Extracted from:

Distributed Services with Go

Your Guide to Reliable, Scalable, and Maintainable Systems

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Distributed Services with Go

Your Guide to Reliable, Scalable, and Maintainable Systems

Travis Jeffery

edited by Dawn Schanafelt and Katharine Dvorak

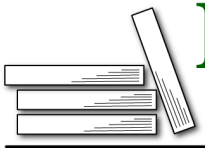
Distributed Services with Go

Your Guide to Reliable, Scalable, and Maintainable Systems

Travis Jeffery

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Dawn Schanafelt and Katharine Dvorak
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-760-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2021

Why Use Service Discovery?

Service discovery is the process of figuring out how to connect to a service. A service discovery solution must keep an up-to-date list (also known as a *registry*) of services, their locations, and their health. Downstream services then query this registry to discover the location of upstream services and connect to them—for example, a web service discovering and connecting to its database. This way, even if the upstream services change (scale up or down, or get replaced), downstream services can still connect to them.

In the pre-cloud days, you could set up "service discovery" with manually managed and configured static addresses, which was workable since applications ran on static hardware. Today, service discovery plays a big part in modern cloud applications where nodes change frequently.

Instead of using service discovery, some developers put load balancers in front of their services so that the load balancers provide static IPs. But for server-to-server communication, where you control the servers and you don't need a load balancer to act as a trust boundary[1] between clients and servers, use service discovery instead. Load balancers add cost, increase latency, introduce single points of failure, and need updates as services scale up and down. If you manage tens or hundreds of microservices, then not using service discovery means you also have to manage tens or hundreds of load balancers and DNS records. For a distributed service like ours, using a load balancer would force us to depend on a load-balancer service like nginx or the various cloud load balancers like AWS's ELB or Google Cloud's Load Balancer. This would increase our operational burden, infrastructure costs, and latency.

In our system, we have two service-discovery problems to solve:

- How will the servers in our cluster discover each other?
- How will the clients discover the servers?

In this chapter, we'll work on implementing the discovery for the servers. Then, after we implement consensus in Chapter 8, Coordinate Your Services with Consensus, on page ?, we'll work on the clients' discovery in Chapter 9, Discover Servers and Load Balance from the Client, on page ?.

Now that you know what service discovery can do, we're ready to embed it into our service.

1. https://en.wikipedia.org/wiki/Trust_boundary

Embed Service Discovery

When you have an application that needs to talk to a service, the tool you use for service discovery needs to perform the following tasks:

- Manage a registry of services containing info such as their IPs and ports;
- Help services find other services using the registry;
- Health check service instances and remove them if they're not well; and
- Deregister services when they go offline.

Historically, people who've built distributed services have depended on separate, stand-alone services for service discovery (such as Consul, ZooKeeper, and Etcd). In this architecture, users of your service run two clusters: one for your service and one for your service discovery. The benefit of using a service-discovery service is that you don't have to build service discovery yourself. The downside to using such a service, from your users' standpoint, is that they have to learn, launch, and operate an extra service's cluster. So using a stand-alone service for discovery removes the burden from your shoulders and puts it on your users'. That means many users won't use your service because the burden is too much for them, and users who *do* take it on won't recommend your service to others as often or as highly.

So why did people who built distributed services use stand-alone service-discovery services, and why did their users put up with the extra burden? Because neither had much of a choice. The people building distributed services didn't have the libraries they needed to embed service discovery into their services, and users didn't have other options.

Fortunately, times have changed. Today, Gophers have Serf—a library that provides decentralized cluster membership, failure detection, and orchestration that you can use to easily embed service discovery into your distributed services. Hashicorp, the company that created it, uses Serf to power its own service-discovery product, Consul, so you're in good company.

Using Serf to embed service discovery into your services means that you don't have to implement service discovery yourself and your users don't have to run an extra cluster. It's a win-win.

When to Depend on a Stand-Alone Service-Discovery Solution

You may encounter cases where depending on a stand-alone service for service discovery makes sense—for example, if you need to integrate your service discovery with many platforms. You sink a lot of effort into that kind of work, and that's likely a poor use

When to Depend on a Stand-Alone Service-Discovery Solution

of your time when you could just use a service like Consul that provides those integrations. In any case, Serf is always a good place to start. Once you've developed your service to solve the core problem it's targeting and your service is stable or close to it, then you will have a good sense of whether you need to depend on a service-discovery service.

Here are some other benefits of building our service with Serf:

- In the early days of building a service, Serf is faster to set up and build our service against than having to set up a separate service.
- It's easier to move from Serf to a stand-alone service than to move from a stand-alone service to Serf, so we still have both options open.
- Our service will be easier and more flexible to deploy, making our service more accessible.

So for our service, we'll use Serf to build service discovery.

Now that we've seen the benefits of using Serf, let's quickly discuss how Serf does its thing.