then Consul manages talking to other servers and routing requests. Other systems make you locate a central server, and then you make requests against that. But there usually is some sort of bootstrapping process to kick things off.

CA: You said “seed.” At some point in time, all the way down to following all the turtles down, we get an egg or a seed there.

JP: There’s a hard-coded IP address or something similar somewhere. If you have a well-known thing to reach out to, it will manage the task of finding you a server to join with.

CA: Service discovery is needed by the whole app. It must be reliable. How is that reliability provided?

JP: That’s a prime concern. It would be a single point of failure if it wasn’t done properly. A key component of any service discovery system is to be distributed and replicated. You need to be able to build out redundancy to whatever level of failure tolerance you want.

And that may mean having redundancy at each level of scale. Within your local area network you might have multiple servers. And then you may also want redundancy by having federations of servers that can talk to each other and
might be in different geographic locations. A distributed system with replication must handle a server going completely offline, or two servers. Partition tolerance is very important. And you need to deal with what happens when the system is down altogether.

In the event of a partition there are different strategies. Consensus algorithms require a minimum number of servers for a quorum. You want it to operate even in degraded modes. If you get below that, maybe you can’t write to your system anymore but you still want to read. The client can say, “Hey, I’m willing to take some stale information. What’s your best information that you had as of this long ago?”

There are many techniques that good service discovery solutions have for managing [partial failures]. Do they perform retries? How do they avoid thundering herds when things come back online? How do they randomize traffic to spread load among different components? How do you scale events, based on the size of the clusters? There are many layers that go into making a robust service discovery system.

CA: You mentioned a consensus algorithm, which would lead us into the CAP (consistency, availability, partition tolerance) theorem, in terms of making tradeoffs?

JP: Different tools have different strategies and tradeoffs. There are the AP-type systems that provide better availability but make it (potentially) much harder to reason about failure modes, because you can end up with two different sides of your cluster working in two different ways.

And then there are systems that go the CP route. Those are often easier to reason about because they have a consensus algorithm that lets you know the state of your cluster. They have a known behavior in the event that things go wrong. Depending on your application, there are considerations. I would say that most systems tend toward a CP-type architecture.

CA: Preferring consistency and partition tolerance over availability?

JP: Yes, but with the caveat that you have read availability even in the loss of a quorum, say. The minority that can’t make progress in terms of making writes can still read the current state as it was at the time the partition happened, for example.

CA: We’ve mentioned databases a couple of times. Suppose rather than running my own database server, I’m hosting in Amazon Web Services, and I’m using Amazon Relational Database Service to provide a MySQL or Postgres server. Would I still be using service discovery to discover the database, even though in theory Amazon is going to keep it pretty stable for me?

JP: I think so, because tools such as Consul have support for what we call external services. So those are static registrations, but they are served in the same manner in terms of Consul’s APIs, as any other service.

Charles Anderson is a software developer who focuses on systems, back-end, and infrastructure programming. Contact him at cander@cander.org.

See www.computer.org/software-multimedia for multimedia content related to this article.