Practical Perl Tools
Seek Wise Consul

DAVID N. BLANK-EDELMAN

David Blank-Edelman is the Technical Evangelist at Apcera (the comments/views here are David's alone and do not represent Apcera/Ericsson) . He has spent close to 30 years in the systems administration/DevOps/SRE field in large multiplatform environments including Brandeis University, Cambridge Technology Group, MIT Media Laboratory, and Northeastern University. He is the author of the O'Reilly Otter book *Automating System Administration with Perl* and is a frequent invited speaker/organizer for conferences in the field. David is honored to serve on the USENIX Board of Directors. He prefers to pronounce Evangelist with a hard 'g'. dnblankedelman@gmail.com

As we build infrastructure that is more and more fluid, the idea of "service discovery" becomes more and more important. Once upon a time, service discovery was trivial. If you wanted to figure out where a particular service resided so you could tell some other thing in your environment how to talk to it, you could walk over to a machine in one of your racks of equipment, read the labels, and point at it. It likely had a fixed IP address in addition to a fixed physical address.

Those halcyon days (which I'm not sure even I am pining for) are almost gone. With the onset of one or any of a list of factors like virtualization, containerization and "cloudy" provisioning, components in a system and the people running them have a much more difficult time locating each other on the fly. There are a number of different good approaches/tools for this problem. Today we are going to look at the open source package Consul from HashiCorp (https://www.consul.io/—you've probably heard of them because they are the makers of Vagrant) and how to interact with it via Perl. If this topic is popular, I'm happy to look at some of the other choices in a future column.

Basic Consul Concepts

The heart of most of these solutions is some sort of distributed/highly available key-value store that provides easy methods for your other infrastructure components to query/interact with it. "Distributed/highly available" in this case means that the package makes it easy to run multiple servers that replicate data between themselves so that if one goes down, your infrastructure continues to hum along. This requires handling all of the gnarly details around this sort of setup (what to do when one goes down and comes back up again with stale data, how to handle network partitions, where are writes handled in the system, deciding on the fly which instance should be "master" and which should be replicas, etc.). The "key-value store" part of the first sentence is not much more complicated than the key-value concept of a Perl hash, so we should feel right at home when we get to working with that portion of Consul.

In practice what this means is that you run a number of Consul servers and Consul agents. The servers are, well, servers. The agents run on or with every service you wish to make discoverable. Their job is to report in to the servers. They do a little bit more than just register their respective service (which you can do with the Perl modules we're going to be discussing). They can perform health checks on your service and register/deregister it from Consul as appropriate when it becomes operational or sick. They also provide query forwarding so Consul queries can be made of both the servers or the agents (who will automatically forward to an appropriate server). These queries can either be via HTTP, or, to make things really easy, via DNS. If you haven't seen this sort of pure magic before, it entails just having the thing that wants to find that service make a DNS query for the right host name (as in {servicename}.service.consul). Super spiffy.

(Not So) Secret Agent Man

As much as I want to dive directly into the Perl part of all of this, I think it will help if we first start out interacting with the system using the built-in agent functionality before we bring in Perl as an external actor. And while I'm making caveats, I'm not really going to demonstrate any of the functionality around health checks or clustering because it basically just works as the doc suggests and is almost orthogonal to the later discussions around Perl. I'm also not going to discuss installation issues—they are covered in the excellent doc at https://www.consul.io/.

So let's look at perhaps the simplest example of using Consul with a single agent (that will also act as a server). This example comes only slightly modified from the Getting Started documentation. There are two ways to tell a Consul agent about a service that it should advertise: through a config file or via the API. Let's do it both ways.

For a config file, we just need to drop in place a tiny JSON file that looks like this:

```
{
  "service": {
    "name": "webserver"
    "port": 80
  }
}
```

and start up the agent in "dev" mode:

```
consul agent -dev -config-dir ./config.d
```

The agent spins up, loads the config, decides it should become a lead server, and is ready to answer requests (I'm leaving all of the output from the somewhat chatty dev mode in place just because I think the election stuff looks cool):

```
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
        Node name: 'dNb-MBP.local'
       Datacenter: 'dc1'
           Server: true (bootstrap: false)
      Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1,
                                  DNS: 8600, RPC: 8400)
     Cluster Addr: 172.27.4.103 (LAN: 8301, WAN: 8302)
   Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
          Atlas: <disabled>

==> Log data will now stream in as it occurs:

    2016/06/29 17:32:29 [INFO] raft: Node at 172.27.4.103:8300
 [Follower] entering Follower state
    2016/06/29 17:32:29 [INFO] serf: EventMemberJoin:
dNb-MBP.local 172.27.4.103
    2016/06/29 17:32:29 [INFO] serf: EventMemberJoin:
dNb-MBP.local.dc1 172.27.4.103
    2016/06/29 17:32:29 [INFO] consul: adding WAN server
dNb-MBP.local.dc1 (Addr: 172.27.4.103:8300) (DC: dc1)
    2016/06/29 17:32:29 [INFO] consul: adding LAN server
dNb-MBP.local (Addr: 172.27.4.103:8300) (DC: dc1)
    2016/06/29 17:32:29 [ERR] agent: failed to sync remote
state: No cluster leader
    2016/06/29 17:32:31 [WARN] raft: Heartbeat timeout
reached, starting election
    2016/06/29 17:32:31 [INFO] raft: Node at
172.27.4.103:8300 [Candidate] entering Candidate state
    2016/06/29 17:32:31 [DEBUG] raft: Votes needed: 1
    2016/06/29 17:32:31 [DEBUG] raft: Vote granted from
172.27.4.103:8300. Tally: 1
    2016/06/29 17:32:31 [INFO] raft: Election won. Tally: 1
    2016/06/29 17:32:31 [INFO] raft: Node at 172.27.4.103:8300
[Leader] entering Leader state
    2016/06/29 17:32:31 [INFO] raft: Disabling
EnableSingleNode (bootstrap)
    2016/06/29 17:32:31 [INFO] consul: cluster leadership
acquired    2016/06/29 17:32:31 [DEBUG] raft: Node
172.27.4.103:8300 updated peer set (2): [172.27.4.103:8300]

    2016/06/29 17:32:31 [DEBUG] consul: reset tombstone
GC to index 2
    2016/06/29 17:32:31 [INFO] consul: New leader elected:
dNb-MBP.local
    2016/06/29 17:32:31 [INFO] consul: member 'dNb-MBP.local'
joined, marking health alive
    2016/06/29 17:32:32 [INFO] agent: Synced service 'consul'
    2016/06/29 17:32:32 [INFO] agent: Synced service
'webserver'
```

We can then query it either via DNS:

```
$ dig @127.0.0.1 -p 8600 webserver.service.consul

; <<>> DiG 9.8.3-P1 <<>> @127.0.0.1 -p 8600 webserver.service.
consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36403
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0,
ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;webserver.service.consul.  IN  A

;; ANSWER SECTION:
webserver.service.consul. 0 IN  A 172.27.4.103
```

or via a call to the API:

```
$ curl -s http://localhost:8500/v1/catalog/service/
webserverljq -M
[
  {
    "Node": "dNb-MBP.local",
    "Address": "172.27.4.103",
    "ServiceID": "webserver",
    "ServiceName": "webserver",
    "ServiceAddress": "",
    "ServicePort": 80,
    "ServiceEnableTagOverride": false,
    "CreateIndex": 5,
    "ModifyIndex": 5
  }
]
```

We can also register a service via the API:

```
$ curl -X POST -d @newservice.json http://localhost:8500/v1/
agent/service/register --header "Content-Type:application/json"
```

This does an HTTP POST of the JSON file with this content:

```
{
  "ID": "webconsole",
  "Name": "webconsole",
  "Port": 8080
}
```

And now we can see it in a query (output excerpted):

```
$ dig @127.0.0.1 -p 8600 webconsole.service.consul
' ;; ANSWER SECTION:
webconsole.service.consul. 0 IN  A 172.27.4.103
```

Key-Value Time

In addition to just being able to register and query service records, Consul also offers the more general key-value store functionality à la etcd or zookeeper. HTTP API calls can be made to store/retrieve a value under a certain key. This is useful if you want to also store some configuration info in Consul. A component can come up, check in with Consul, and get not only pointers to the services it might need but also its configuration values. Consul's key-value support is more sophisticated than what Perl's hashes offer, so perhaps the analogy earlier was a little simplistic. In addition to the usual GET/SET operations, it offers transactions (multiple operations at once that either succeed or fail all together), locks, test-before-set and watch for changes functionality.

Simple work with keys just involves PUT or GET operations to the /v1/kv/<key> endpoint. We could do this via curl commands (similar to exactly what we did before), but better yet, let's use this as an excuse to get into the land of Perl.

Perl Meet Consul

One of the purposes of showing all of these command lines is to demonstrate the ease of interacting with the system. Translating these operations directly to an equivalent generic Perl module would be easy:

- curl calls easily become HTTP::Tiny (or whatever the HTTP module of choice is for you) calls.
- DNS queries are easily done via Net::DNS. The only tricky thing here would be to make sure you specify either the port option to Net::DNS::Resolver->new() or be sure to set the port via the port() method to the port that Consul is using.

We've used this stuff time and time again in the past, so instead let's take a quick look at the more customized modules for Consul. The two I'm aware of are "Consul" and "Consul::Simple." The one thing Consul::Simple has that I think is kind of neat (and possible to easily implement while using the other module) is the ability to set a prefix for key-value operations. As the doc says, this essentially gives you "namespaces," meaning you could have keys named "namespace/thing" so you could use different namespaces for different kinds of keys (e.g., "dev/something," "prod/something," etc.). Unfortunately, Consul::Simple hasn't been touched in a couple of years, so I'm going to focus on the module just called "Consul."

You'll be pleased and I suspect unsurprised to know that using this module looks just like the previous command line operations only simpler. Here's how we might set and retrieve key/values:

```
use Consul;

# talk to localhost by default
my $consul = Consul->kv;

# set some values
my $status;
$status = $consul->put( 'lisa2016' => 'boston' );
die "1st put failed" unless defined $status;
$consul->put( 'sreconEU2016' => 'dublin' );
die "2nd put failed" unless defined $status;

# retrieve them
my $response = $consul->get('lisa2016');
print 'LISA is in ' . $response->value . " in 2016\n";
$response = $consul->get('sreconEU2016');
print 'SREconEU is in ' . $response->value . " in 2016\n";

# let's try the namespaces idea
$status = $consul->put( 'conferences/2017/lisa2017'
  => 'san francisco' );
$status = $consul->put( 'conferences/2017/srecon2017'
   => 'san francisco' );
```

```
# returns all of the matching keys
my $pairs = $consul->get_all('conferences/2017');

foreach my $pair (@$pairs) {
   print $pair->key, "\n";
}
```

I'm hoping that the code above is fairly self-explanatory. We no longer have to worry about HTTP endpoints; the module is taking care of all that for us. The other endpoints in the API are also equally available should we want to register or retrieve services, or receive or change internal Consul configuration (for example, around clustering).

I'd encourage you to play with Consul and all it can do. Take care, and I'll see you next time.