Open in app

David Curran

1 Follower · About Follow

Microservices running with Docker and Consul for service discovery

D David Curran Oct 9, 2018 · 7 min read

Previously I used Kubernetes to build a simple microservices architecture that would populate a MongoDB instance with generated data and allow users to consume that data through a Django based API. I recently became aware of Consul and the powerful capabilities it has so I thought I could re-implement the same project using this.

What is Consul?

https://www.consul.io/intro/index.html gives the best explanation and there’s no point in parroting it here.

> *"Consul is a service mesh solution providing a full featured control plane with service discovery, configuration, and segmentation functionality"*

This is the primary function I'm using it for, service discovery. This lets me build a cluster of containers and not worry about what ports they make available as Consul can take care of that.

Scope

The scope of this project is to combine many technologies to build a scalable and reliable containerised application with automated service discovery. To do this I will also leverage two other services, Registrator and Nginx reverse proxy.

Registrator attaches itself to the Docker socket allowing it to see when a container changes state. It can then update Consul (or other similar service) about this change. Nginx acts as a proxy to send traffic for the micro-services to an available back end.

The Environment

I have 2 CoreOS virtual machines running on the same network as part of a cluster. Running CoreOS means I have just about everything I need to run containers already set up. I use a load balancing service available from the service provider to direct traffic to these nodes.

Consul

We'll start with Consul and how to set this up. There are many ways to get a Consul cluster up and running such as in single node mode and dev mode. I am using a cluster method so that I can have scalability and reliability.

Consul is readily available as a Docker container and there are a couple of well used options from Docker Hub. I am using progrium/consul as it has some options to help with setup, including giving you copy/paste commands to run on each node.

On node1 then node2:

```
docker run --rm progrium/consul cmd:run 172.28.128.3 -d -v /mnt:/data

docker run --rm progrium/consul cmd:run 10.0.0.203:10.0.0.14 -d -v
/mnt:/data
```

We need to configure the first host by running the output of the first command, make sure that the option `-bootstrap-expect` is set to the number of hosts you want in the cluster. Expose ports on the server internal IP except for port 53 to keep the DNS service on the Docker IP.

```
docker run --name node1 -h node1 \
-p 10.0.0.14:8300:8300 \
-p 10.0.0.14:8301:8301 \
-p 10.0.0.14:8301:8301/udp \
-p 10.0.0.14:8302:8302 \
-p 10.0.0.14:8302:8302/udp \
-p 10.0.0.14:8400:8400 \
-p 10.0.0.14:8500:8500 \
-p 172.17.0.1:53:53/udp \
progrium/consul -server -advertise 10.0.0.14 -bootstrap-expect 2
```

And on the second host we run the output of the second command, replacing the original `-bootstrap-expect` option with `-join` to give the IP address of the other node to Consul.

```
docker run --name node2 -h node2 -p 10.0.0.203:8300:8300 \
-p 10.0.0.203:8301:8301 \
-p 10.0.0.203:8301:8301/udp \
-p 10.0.0.203:8302:8302 \
-p 10.0.0.203:8302:8302/udp \
-p 10.0.0.203:8400:8400 \
-p 10.0.0.203:8500:8500 \
-p 172.17.0.1:53:53/udp \
progrium/consul -server -advertise 10.0.0.203 -join 10.0.0.14
```

Now we have our Consul cluster set up, we can query the API to see what nodes are available.

```
$ curl home.consul:2225/v1/catalog/nodes
[
    {"Node":"node1","Address":"10.0.0.14")},
    {"Node":"node2","Address":"10.0.0.203")}
]
```

Service discovery

So now we've got 2 nodes set up and we can run Docker containers on them but we still have to manage ports and port forwarding to the services etc. That is difficult to maintain at large scale, so how to solve this problem. There is a service called registrator which plugs in to Consul and other similar tools and just runs as another container.

```
docker run -d \
-v /var/run/docker.sock:/tmp/docker.sock \
--name registrator -h registrator \
gilderlabs/registrator:latest consul://10.0.0.14:8500
```

This command uses the gilderlabs/registrator container image, mounts the Docker socket as a volume and gives it the API URL for Consul.

Registrator needs to mount the Docker socket so that it can see when containers appear and disappear and then update Consul. It communicates with Consul via the URI given in the command. Other service discovery tools would be integrated in the same way.

Once we've built the app we'll be able to query the Consul API again to see the services:

```
$ curl home.consul:2225/v1/catalog/services
{
 "consul": [],
 "consul-53": ["udp"],
 "consul-8300": [],
 "consul-8301": ["udp"],
 "consul-8302": ["udp"],
 "consul-8400": [],
 "consul-8500": [],
 "data-django-app": [],
 "nginx": [],
 "nginx-consul-80": [],
 "nginx-consul-8081": [],
 "nginx-consul-8100": [],
}
```

Build the APP

I've already gone through the specifics of each part of the app in a previous blog post so I'll just show how it all plugs together. Simply, I'm running each container on both nodes:

```
$ docker ps
CONTAINER ID        IMAGE                             COMMAND
CREATED             STATUS              PORTS
NAMES
35dbbeca2c58        schizoid90/nginx-consul:0.19.5    "/bin/start.sh"
16 minutes ago      Up 16 minutes       0.0.0.0:8081->8081/tcp,
0.0.0.0:8100->8100/tcp, 0.0.0.0:8080->80/tcp
nginx

2f29c2bfec97        nginx                             "nginx -g
'daemon of…"   4 hours ago         Up 4 hours          0.0.0.0:32782-
>80/tcp
frontend

d21710acccd0        schizoid90/data-django-app
"/usr/local/bin/pyth…"   6 hours ago         Up 6 hours
0.0.0.0:32777->8000/tcp
hungry_golick

866b94365c78        progrium/consul                   "/bin/start -
server …"   7 days ago          Up 7 days          10.0.0.14:8300-
8302->8300-8302/tcp, 10.0.0.14:8400->8400/tcp, 172.17.0.1:53->53/tcp,
172.17.0.1:53->53/udp, 10.0.0.14:8500->8500/tcp, 10.0.0.14:8301-8302-
>8301-8302/udp   consul

3ac04db0abef        gliderlabs/registrator:latest
"/bin/registrator co…"   11 days ago         Up 7 days
registrator
```

You'll notice that the ports for data-django-app and frontend are randomly generated by Docker, Consul can see these allowing us to connect to our containers so there's no need to set specific ports. The ports open for nginx-consul are for an attempted MongoDB cluster (8100 but more on this lower down) and the Consul front end (8081).

Connect to the app

We have our app built and Consul can see everything because Registrator lets it know when a container comes up or down. How can we connect to the app from the outside though? Hashicorp, as usual, have a way to solve this problem. Consul-template is a way to replace values from Consul into the filesystem. You can partner this with a service such as Nginx reverse proxy or HAProxy to direct traffic to your containers. I decided on running Nginx as a container on my nodes. For each of the services I've used a different template file with an upstream and server defined. These are named .ctmpl (Consul template) and should mimic whatever file you want to create with some Consul logic thrown in.

```
upstream data-djang {
  {{range service "data-django-app"}}server {{.Address}}:{{.Port}}
max_fails=3 fail_timeout=60 weight=1;
  {{else}}server 127.0.0.1:65535; # force a 502{{end}}
}

server {
  listen 80;
  server_name datadjango.consul;

charset utf-8;

location / {
    proxy_pass http://data-djang;
  }
}
```

Using this configuration file, consul-template loops over all consul services with the name "data-django-app" and gets the address:port then adds this to the actual file. The upstream ends up looking like

```
upstream data-djang {
  server 10.0.0.14:32777 max_fails=3 fail_timeout=60 weight=1;
  server 10.0.0.203:32775 max_fails=3 fail_timeout=60 weight=1;
}
```

These ctmpl files are put in place on image build by the script start.sh, this starts Nginx then runs the consul-template command to move the .ctmpl files into the correct location.

```
#!/bin/bash
service nginx start
consul-template -consul-addr=$CONSUL_URL \
-
template="/templates/frontend.ctmpl:/etc/nginx/conf.d/frontend.conf:s
ervice nginx reload" \
...
```

$CONSUL_URL is taken as an environment variable when running the container (defaults to consul:8500).

```
docker run -p 8080:80 -p 8081:8081 -p8100:8100 -it --name nginx --
volume /mnt/nginx:/templates/ --link consul:consul schizoid90/nginx-
consul:0.19.5
```

I have the .ctmpl files on the host server under /mnt/nginx so these need mounting under /templates. We also link our consul container to the "consul" keyword so that we can connect to consul:8500 to get the information needed to generate working config from the template files.

If we stop, remove, add or restart any of the containers then the configuration will update automatically as well.

Now we can use a load balancing solution to send traffic on a public port to one of the Consul nodes.

Conclusion

We've seen how we can use Consul, Registrator and Consul-Template to get traffic to our containers without having to know anything about what ports they're listening on or how many there are and what host they're running on.

You may notice that I haven't started a MongoDB container, this is because the work needed to get a distributed cluster on CoreOS was a bit too involved for a quick example and as such I used the instance running in Kubernetes which has already done the heavy lifting required for persistent storage etc.

When I was attempting to configure MongoDB to be part of this cluster I came across the obvious issue of using Nginx to proxy the connections to the MongoDB instances. The issue being that Nginx was expecting HTTP based communication but MongoDB doesn't know how to interpret these. I added a new block to nginx.conf for this container to allow basic TCP proxying.

```
stream {
    include /etc/nginx/conf.d/*.strm; # include stream files
}
```

Then it is a case of telling Consul-Template to modify the relevant .ctmpl file and move it to /etc/nginx/conf.d/mongodb.strm. It is also necessary to use a different port as Nginx can only listen for HTTP connections on the port specified by a http block.

Downloads

Modified Ngix-consul image

Nginx-consul bitbucket

Docker Consul Consul Template Nginx