SmartStack: Service Discovery in the Cloud

What is SmartStack?

SmartStack is an automated service discovery and registration framework. It makes the lives of engineers easier by transparently handling creation, deletion, failure, and maintenance work of the machines running code within your organization. We believe that our approach to this problem is among the best possible: simpler conceptually, easier to operate, more configurable, and providing more introspection than any of its kind. The SmartStack way has been battle-tested at Airbnb over the past year, and has broad applicability in many organizations, large and small.

SmartStack's components – Nerve and Synapse – are available on GitHub! Read on to learn more about the magic under the hood.

The problem of services in an SOA

Companies like Airbnb often start out as monolithic applications – a kind of swiss army knife which performs all of the functions of the organization. As traffic (and the number of engineers working on the product) grows, this approach doesn't scale. The code base becomes too complicated, concerns are not cleanly separated, changes from many engineers touching many different parts of the codebase go out together, and performance is determined by the worst-performing sections in the application.

The solution to this problem is services: individual, smaller code bases, running on separate machines with separate deployment cycles, that more cleanly address more targeted problem domains. This is called a services-oriented architecture: SOA.

As you build out services in your architecture, you will notice that instead

of maintaining a single pool of general-purpose application servers, you are now maintaining many smaller pools. This leads to a number of problems. How do you direct traffic to each machine in the pool? How do you add new machines and remove broken or retired ones? What is the impact of a single broken machine on the rest of the application?

Dealing with these questions, across a collection of several services, can quickly grow to be a full-time job for several engineers, engaging in an arduous, manual, error-prone process fraught with peril and the potential for downtime.

The properties of an ideal solution

The answer to the problem of services is automation – let computers do the dirty work of maintaining your backend pools. But there are many ways to implement such automation. Let's first outline the properties of an ideal implementation:

- Backends capable of serving requests for a particular service would automatically receive these requests
- Load would be intelligently distributed among backends, so no backend is ever doing more work than the rest and so that requests are automatically routed to the least busy backend
- A backend that develops problems would automatically stop receiving traffic
- It should be possible to remove load from a backend without affecting anything else in the system, so that debugging is possible
- There should be perfect introspection, so that you always know which backends are available, what kind of load they are receiving, and from which consumers
- Consumers should be minimally affected by changes in the list of backends; ideally, such changes would be transparent to consumers
- There should be no single points of failure – the failure of any machine anywhere in the system should have no impact

We can use these criteria to evaluate potential solutions. For instance, manually changing a list of backends which has been hard-coded in a client, and then deploying the client, immediately fails many of these criteria by wide margins. How do some other approaches stack up?

Solutions which don't work

Many commonly-used approaches to service discovery don't actually work very well in practice. To understand why SmartStack works so well, it can be helpful to first understand why other solutions do not.

DNS

The simplest solution to registration and discovery is to just put all of your backends behind a single DNS name. To address a service, you contact it by DNS name and the request should get to a random backend.

The registration component of this is fairly well-understood. On your own infrastructure, you can use dynamic DNS backends like BIND-DLZ for registration. In the cloud, with hosted DNS like Route53, simple API calls suffice. In AWS, if you use round-robin CNAME records you would even get split horizon for free, so the same records would work from both inside and outside AWS.

However, using DNS for service discovery is fraught with peril. First, consumers have to poll for all changes – there's no way to push state. Also, DNS suffers from propagation delays; even after your monitoring detects a failure and issues a de-registration command to DNS, there will be at least a few seconds before this information gets to the consumers. Worse, because of the various layers of caching in the DNS infrastructure, the exact propagation delay is often non-deterministic.

When using the naive approach, in which you just address your service by name, there's no way to determine which boxes get traffic. You get the

equivalent of random routing, with loads chaotically piling up behind some backends while others are left idle.

Worst of all, many applications cache DNS resolution once, at startup. For instance, Nginx will cache the results of the initial name resolution unless you use a configuration file hack. The same is true of HAProxy. Figuring out that your application is vulnerable can be costly, and fixing the problem harder still.

Note that we are here referring to native use of DNS by client libraries. There are ways to use the DNS system more intelligently to do service discovery – where we use Zookeeper in SmartStack, we could use DNS too without losing too much functionality. But simply making an HTTP request to `myservice.mydomain.com` via an HTTP library does not work well.

Central load balancing

If you're convinced that DNS is the wrong approach for service discovery, you might decide to take the route of centralizing your service routing. In this approach, if `service a` wants to talk to `service b`, it should talk to a load balancer, which will properly route the request. All services are configured with the method of finding the load balancer, and the load balancer is the only thing that needs to know about all of the backends.

This approach sounds promising, but in reality it doesn't buy you much. First, how do your services discover the load balancer? Often, the answer is DNS, but now you've introduced more problems over the DNS approach than you've solved – if your load balancer goes down, you are still faced with all of the problems of service discovery over DNS. Also, a centralized routing layer is a big fat point of failure. If that layer goes down, everything else goes with it. Reconfiguring the load balancer with new backends – a routine operation – becomes fraught with peril.

Next, what load balancer do you choose? In a traditional data center, you might be tempted to reach for a couple of hardware devices like an F5. But in the cloud this is not an option. On AWS, you might be tempted to use ELB, but ELBs are terrible at internal load balancing because they only have public IPs. Traffic from one of your servers to another would have to leave your private network and re-enter it again. Besides introducing latency, it wreaks havoc on security groups. If you decide to run your own load balancing layer on EC2 instances, you will end up just pushing the problem of service discovery one layer up. Your load balancer is now no longer a special, hardened device; it's just another instance, just as prone to failure but just especially critical to your operations.

In-app registration/discovery

Given the problems of DNS and central load balancing, you might decide to just solve the problem with code. Instead of using DNS inside your app to discover a dependency, why not use some different, more specialized mechanism?

This is a popular approach, which is found in many software stacks. For instance, Airbnb initially used this model with our Twitter commons services. Our java services running on Twitter commons automatically registered themselves with zookeeper, a central point of configuration information which replaces DNS. Apps that wanted to talk to those services would ask Zookeeper for a list of available backends, with periodic refresh or a subscription via zookeeper watches to learn about changes in the list.

However, even this approach suffers from a number of limitations. First, it works best if you are running on a unified software stack. At Twitter, where most services run on the JVM, this is easy. However, at Airbnb, we had to implement our own zookeeper client pool in ruby in order to communicate with ZK-registered services. The final implementation was not well-hardened, and resulted in service outages whenever Zookeeper was down.

Worse, as we began to develop more services for stacks like Node.js, we foresaw ourselves being forced to implement the same registration and discovery logic in language after language. Sometimes, the lack or immaturity of relevant libraries would further hamper the effort. Finally, sometimes you would like to run apps which you did not write yourself but still have them consume your infrastructure: Nginx, a CI server, rsyslog or other systems software. In these cases, in-app discovery is completely impossible.

Finally, this approach is very difficult operationally. Debugging an app that registers itself is impossible without stopping the app – we've had to resort to using iptables to block the Zookeeper port on machines we were investigating. Special provisions have to be made to examine the list of backends that a particular app instance is currently communicating with. And again, intelligent load balancing is complicated to implement.

The SmartStack way

SmartStack is our solution to the problem of SOA. SmartStack works by taking the whole problem out-of-band from your application, in effect abstracting it away. We do this with two separate applications that run on the same machine as your app: Nerve, for service registration, and Synapse, for service discovery.

Nerve

Nerve does nothing more complicated than what Twitter commons already did. It creates ephemeral nodes in Zookeeper which contain information about the address/port combos for a backend available to serve requests for a particular service.

In order to know whether a particular backend can be registered, Nerve performs health checks. Every service that you want to register has a list of health checks, and if any of them fail the backend is de-registered.

Although a health check can be as simple as "is this app reachable over TCP or HTTP", properly integrating with Nerve means implementing and exposing a health check via your app. For instance, at Airbnb, every app that speaks HTTP exposes a `/health` endpoint, which returns a `200 OK` if the app is healthy and a different status code otherwise. This is true across our infrastructure; for instance, try visiting https://www.airbnb.com/health in your browser!

For non-HTTP apps, we've written checks that speak the particular protocol in question. A redis check, for instance, might try to write and then read a simple key. A rabbitmq check might try to publish and then consume a message.

Synapse

Synapse is the magic behind service discovery at Airbnb. It runs beside your service, and handles making your service dependencies available to use, transparently to your app. Synapse reads the information in Zookeeper for available backends, and then uses that information to configure a local HAProxy process. When a client wants to talk to a service, it just talks to the local HAProxy, which takes care of properly routing the request.

This is, in essence, a decentralized approach to the centralized load load balancing solution discussed above. There's no bootstrapping problem because there's no need to discover the load balancing layer – it's always present on localhost.

The real work is performed by HAProxy, which is extremely stable and battle-tested. The Synapse process itself is just an intermediary – if it goes down, you will not get notification about changes but everything else will still function. Using HAProxy gives us a whole host of advantages. We get all of the powerful logging and introspection. We get access to very advanced load-balancing algorithms, queueing controls, retries, and

timeouts. There is built-in health checking which we can use to guard against network partitions (where we learn about an available backend from Nerve, but can't actually talk to it because of the network). In fact, we could probably fill an entire additional blog post just raving about HAProxy and how we configure it.

Although Synapse writes out the entire HAProxy config file every time it gets notified about changes in the list of backends, we try to use HAProxy's stats socket to make changes when we can. Backends that already exist in HAProxy are put into maintenance mode via the stats socket when they go down, and then recovered when they return. We only have to restart HAProxy to make it re-read its config file when we have to add an entirely new backend.

The benefits of SmartStack

SmartStack has worked amazingly well here at Airbnb. To return to our previous checklist, here is how we stack up:

- Within a health check interval's delay of a backend becoming healthy, it is made available in Zookeeper; this makes it instantly available to consumers via Synapse's Zookeeper watches.
- We detect problems within a health check interval, and take backends out of rotation. A mechanism which allows services to notify Nerve that they're not healthy is planned, to reduce the interval further. In the meantime, deploys can stop Nerve when they start, and then re-start it at the end.
- Synapse acts on information the moment it's published in Zookeeper, and reconfiguring HAProxy is very very fast most of the time. Because we utilize HAProxy's stats socket for many changes, we don't even restart the process unless we have to add new backends.
- Because our infrastructure is distributed, we cannot do centralized planning. But HAProxy provides very configurable queueing semantics. For our biggest clients, we set up intelligent queueing at

the HAProxy layer; for others, we at least guarantee round-robin.
- Doing debugging or maintenance on a backend is as simple as stopping the Nerve process on the machine; nothing else is affected!
- You can see exactly which backends are available simply by looking at the HAProxy status page. Because of HAProxy's excellent log output, you also get amazing aggregate and per-request information, including statistics on number of behavior of requests right in rsyslog.
- The infrastructure is completely distributed. The most critical nodes are the Zookeeper nodes, and Zookeeper is specifically designed to be distributed and robust against failure.

After using SmartStack in production at Airbnb for the past year, we remain totally in love with it. It's simple to use, frees us from having to think about many problems, and easy to debug when something goes wrong.

Building on SmartStack

Besides the basic functionality of SmartStack – allowing services to talk to one another – we've built some additional tooling directly on top of it.

Charon

Charon is Airbnb's front-facing load balancer. Previous to Charon, we used Amazon's ELB. However, ELB did not provide us with good introspection into our production traffic. Also, the process of adding and removing instances to the ELB was clunky – we had a powerful service discovery framework, and then we had a second one just for front-facing traffic. Finally, because ELB is based on EBS, we were worried about the impact of another EBS outage.

We were already putting Nginx as the first thing behind the ELB. This is because our web traffic is split between several user-facing apps – https://www.airbnb.com is not served by the same servers as

https://www.airbnb.com/help, for example. With Charon, traffic from Akamai hits one of a few Nginx servers directly. Each of these servers is running Synapse, with the user-facing services enabled. As we match the URI against locations, traffic is sent, via the correct HAProxy for that service, to the optimal backend to handle that request.

In effect, Charon makes routing from users to our user-facing services just like routing between backend services. We get all of the same benefits – health checks, the instant addition and removal of instances, and correct queueing behavior via HAProxy.

Dyno

What we do for user-facing services, we also do for internal services like dashboards or monitors. Wild card DNS sends all *.dyno requests to a few dyno boxes. Nginx extracts the service name from the `Host` header and then directs you to that service via Synapse and HAProxy.

Future Work

The achilles heel of SmartStack is Zookeeper. Currently, the failure of our Zookeeper cluster will take out our entire infrastructure. Also, because of edge cases in the zk library we use, we may not even be handling the failure of a single ZK node properly at the moment.

Most recent work has been around building better testing infrastructure around SmartStack. We can now spin up an integration test for the entire platform easily with our Chef cookbook and Vagrant. We hope that with better testing, we can make headway on making SmartStack even more resilient.

We are also considering adding dynamic service registration for Nerve. Currently, the discovery that Nerve performs is hardcoded into Nerve via its config file. In practice, because we manage our infrastructure with

configuration management (Chef) this is not a huge problem for us. If we deploy a new service on a machine, the Chef code that deploys the service also updates the Nerve config.

However, in environments like Mesos, where a service might be dynamically scheduled to run on a machine at any time, it would be awesome to have the service easily register itself with Nerve for health checking. This introduces tighter coupling between the services and SmartStack than we currently have, so we are thinking carefully through this approach to make sure we build the correct API.

Give it a shot!

We would love for SmartStack to prove it's value in other organizations. Give it a shot, and tell us how it works for you!

Probably the easiest way to get started is by using the Chef cookbook. The code there can create a comprehensive test environment for SmartStack and run it through automated integration tests. It is also the code we use to run SmartStack in production, and provides what we think is a solid approach to managing SmartStack configuration. There is also excellent documentation on running and debugging SmartStack included with the cookbook.

If you would like to configure Nerve and Synapse yourself, check out documentation in their respective repositories. We've included example config files to get you started with each.

TL;DR

Scaling a web infrastructure requires services, and building a service-oriented infrastructure is hard. Make it EASY, with SmartStack's automated, transparent service discovery and registration: cruise control for your distributed infrastructure.

Want to work with us? ***We're hiring!***

View Open Positions