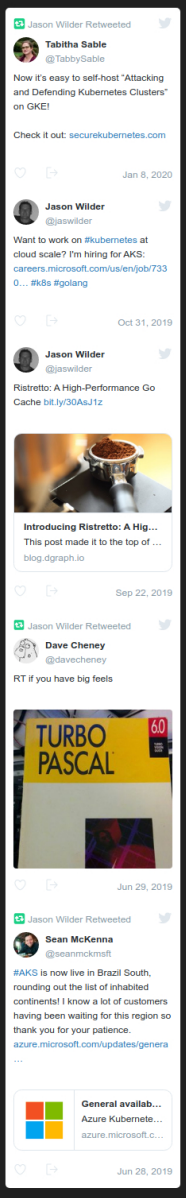
Jason Wilder's Blog

Software developer and architect interested in scalability, performance and distributed systems.



Open-Source Service Discovery

Feb 4, 2014 · 14 minute read · 31 Comments

service discovery | doozer | etcd | zookeeper | architecture | smartstack | nsq | skydns

Service discovery is a key component of most distributed systems and service oriented architectures. The problem seems simple at first: *How do clients determine the IP and port for a service that exist on multiple hosts?*

Usually, you start off with some static configuration which gets you pretty far. Things get more complicated as you start deploying more services. With a live system, service locations can change quite frequently due to auto or manual scaling, new deployments of services, as well as hosts failing or being replaced.

Dynamic service registration and discovery becomes much more important in these scenarios in order to avoid service interruption.

This problem has been addressed in many different ways and is continuing to evolve. We're going to look at some open-source or openly-discussed solutions to this problem to understand how they work. Specifically, we'll look at how each solution uses strong or weakly consistent storage, runtime dependencies, client integration options and what the tradeoffs of those features might be.

We'll start with some strongly consistent projects such as Zookeeper, Doozer and Etcd which are typically used as coordination services but are also used for service registries as well.

We'll then look at some interesting solutions specifically designed for service registration and discovery. We'll examine Airbnb's SmartStack, Netflix's Eureka, Bitly's NSQ, Serf, Spotify and DNS and finally SkyDNS.

The Problem

There are two sides to the problem of locating services. *Service Registration* and *Service Discovery*.

- **Service Registration** - The process of a service registering its location in a central registry. It usually register its host and port and sometimes authentication credentials, protocols, versions numbers, and/or environment details.
- **Service Discovery** - The process of a client application querying the central registry to learn of the location of services.

Any service registration and discovery solution also has other development and operational aspects to consider:

- **Monitoring** - What happens when a registered service fails? Sometimes it is unregistered immediately, after a timeout, or by another process. Services are usually required to implement a heartbeating mechanism to ensure liveness and clients typically need to be able to handle failed services reliably.
- **Load Balancing** - If multiple services are registered, how do all the clients balance the load across the services? If there is a master, can it be deteremined by a client correctly?
- **Integration Style** - Does the registry only provide a few language bindings, for example, only Java? Does integrating require embedding registration and discovery code into your application or is a *sidekick* process an option?
- **Runtime Dependencies** - Does it require the JVM, Ruby or something that is not compatible with your environment?
- **Availability Concerns** - Can you lose a node and still function? Can it be upgraded without incurring an outage? The registry will grow to be a central part of your architecture and could be a single point of failure.

General Purpose Registries

These first three registries use strongly consistent protocols and are actually general purpose, consistent datastores. Although we're looking at them as service registries, they are typically used for coordination services to aid in leader election or centralized locking with a distributed set of clients.

Zookeeper

Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. It's written in Java, is strongly consistent (CP) and uses the Zab protocol to coordinate changes across the ensemble (cluster).

Zookeeper is typically run with three, five or seven members in the ensemble. Clients use language specific bindings in order to access the ensemble. Access is typically embedded into the client applications and services.

Service registration is implemented with ephemeral nodes under a namespace. Ephemeral nodes only exist while the client is connected so typically a backend service registers itself, after startup, with its location information. If it fails or disconnects, the node disappears from the tree.

Service discovery is implemented by listing and watching the namespace for the service. Clients receive all the currently registered services as well as notifications when a service becomes unavailable or new ones register. Clients also need to handle any load balancing or failover themselves.

The Zookeeper API can be difficult to use properly and language bindings might have subtle differences that could cause problems. If you're using a JVM based language, the Curator Service Discovery Extension might be of some use.

Since Zookeeper is a CP system, when a partition occurs, some of your system will not be able to register or find existing registrations even if they could function properly during the partition. Specifically, on any non-quorum side, reads and writes will return an error.

Doozer

Doozer is a consistent, distributed data store. It's written in Go, is strongly consistent and uses Paxos to maintain consensus. The project has been around for a number of years but has stagnated for a while and now has close to 160 forks. Unfortunately, this makes it difficult to know what the actual state of the project is and whether is is suitable for production use.

Doozer is typically run with three, five or seven nodes in the cluster. Clients use language specific bindings to access the cluster and, similar to Zookeeper, integration is embedded into the client and services.

Service registration is not as straightforward as with Zookeeper because Doozer does not have any concept of ephemeral nodes. A service can register itself under a path but if the service becomes unavailable, it won't be removed automatically

There are a number of ways to address this issue. One option might be to add a timestamp and heartbeating mechanism to the registration process and handle expired entries during the discovery process or with another cleanup processes.

Service discovery is similar to Zookeeper in that you can list all the entries under a path and then wait for any changes to that path. If you use a timestamp and heartbeat during registration, you would ignore or delete any expired entries during discovery.

Like Zookeeper, Doozer is also a CP system and has the same consequences when a partition occurs.

Etcd

Etcd is a highly-available, key-value store for shared configuration and service discovery. Etcd was inspired by Zookeeper and Doozer. It's written in Go, uses Raft for consensus and has a HTTP+JSON based API.

Etcd, similar to Doozer and Zookeeper, is usually run with three, five or seven nodes in the cluster. Clients use a language specific binding or implement one using an HTTP client.

Service registration relies on using a key TTL along with heartbeating from the service to ensure the key remains available. If a services fails to update the key's TTL, Etcd will expire it. If a service becomes unavailable, clients will need to handle the connection failure and try another service instance.

Service discovery involves listing the keys under a directory and then waiting for changes on the directory. Since the API is HTTP based, the client application keeps a long-polling connection open with the Etcd cluster.

Since Etcd uses Raft, it should be a strongly-consistent system. Raft requires a leader to be elected and all client requests are handled by the leader. However, Etcd also seems to support reads from non-leaders using this undocumented consistent parameter which would improve availabilty in the read case. Writes would still need to be handled by the leader during a partition and could fail.

Single Purpose Registries

These next few registration services and approaches are specifically tailored to service registration and discovery. Most have come about from actual production use cases while others are interesting and different approaches to the problem. Whereas Zookeeper, Doozer and Etcd could also be used for distributed coordination, these solutions don't have that capability.

Airbnb's SmartStack

Airbnb's SmartStack is a combination of two custom tools, Nerve and Synapse that leverage haproxy and Zookeeper to handle service registration and discovery. Both Nerve and Synapse are written in Ruby.

Nerve is a *sidekick* style process that runs as a separate process alongside the application service. Nerve is reponsible for registering services in Zookeeper. Applications expose a `/health` endpoint, for HTTP services, that Nerve continuously monitors. Provided the service is available, it will be registered in Zookeper.

The *sidekick* model eliminates the need for a service to interact with Zookeeper. It simply needs a monitoring endpoint in order to be registered. This makes it much easier to support services in different languages where robust Zookeeper binding might not exist. This also provides many of benefits of the Hollywood principle.

Synapse is also a *sidekick* style process that runs as a separate process alongside the service. Synapse is responsible for service discovery. It does this by querying Zookeeper for currently registered services and reconfigures a locally running haproxy instance. Any clients on the host that need to access another service always accesses the local haproxy instance which will route the request to an available service.

Synapse's design simplifies service implementations in that they do not need to implement any client side load balancing or failover and they do not need to depend on Zookeepr or its language bindings.

Since SmartStack relies on Zookeeper, some registrations and discovery may fail during a partition. They point out that Zookeepr is their "Achilles heel" in this setup. Provided a service has been able to discover the other services, at least once, before a partition, it should still have a snapshot of the services after the partition and may be able to continue operating during the partition. This aspect improves the availability and reliability of the overall system.

Update: If you're intested in a SmartStack style solution for docker containers, check out docker service discovery.

Netflix's Eureka

Eureka is Netflix's middle-tier, load balancing and discovery service. There is a server component as well as a smart-client that is used within application services. The server and client are written in Java which means the ideal use case would be for the services to also be imlemented in Java or another JVM compatible language.

The Eureka server is the registry for services. They recommend running one Eureka server in each availability zone in AWS to form a cluster. The servers replicate their state to each other through an asynchronous model which means each instance may have a slightly, different picture of all the services at any given time.

Service registration is handled by the client component. Services embed the client in their application code. At runtime, the client registers the service and periodically sends heartbeats to renew its leases.

Service discovery is handled by the smart-client as well. It retrieves the current registrations from the server and caches them locally. The client periodically refreshes its state and also handles load balancing and failovers.

Eureka was designed to be very resilient during failures. It favors availabilty over strong consistency and can operate under a number of different failure modes. If there is a partition within the cluster, Eureka transitions to a self-preservation state. It will allow services to be discovered and registered during a partition and when it heals, the members will merge their state again.

Bitly's NSQ lookupd

NSQ is a realtime, distributed messaging platform. It's written in Go and provides an HTTP based API. While it's not a general purpose service registration and discovery tool, they have implemented a novel model of service discovery in their nsqlookupd agent in order for clients to find nsqd instances at runtime.

In an NSQ deployment, the nsqd instances are essentially the service. These are the message stores. nsqlookupd is the service registry. Clients connect directly to nsqd instances but since these may change at runtime, clients can discover the

available instances by querying nsqlookupd instances.

For service registration, each nsqd instance periodically sends a heartbeat of its state to each nsqlookupd instance. Their state includes their address and any queues or topics they have.

For discovery, clients query each nsqlookupd instance and merge the results.

What is interesting about this model is that the nsqlookupd instances *do not know about each other*. It's the responsibility of the clients to merge the state returned from each stand-alone nsqlookupd instance to determine the overal state. Because each nsqd instance heartbeats its state, each nsqlookupd eventually has the same information provided each nsqd instance can contact all available nsqlookupd instances.

All the previously discussed registry components all form a cluster and use strong or weakly consistent consensus protocols to maintain their state. The NSQ design is inherently weakly consistent but very tolerant to partitions.

Serf

Serf is a decentralized solution for service discovery and orchestration. It is also written in Go and is unique in that uses a gossip based protocol, SWIM for membership, failure detection and custom event propogation. SWIM was designed to address the unscalability of traditional heart-beating protocols.

Serf consists of a single binary that is installed on all hosts. It can be run as an agent, where it joins or creates a cluster, or as a client where it can discover the members in the cluster.

For service registration, a serf agent is run that joins an existing cluster. The agent is started with custom tags that can identify the hosts role, env, ip, ports, etc. Once joined to the cluster, other members will be able to see this host and it's metadata.

For discovery, serf is run with the `members` command which returns the current members of the cluster. Using the members output, you can discover all the hosts for a service based on the tags their agent is running.

Serf is a relatively new project and is evolving quickly. It is the only project in this post that does not have a central registry architectural style which makes it unique. Since it uses a asynchronous, gossip based protocol, it is inherently weakly-consistent but more fault tolerant and available.

Spotify and DNS

Spotify described their use of DNS for service discovery in their post In praise of "boring" technology. Instead of using a newer, less mature technology they opted to build on top of DNS. Spotify views DNS as a "distributed, replicated database tailored for read-heavy loads."

Spotify uses the relatively unknown SRV record which is intended for service discovery. SRV records can be thought of as a more generalized MX record. They allow you to define a service name, protocol, TTL, priority, weight, port and target host. Basically, everything a client would need to find all available services and load balance against them if necessary.

Service registration is complicated and fairly static in their setup since they manage all zone files under source control. Discovery uses a number of different DNS client librarires and custom tools. They also run DNS caches on their services to minimize load on the root DNS server.

They mention at the end of their post that this model has worked well for them but they are starting to outgrow it and are investigating Zookeeper to support both static and dynamic registration.

SkyDNS

SkyDNS is a relatively new project that is written in Go, uses RAFT for consensus and also provides a client API over HTTP and DNS. It has some similarities to Etcd and Spotify's DNS model and actually uses the same RAFT implementation as Etcd, go-raft.

SkyDNS servers are clustered together, and using the RAFT protocol, elect a leader. The SkyDNS servers expose different endpoints for registration and discovery.

For service registration, services use an HTTP based API to create an entry with a TTL. Services must heartbeat their state periodically. SkyDNS also uses SRV records but extends them to also support service version, environment, and region.

For discovery, clients use DNS and retrieve the SRV records for the services they need to contact. Clients need to implement any load balancing or failover and will likely cache and refresh service location data periodically.

Unlike Spotify's use of DNS, SkyDNS does support dynamic service registration and is able to do this without depending on another external service such as Zookeeper or Etcd.

If you are using docker, skydock might be worth checking out to integrate your containers with SkyDNS automatically.

Overall, this is an interesting mix of old (DNS) and new (Go, RAFT) technology and will be interesting to see how the project evolves.

Summary

We've looked at a number of general purpose, strongly consistent registries (Zookeeper, Doozer, Etcd) as well as many custom built, eventually consistent ones (SmartStack, Eureka, NSQ, Serf, Spotify's DNS, SkyDNS).

Many use embedded client libraries (Eureka, NSQ, etc..) and some use separate sidekick processes (SmartStack, Serf).

Interestingly, of the dedicated solutions, all of them have adopted a design that prefers availability over consistency.

Name	Type	AP or CP	Language	Dependencies	Integration
Zookeeper	General	CP	Java	JVM	Client Binding
Doozer	General	CP	Go		Client Binding
Etcd	General	Mixed (1)	Go		Client Binding/HTTP
SmartStack	Dedicated	AP	Ruby	haproxy/Zookeeper	Sidekick (nerve/synapse)
Eureka	Dedicated	AP	Java	JVM	Java Client

NSQ (lookupd)	Dedicated	AP	Go		Client Binding
Serf	Dedicated	AP	Go		Local CLI
Spotify (DNS)	Dedicated	AP	N/A	Bind	DNS Library
SkyDNS	Dedicated	Mixed (2)	Go		HTTP/DNS Library

(1) If using the `consistent` parameter, inconsistent reads are possible

(2) If using a caching DNS client in front of SkyDNS, reads could be inconsistent

Mike Clarke · 5 years ago
`doozer` really shouldn't even be in the conversation here - nobody uses this in production...
38 | Reply · Share ›

bketelsen · 6 years ago
Really nice writeup. Thanks for mentioning SkyDNS and SkyDock, it's clear you took the time to understand all of these solutions before writing on them.
7 | Reply · Share ›

Jason Wilder Mod ➔ bketelsen · 6 years ago
Thanks Brian!
2 | Reply · Share ›

Spud · 6 years ago
http://consul.io is based on serf and released since the article was written.
5 | Reply · Share ›

carlivar ➔ Spud · 6 years ago
This looks most interesting to me due to the high-level features, such as service monitoring.
| Reply · Share ›

.:. brainsik · 5 years ago
Enjoyed this writeup even though it's close to a year old. Please consider doing a 2015 update. It'd be interesting to hear what's happened to the various projects and hear about newcomers.
3 | Reply · Share ›

Joseph Lee · 3 years ago
I mostly agree with your points... Except that F# changes everything : you get a highly polished language, packed with 30years of ML goodness, an open source community. But overall it is recommended.
| Reply · Share ›

Tiantian Gao · 4 years ago
That article is great, give all I need to know about service discovery
| Reply · Share ›

Moayad Abu Jaber · 4 years ago
its a nice article, I have couple of question. first one is this really support or implement the SOA principle (http://serviceorientation.c...
the second question, as what I see there is no need for invest in expensive load balancer like F5 we should start things to use dynamic service discovery in our environments.
| Reply · Share ›

Moayad Abu Jaber ➔ Moayad Abu Jaber · 4 years ago
I think the article written be airbnb for for smartstack it answer my questions. http://nerds.airbnb.com/sma...
| Reply · Share ›

Nanduni Indeewaree · 4 years ago
Very nice article. Thanks a lot. It was very useful.
| Reply · Share ›

vlan · 4 years ago
JINI, a distributed Java component technology framework created by Sun around 1999 continues its existence as Apache River. I suggest everybody to take a look at it. This technology has everything since long ago other ones are just trying to reinvent. Check out Rio (www.rio-project.org) a container system for JINI services that makes a piece of cake to develop, deploy and manage those services.
| Reply · Share ›

Luca · 4 years ago · edited
I think you missed Consul though... - thanks anyway
--

Edit 1:
i've just noticed the post below: Consul was not available at the time of writing...
| Reply · Share ›

unniwarrier · 4 years ago
Great roundup, thanks. Gives me a comprehensive look at this landscape. Suggest keeping this post updated as we go along.
| Reply · Share ›

JMSimpson · 4 years ago
There is a new discovery protocol that the US DoD is working on that is also open sourced. It's more similar to Bonjour (mDNS), WS-Discovery and SSDP, but the basic use-case is the same. The project is called Argo (www.argo.ws) and it is available on Github. The main alternative use-case for the Argo protocol is Network-Based Moving Target Defense against cyber attacks.
| Reply · Share ›

Peter Salas · 5 years ago
Exceptional article! It's amazing the progress the industry has made in solving highly specialized and distributed applications. Personally, I've worked with a model close to Netflix's Eureka was much success; but definitely saw some other interesting approaches that have different pro's and con's.
| Reply · Share ›

Andrew Pennebaker · 5 years ago · edited
Proper use of "its" vs "it's":

* "registering it's location" should be "registering its location"
* "with it's location" should be "with its location"
* "Zookeepr or it's language bindings" should be ""Zookeeper or its language bindings""
* "renew it's leases" should be "renew its leases"
* "refreshes it's state" should be "refreshes its state"
* "heartbeat of it's state" should be "heartbeat of its state"

Also:

* "expose a /heath endpoint" should be "expose a /health endpoint"
* "to each of nsqlookupd instance" should be "to each nsqlookupd instance"

Reply · Share ›

Richard · 5 years ago

"it's" -> "its", in many places. The first means "it is." The second is a genitive (possessive) pronoun.

Reply · Share ›

Chongqing Zhao · 5 years ago · edited

useful

Reply · Share ›

Curtis · 5 years ago

Well done, Jason! This was a very informative read for me. BTW, I know it is 10 months later, but it looks like SkyDNS 2 uses etcd.

Reply · Share ›

Piyush Kansal · 5 years ago

Thanks for the nice writeup. Quite crisp and informative.

Reply · Share ›

Bobo Lin · 5 years ago · edited

really really good article , thanks . skydns and serf looks pretty good , smartstack also great , but seems heavy on client side

Reply · Share ›

lifei · 5 years ago

Thanks Brian!

Reply · Share ›

Sheldon Hearn · 5 years ago

See also DDSL: Dynamic distributed service registration and location library (Scala) and CLI tool, using ZooKeeper as data store, but no registry service component. https://github.com/mbknor/ddsl

Reply · Share ›

Amin Jams · 6 years ago

Thank you Jason. Really good write up. It cleared a lot of questions I had.

Reply · Share ›

Omer Katz · 6 years ago

What about Consul?

Reply · Share ›

beier · 6 years ago

Thanks for the summary Jason! I've been following Serf for a while and really liked it' truely distributed model. In most of the cases service R/D don't need consistency but more for availability. Looking forward to a production ready version of serf!.

Reply · Share ›

Vincent Bernat · 6 years ago · edited

Interesting survey. It seems that the integration can be quite difficult if you have many services. One possible solution (that I have contributed to) is to use Zookeeper + zkfarmer (https://github.com/rs/zkfarmer). It uses the local filesystem to interact with zookeeper. You write a value in the local FS and it is replicated to zookeeper. If a value change in zookeeper, it is replicated in the local filesystem (in a PHP file for example). Unfortunately, it is not (yet) able to execute a script on change (which is a useful feature of Serf to react to events).

Reply · Share ›

Jason Wilder Mod ➔ Vincent Bernat · 6 years ago

Nice! I had not seen that before. Reminds me of https://github.com/kelseyhi... for etcd.

Reply · Share ›

Guest · 6 years ago

Hi Jason, Thanks for sharing such a valuable analysis, great article! I've been doing some research at the moment for improvement of in-house Discovery Service on my project. We have a number of nodes in a cluster accountable for discovery service, higly available. In order to get some service each client app sends a multicast message to all these nodes in the cluster. The very first response defines a particular node to work with. This is an overhead and I'm thinking to use some kind of leader election algorithm where only a single leader responds. What do you think?

Reply · Share ›

Jason Wilder Mod ➔ Guest · 6 years ago

That sounds somewhat similar in concept to NATS: http://blog.gopheracademy.c....

Reply · Share ›