11 A primer on container orchestration

This chapter covers

- Managing simple Docker services with systemd
- Managing multi-host Docker services with Helios
- Using Hashicorp's Consul for service discovery
- Service registration using Registrator

The technology Docker is built on has existed for a while in different forms, but Docker is the solution that's managed to grab the interest of the technology industry. This puts Docker in an enviable position—Docker's mindshare did the initial job of kickstarting an ecosystem of tools, which became a self-perpetuating cycle of people being drawn into the ecosystem and contributing back to it.

This is particularly evident when it comes to orchestration. After seeing a list of company names with offerings in this space, you'd be forgiven for thinking that everyone has their own opinion on how to do things and has developed their own tool.

Although the ecosystem is a huge strength of Docker (and is why we've been drawing from it so much in this book), the sheer quantity of possible orchestration tools can be overwhelming to novices and veterans alike. This chapter will tour some of the most notable tools and give you a feel for the high-level offerings, so you're better informed when it comes to evaluating what you want a framework to do for you.

There are many ways of arranging family trees of the orchestration tools. Figure 11.1 shows some of the tools we're familiar with. At the root of the tree is `docker run`, the most common way to start a container. Everything inspired by Docker is an offshoot of this. On the left side are tools that treat groups of containers as a single entity. The middle shows the tools focused on managing containers under the umbrella of systemd and service files. Finally, the right side treats individual containers as just that. As you move down the branches, the tools end up doing more for you, be it working across multiple hosts or taking the tedium of manual container deployment away from you.

You'll note two seemingly isolated areas on the diagram—Mesos and the Consul/etcd/Zookeeper group. Mesos is an interesting case—it existed before Docker, and the support it has for Docker is an added feature rather than core functionality. It works very well, though, and should be evaluated carefully, if only to see what features from it you might want in other tools. By contrast, Consul, etcd, and Zookeeper aren't orchestration tools at all. Instead, they provide the important complement to orchestration: service discovery.

This chapter and the next will navigate this orchestration ecosystem. In this chapter we'll introduce tools that give you more fine-grained control and may feel like less of a jump coming from managing containers manually. We'll look at managing Docker containers on a single host and across multiple hosts, and then at saving and

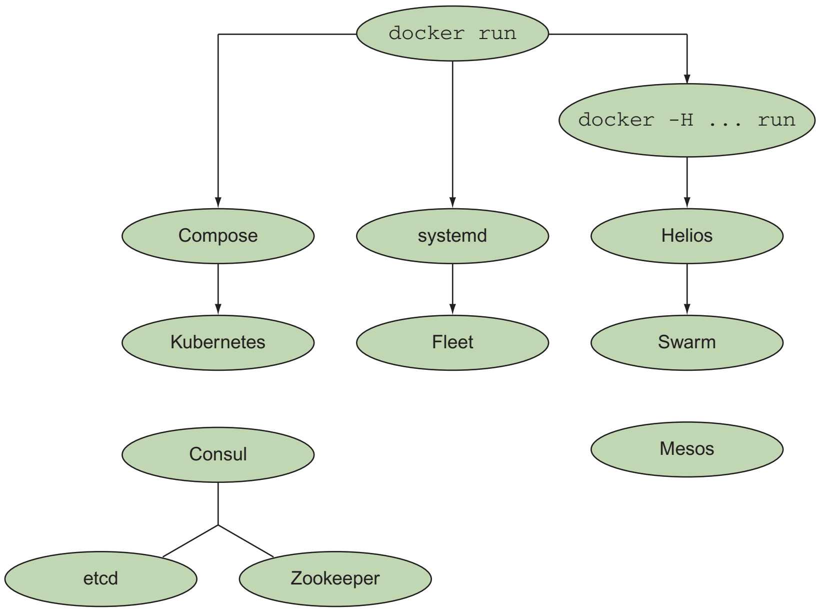

Figure 11.1 Orchestration tools in the Docker ecosystem

retrieving information about where containers have been deployed. Then, in the next chapter, we'll look at more complete solutions that abstract away a lot of the detail.

As you read these two chapters, it might be helpful to take a step back as you come to each orchestration tool and try to come up with a scenario the tool would be useful in. This will help clarify whether a particular tool is relevant for you. We'll offer some examples along the way to get you started.

We'll start slow by turning our gaze inward to a single computer.

11.1 Simple single-host Docker

Managing the containers on your local machine can be a painful experience. The features provided by Docker for managing long-running containers are relatively primitive, and starting up containers with links and shared volumes can be a frustratingly manual process.

In chapter 10 we looked at using Docker Compose to make managing links easier, so we'll deal with the other pain point now and see how the management of long-running containers on a single machine can be made more robust.

TECHNIQUE 82 Managing your host's containers with systemd

In this technique we'll take you through setting up a simple Docker service with systemd. If you're already familiar with systemd, this chapter will be relatively easy to follow, but we assume no prior knowledge of the tool.

Using systemd to control Docker can be useful for a mature company with an operations team that prefers to stick to proven technologies that they already understand and have the tooling for.

PROBLEM

You want to manage the running of Docker container services on your host.

SOLUTION

Use systemd to manage your container services.

systemd is a system-management daemon that replaced SysV init scripts in Fedora some time ago. It manages services on your system—everything from mount points to processes to one-shot scripts—as individual "units." It's growing in popularity as it spreads to other distributions and operating systems, though some systems (Gentoo being an example at the time of writing) may have problems installing and enabling it. It's worth looking around for experiences other people have had with systemd on a setup similar to yours.

In this technique we'll demonstrate how the startup of your containers can be managed by systemd by running the to-do app from chapter 1.

INSTALLING SYSTEMD

If you don't have systemd on your host system (you can check by running `systemctl status` and seeing whether you get a coherent response), you can install it directly on your host OS using your standard package manager.

If you're not comfortable interfering with your host system in this way, the recommended way to play with it is to use Vagrant to provision a systemd-ready VM, as shown in the following listing. We'll cover it briefly here, but see appendix C for more advice on installing Vagrant.

Listing 11.1 A Vagrant setup

```
$ mkdir centos7_docker
 $ cd centos7_docker
 $ vagrant init jdiprizio/centos-docker-io
 $ vagrant up          )
 $ vagrant ssh
```

Creates and enters a new folder

Initializes the folder for use as a Vagrant environment, specifying the Vagrant image

Brings up the VM

SSHes into the VM

NOTE At the time of writing, jdiprizio/centos-docker-io is a suitable and available VM image. If it's no longer available when you're reading this, you can replace that string in the preceding listing with another image name. You can search for one on HashiCorp's "Discover Vagrant Boxes" page: https://app.vagrantup.com/boxes/search ("box" is the terminology Vagrant uses to refer to a VM image). To find this image, we searched for "docker centos". You may need to look up help for the command-line `vagrant box add` command to figure out how to download your new VM before attempting to start it.

SETTING UP A SIMPLE DOCKER APPLICATION UNDER SYSTEMD

Now that you have a machine with systemd and Docker on it, we'll use it to run the to-do application from chapter 1.

Systemd works by reading configuration files in the simple INI file format.

TIP INI files are simple text files with a basic structure composed of sections, properties, and values.

First you need to create a service file as root in /etc/systemd/system/todo.service, as shown in the next listing. In this file you tell systemd to run the Docker container with the name "todo" on port 8000 on this host.

Listing 11.2 /etc/systemd/system/todo.service

```
[Unit]
 Description=Simple ToDo Application
After=docker.service
 Requires=docker.service

[Service]
 Restart=always
 ExecStartPre=/bin/bash \
-c '/usr/bin/docker rm -f todo || /bin/true'
```

The Unit section defines generic information about the systemd object.

Starts this unit after the Docker service is started

The Docker service needs to be running for this unit to successfully run.

The Service section defines the configuration information specific to systemd service unit types.

If the service terminates, always restarts it.

If the service termExecStartPre defines a command that will be run before the unit is started. To ensure the container is removed before you start it, you remove it with prejudice here.inates, always restarts it.

```
ExecStartPre=/usr/bin/docker pull dockerinpractice/todo
ExecStart=/usr/bin/docker run --name todo \
-p 8000:8000 dockerinpractice/todo
ExecStop=/usr/bin/docker rm -f todo

[Install]
WantedBy=multi-user.target
```

ExecStart defines the command to be run when the service is started.

Makes sure the image is downloaded before you run the container

ExecStop defines the command to be run when the service is stopped.

The Install section contains information for systemd when enabling the unit.

Informs systemd that you want this unit to be started when it enters the multi-user target stage

This configuration file should make it clear that systemd offers a simple declarative schema for managing processes, leaving the details of dependency management up to the systemd service. This doesn't mean that you can ignore the details, but it does put a lot of tools at your disposal for managing Docker (and other) processes.

> **NOTE** Docker doesn't set any container restart policies by default, but be aware that any you set will conflict with most process managers. Don't set restart policies if you're using a process manager.

Enabling a new unit is just a matter of invoking the `systemctl enable` command. If you want this unit to start automatically when the system boots, you can also create a symlink in the multi-user.target.wants systemd directory. Once done, you can start the unit with `systemctl start`.

```
$ systemctl enable /etc/systemd/system/todo.service
$ ln -s '/etc/systemd/system/todo.service' \
'/etc/systemd/system/multi-user.target.wants/todo.service'
$ systemctl start todo.service
```

Then just wait for it to start. If there's a problem, you'll be informed.

To check that all is OK, use the `systemctl status` command. It will print out some general information about the unit, such as how long it's been running and the process ID, followed by a number of log lines from the process. In this case, seeing `Swarm server started port 8000` is a good sign:

```
[root@centos system]# systemctl status todo.service
todo.service - Simple ToDo Application
   Loaded: loaded (/etc/systemd/system/todo.service; enabled)
   Active: active (running) since Wed 2015-03-04 19:57:19 UTC; 2min 13s ago
  Process: 21266 ExecStartPre=/usr/bin/docker pull dockerinpractice/todo \
(code=exited, status=0/SUCCESS)
  Process: 21255 ExecStartPre=/bin/bash -c /usr/bin/docker rm -f todo || \
/bin/true (code=exited, status=0/SUCCESS)
  Process: 21246 ExecStartPre=/bin/bash -c /usr/bin/docker kill todo || \
/bin/true (code=exited, status=0/SUCCESS)
 Main PID: 21275 (docker)
   CGroup: /system.slice/todo.service
           ??21275 /usr/bin/docker run --name todo
➥ -p 8000:8000 dockerinpractice/todo
```

```
Mar 04 19:57:24 centos docker[21275]: TodoApp.js:117:        \
// TODO scroll into view
Mar 04 19:57:24 centos docker[21275]: TodoApp.js:176:        \
if (i>=list.length()) { i=list.length()-1; } // TODO .length
Mar 04 19:57:24 centos docker[21275]: local.html:30:    \
<!-- TODO 2-split, 3-split -->
Mar 04 19:57:24 centos docker[21275]: model/TodoList.js:29:        \
// TODO one op - repeated spec? long spec?
Mar 04 19:57:24 centos docker[21275]: view/Footer.jsx:61:        \
// TODO: show the entry's metadata
Mar 04 19:57:24 centos docker[21275]: view/Footer.jsx:80:        \
todoList.addObject(new TodoItem()); // TODO create default
Mar 04 19:57:24 centos docker[21275]: view/Header.jsx:25:        \
// TODO list some meaningful header (apart from the id)
Mar 04 19:57:24 centos docker[21275]: > todomvc-swarm@0.0.1 start /todo
Mar 04 19:57:24 centos docker[21275]: > node TodoAppServer.js
Mar 04 19:57:25 centos docker[21275]: Swarm server started port 8000
```

You can now visit the server on port 8000.

DISCUSSION

The principles in this technique can be applied to more than just systemd—most process managers, including other init systems, can be configured in a similar way. If you're interested, you could leverage this to replace existing services running on your system (perhaps a PostgreSQL database) with dockerized ones.

In the next technique, we'll take this further by implementing in systemd the SQLite server we created in technique 77.

TECHNIQUE 83 Orchestrating the startup of your host's containers

Unlike docker-compose (at the time of writing), systemd is a mature technology ready for production. In this technique we'll show you how to achieve local orchestration functionality that's similar to docker-compose using systemd.

NOTE If you run into trouble with this technique, you may need to upgrade your version of Docker. Version 1.7.0 or greater should work fine.

PROBLEM

You want to manage more complex container orchestration on one host in production.

SOLUTION

Use systemd with dependent services to manage your containers.

To demonstrate the use of systemd for a more complex scenario, we're going to re-implement the SQLite TCP server example from technique 77 in systemd. Figure 11.2 illustrates the dependencies for our planned systemd service unit configuration.

This is a similar schema to what you saw with the Docker Compose example in technique 77. A key difference here is that rather than the SQLite service being treated as a single monolithic entity, each container is a discrete entity. In this scenario, the SQLite proxy can be stopped independently of the SQLite server.

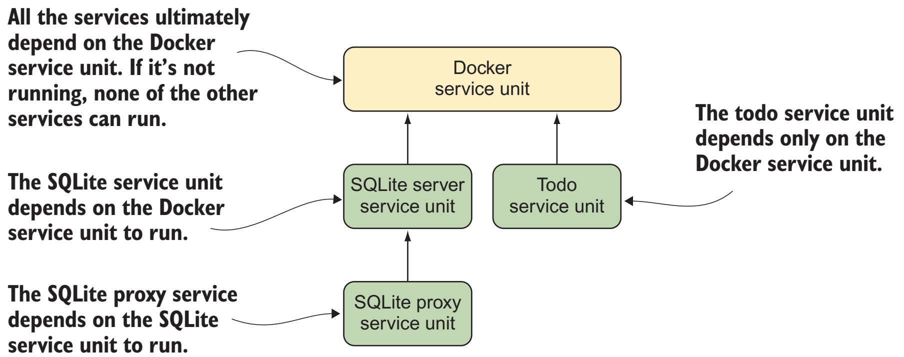
All the services ultimately depend on the Docker service unit. If it's not running, none of the other services can run.
Docker service unit
The todo service unit depends only on the Docker service unit.
The SQLite service unit depends on the Docker service unit to run.
SQLite server service unit
Todo service unit
The SQLite proxy service depends on the SQLite service unit to run.
SQLite proxy service unit

Figure 11.2 systemd unit dependency graph

Here's the listing for the SQLite server service. As before, it depends on the Docker service, but it has a couple of differences from the to-do example in the previous technique.

Listing 11.3 /etc/systemd/system/sqliteserver.service

```
[Unit]
 Description=SQLite Docker Server
After=docker.service
 Requires=docker.service

[Service]
Restart=always
ExecStartPre=-/bin/touch /tmp/sqlitedbs/test
 ExecStartPre=-/bin/touch /tmp/sqlitedbs/live
 ExecStartPre=/bin/bash \
-c '/usr/bin/docker kill sqliteserver || /bin/true'
 ExecStartPre=/bin/bash \
-c '/usr/bin/docker rm -f sqliteserver || /bin/true'
 ExecStartPre=/usr/bin/docker \
pull dockerinpractice/docker-compose-sqlite
 ExecStart=/usr/bin/docker run --name sqliteserver \
 -v /tmp/sqlitedbs/test:/opt/sqlite/db \
dockerinpractice/docker-compose-sqlite /bin/bash -c \
'socat TCP-L:12345,fork,reuseaddr \
EXEC:"sqlite3 /opt/sqlite/db",pty'
ExecStop=/usr/bin/docker rm -f sqliteserver

[Install]
WantedBy=multi-user.target
```

The Unit section defines generic information about the systemd object.

The Docker service needs to be running for this unit to successfully run.

Starts this unit after the Docker service is started

These lines ensure that the SQLite database files exist before the service starts up. The dash before the touch command indicates to systemd that startup should fail if the command returns an error code.

Makes sure the image is downloaded before you run the container

ExecStartPre defines a command that will be run before the unit is started. To ensure the container is removed before you start it, you remove it with prejudice here.

ExecStop defines the command to be run when the service is stopped.

ExecStart defines the command to be run when the service is started. Note that we've wrapped the socat command in a "/bin/bash -c" call to avoid confusion, as the ExecStart line is run by systemd.

TIP Paths must be absolute in systemd.

Now comes the listing for the SQLite proxy service. The key difference here is that the proxy service depends on the server process you just defined, which in turn depends on the Docker service.

Listing 11.4 /etc/systemd/system/sqliteproxy.service

```
[Unit]
Description=SQLite Docker Proxy
After=sqliteserver.service
 Requires=sqliteserver.service

[Service]
Restart=always
ExecStartPre=/bin/bash -c '/usr/bin/docker kill sqliteproxy || /bin/true'
ExecStartPre=/bin/bash -c '/usr/bin/docker rm -f sqliteproxy || /bin/true'
ExecStartPre=/usr/bin/docker pull dockerinpractice/docker-compose-sqlite
ExecStart=/usr/bin/docker run --name sqliteproxy \
-p 12346:12346 --link sqliteserver:sqliteserver \
dockerinpractice/docker-compose-sqlite /bin/bash \
-c 'socat TCP-L:12346,fork,reuseaddr TCP:sqliteserver:12345'
 ExecStop=/usr/bin/docker rm -f sqliteproxy

[Install]
WantedBy=multi-user.target
```

The proxy unit must run after the sqliteserver service defined previously.

The proxy requires that the server instance be running before you start it up.

Runs the container

With these two configuration files, we've laid the groundwork for installing and running the SQLite service under systemd's control. Now we can enable these services:

```
$ sudo systemctl enable /etc/systemd/system/sqliteserver.service
ln -s '/etc/systemd/system/sqliteserver.service' \
'/etc/systemd/system/multi-user.target.wants/sqliteserver.service'
$ sudo systemctl enable /etc/systemd/system/sqliteproxy.service
ln -s '/etc/systemd/system/sqliteproxy.service' \
'/etc/systemd/system/multi-user.target.wants/sqliteproxy.service'
```

And start them up:

```
$ sudo systemctl start sqliteproxy
$ telnet localhost 12346
[vagrant@centos ~]$ telnet localhost 12346
Trying ::1...
Connected to localhost.
Escape character is '^]'.
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from t1;
select * from t1;
test
```

Note that because the SQLite proxy service depends on the SQLite server service to run, you only need to start the proxy—the dependencies get started automatically.

DISCUSSION

One of the challenges when administering a long-running application on a local machine is the management of dependency services. For example, a web application might expect to be running in the background as a service but might also depend on a database and a web server. This may sound familiar—you covered a web-app-db structure in technique 13.

Technique 76 showed how to set up this kind of structure with dependencies and so on, but tools like systemd have been working on this problem for a while and may offer flexibility that Docker Compose doesn't. For example, once you've written your service files, you can start any of them you want, and systemd will handle starting up any dependent services, even starting the Docker daemon itself if necessary.

11.2 Manual multi-host Docker

Now that you're comfortable with some fairly complicated arrangements of Docker containers on a machine, it's time to think bigger—let's move on to the world of multiple hosts to enable us to use Docker on a larger scale.

In the rest of this chapter, you're going to manually run a multi-host environment with Helios to introduce you to multi-host Docker concepts. In the next chapter, you'll see more automated and sophisticated ways to achieve the same result and more.

TECHNIQUE 84 Manual multi-host Docker with Helios

It can be intimidating to hand over all control of provisioning a group of machines to an application, so it doesn't hurt to ease yourself in with a more manual approach.

Helios is ideal for companies that have mostly static infrastructures and are interested in using Docker for their critical services but (understandably) want human oversight in the process.

PROBLEM

You want to be able to provision multiple Docker hosts with containers but retain manual control over what runs where.

SOLUTION

Use the Helios tool from Spotify to precisely manage containers on other hosts.

Helios is the tool Spotify currently uses to manage their servers in production, and it has the pleasing property of being both easy to get started with and stable (as you'd hope). Helios allows you to manage the deployment of Docker containers across multiple hosts. It gives you a single command-line interface that you can use to specify what you want to run and where to run it, as well as the ability to take a look at the current state of play.

Because we're just introducing Helios, we're going to run everything on a single node inside Docker for simplicity—don't worry, anything relevant to running on multiple hosts will be clearly highlighted. The high-level architecture of Helios is outlined in figure 11.3.

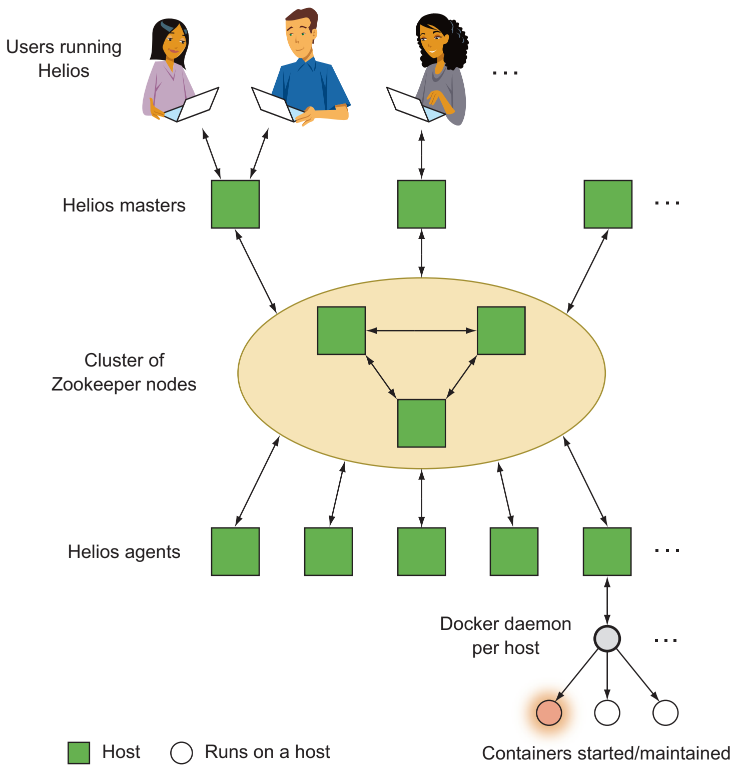

Figure 11.3 A birds-eye view of a Helios installation

As you can see, there's only one additional service required when running Helios: Zookeeper. Helios uses Zookeeper to track the state of all of your hosts and as a communication channel between the masters and agents.

> **TIP** Zookeeper is a lightweight distributed database written in Java that's optimized for storing configuration information. It's part of the Apache suite of open source software products. It's similar in functionality to etcd (which you learned about in chapter 9, and which you'll see again in this chapter).

All you need to know for this technique is that Zookeeper stores data such that it can be distributed across multiple nodes (for both scalability and reliability) by running multiple Zookeeper instances. This may sound familiar to our description of etcd in chapter 9—these two tools have significant overlap.

To start the single Zookeeper instance we'll use in this technique, run the following command:

```
$ docker run --name zookeeper -d jplock/zookeeper:3.4.6
cd0964d2ba18baac58b29081b227f15e05f11644adfa785c6e9fc5dd15b85910
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' zookeeper
172.17.0.9
```

NOTE When starting a Zookeeper instance on its own node, you'll want to expose ports to make it accessible to other hosts and use volumes to persist data. Take a look at the Dockerfile on the Docker Hub for details about which ports and folders you should use (https://hub.docker.com/r/jplock/zookeeper/~/dockerfile/). It's also likely you'll want to run Zookeeper on multiple nodes, but configuring a Zookeeper cluster is beyond the scope of this technique.

You can inspect the data Zookeeper has stored by using the zkCli.sh tool, either interactively or by piping input to it. The initial startup is quite chatty, but it'll drop you into an interactive prompt where you can run commands against the file-tree-like structure Zookeeper stores data in.

```
$ docker exec -it zookeeper bin/zkCli.sh
Connecting to localhost:2181
2015-03-07 02:56:05,076 [myid:] - INFO  [main:Environment@100] - Client >
environment:zookeeper.version=3.4.6-1569965, built on 02/20/2014 09:09 GMT
2015-03-07 02:56:05,079 [myid:] - INFO  [main:Environment@100] - Client >
environment:host.name=917d0f8ac077
2015-03-07 02:56:05,079 [myid:] - INFO  [main:Environment@100] - Client >
environment:java.version=1.7.0_65
2015-03-07 02:56:05,081 [myid:] - INFO  [main:Environment@100] - Client >
environment:java.vendor=Oracle Corporation
[...]
2015-03-07 03:00:59,043 [myid:] - INFO
➥ [main-SendThread(localhost:2181):ClientCnxn$SendThread@1235] -
➥ Session establishment complete on server localhost/0:0:0:0:0:0:0:1:2181,
➥ sessionid = 0x14bf223e159000d, negotiated timeout = 30000

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0] ls /
[zookeeper]
```

Nothing's running against Zookeeper yet, so the only thing currently being stored is some internal Zookeeper information. Leave this prompt open, and we'll revisit it as we progress.

Helios itself is split into three parts:

- *The master*—This is used as an interface for making changes in Zookeeper.
- *The agent*—This runs on every Docker host, starts and stops containers based on Zookeeper, and reports state back.
- *The command-line tools*—These are used to make requests to the master.

Figure 11.4 shows how the final system is strung together when we perform an operation against it (the arrows indicate data flow).

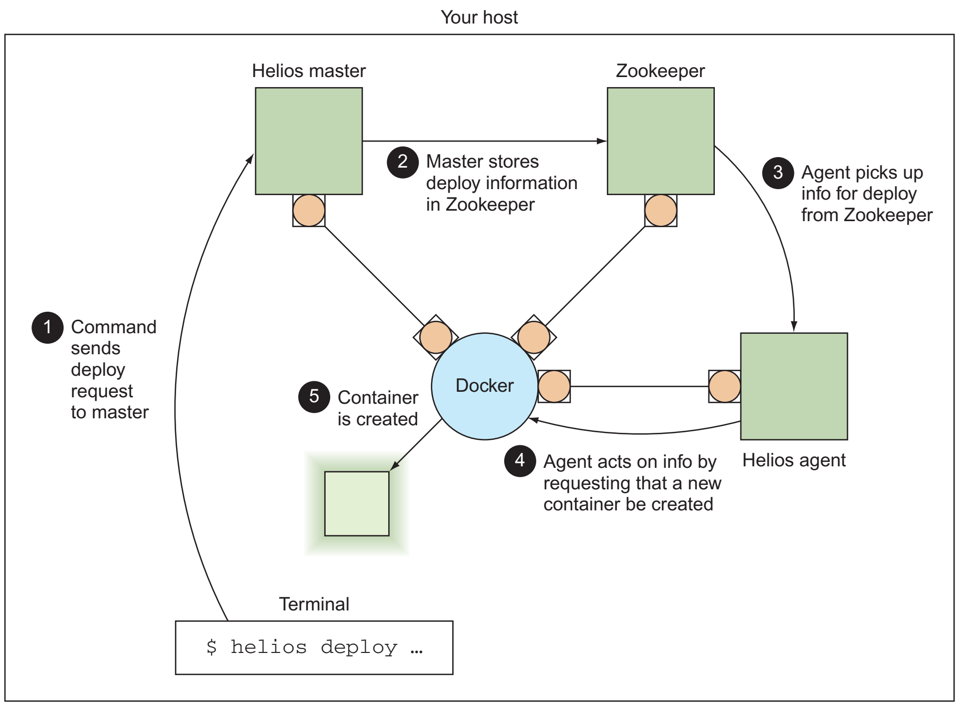
Your host
Helios master
Zookeeper
2 Master stores deploy information in Zookeeper
3 Agent picks up info for deploy from Zookeeper
1 Command sends deploy request to master
Docker
5 Container is created
4 Agent acts on info by requesting that a new container be created
Helios agent
Terminal
$ helios deploy …

Figure 11.4 Starting a container on a single-host Helios installation

Now that Zookeeper is running, it's time to start Helios. We need to run the master while specifying the IP address of the Zookeeper node we started earlier:

```
$ IMG=dockerinpractice/docker-helios
$ docker run -d --name hmaster $IMG helios-master --zk 172.17.0.9
896bc963d899154436938e260b1d4e6fdb0a81e4a082df50043290569e5921ff
$ docker logs --tail=3 hmaster
03:20:14.460 helios[1]: INFO  [MasterService STARTING] ContextHandler: >
Started i.d.j.MutableServletContextHandler@7b48d370{/,null,AVAILABLE}
03:20:14.465 helios[1]: INFO  [MasterService STARTING] ServerConnector: >
Started application@2192bcac{HTTP/1.1}{0.0.0.0:5801}
03:20:14.466 helios[1]: INFO  [MasterService STARTING] ServerConnector: >
Started admin@28a0d16c{HTTP/1.1}{0.0.0.0:5802}
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' hmaster
172.17.0.11
```

Now let's see what's new in Zookeeper:

```
[zk: localhost:2181(CONNECTED) 1] ls /
[history, config, status, zookeeper]
[zk: localhost:2181(CONNECTED) 2] ls /status/masters
[896bc963d899]
[zk: localhost:2181(CONNECTED) 3] ls /status/hosts
[]
```

It looks like the Helios master has created a bunch of new pieces of configuration, including registering itself as a master. Unfortunately we don't have any hosts yet.

Let's solve this by starting up an agent that will use the current host's Docker socket to start containers on:

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock -d --name hagent \
dockerinpractice/docker-helios helios-agent --zk 172.17.0.9
5a4abcb271070d0171ca809ff2beafac5798e86131b72aeb201fe27df64b2698
$ docker logs --tail=3 hagent
03:30:53.344 helios[1]: INFO  [AgentService STARTING] ContextHandler: >
Started i.d.j.MutableServletContextHandler@774c71b1{/,null,AVAILABLE}
03:30:53.375 helios[1]: INFO  [AgentService STARTING] ServerConnector: >
Started application@7d9e6c27{HTTP/1.1}{0.0.0.0:5803}
03:30:53.376 helios[1]: INFO  [AgentService STARTING] ServerConnector: >
Started admin@2bceb4df{HTTP/1.1}{0.0.0.0:5804}
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' hagent
172.17.0.12
```

Again, let's check back in Zookeeper:

```
[zk: localhost:2181(CONNECTED) 4] ls /status/hosts
[5a4abcb27107]
[zk: localhost:2181(CONNECTED) 5] ls /status/hosts/5a4abcb27107
[agentinfo, jobs, environment, hostinfo, up]
[zk: localhost:2181(CONNECTED) 6] get /status/hosts/5a4abcb27107/agentinfo
{"inputArguments":["-Dcom.sun.management.jmxremote.port=9203", [...]
[...]
```

You can see here that `/status/hosts` now contains one item. Descending into the Zookeeper directory for the host reveals the internal information Helios stores about the host.

> **NOTE** When running on multiple hosts, you'll want to pass `--name $(hostname -f)` as an argument to both the Helios master and agent. You'll also need to expose ports 5801 and 5802 for the master and 5803 and 5804 for the agent.

Let's make it a bit easier to interact with Helios:

```
$ alias helios="docker run -i --rm dockerinpractice/docker-helios \
helios -z http://172.17.0.11:5801"
```

The preceding alias means that invoking `helios` will start a throwaway container to perform the action you want, pointing at the correct Helios cluster to begin with. Note that the command-line interface needs to be pointed at the Helios master rather than Zookeeper.

Everything is now set up. We're able to easily interact with our Helios cluster, so it's time to try an example.

```
$ helios create -p nc=8080:8080 netcat:v1 ubuntu:14.04.2 -- \
sh -c 'echo hello | nc -l 8080'
Creating job: {"command":["sh","-c","echo hello | nc -l 8080"], >
"creatingUser":null,"env":{},"expires":null,"gracePeriod":null, >
"healthCheck":null,"id": >
"netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac", >
"image":"ubuntu:14.04.2","ports":{"nc":{"externalPort":8080, >
"internalPort":8080,"protocol":"tcp"}},"registration":{}, >
"registrationDomain":"","resources":null,"token":"","volumes":{}}
Done.
netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac
$ helios jobs
JOB ID            NAME   VERSION HOSTS COMMAND                    ENVIRONMENT
netcat:v1:2067d43 netcat v1      0     sh -c "echo hello | nc -l 8080"
```

Helios is built around the concept of *jobs*—everything to be executed must be expressed as a job before it can be sent to a host to be executed. At a minimum, you need an image with the basics Helios needs to know to start the container: a command to execute and any port, volume, or environment options. You may also want a number of other advanced options, including health checks, expiry dates, and service registration.

The previous command creates a job that will listen on port 8080, print "hello" to the first thing that connects to the port, and then terminate.

You can use `helios hosts` to list hosts available for job deployment, and then actually perform the deployment with `helios deploy`. The `helios status` command then shows us that the job has successfully started:

```
$ helios hosts
HOST         STATUS         DEPLOYED RUNNING CPUS MEM  LOAD AVG MEM USAGE >
OS                        HELIOS  DOCKER
5a4abcb27107.Up 19 minutes 0        0       4    7 gb 0.61     0.84      >
Linux 3.13.0-46-generic 0.8.213 1.3.1 (1.15)
$ helios deploy netcat:v1 5a4abcb27107
Deploying Deployment{jobId=netcat:v1: >
2067d43fc2c6f004ea27d7bb7412aff502e3cdac, goal=START, deployerUser=null} >
on [5a4abcb27107]
5a4abcb27107: done
$ helios status
JOB ID            HOST         GOAL  STATE   CONTAINER ID PORTS
netcat:v1:2067d43 5a4abcb27107.START RUNNING b1225bc      nc=8080:8080
```

Of course, we now want to verify that the service works:

```
$ curl localhost:8080
hello
$ helios status
JOB ID            HOST         GOAL  STATE         CONTAINER ID PORTS
netcat:v1:2067d43 5a4abcb27107.START PULLING_IMAGE b1225bc      nc=8080:8080
```

The result of `curl` clearly tells us that the service is working, but `helios status` is now showing something interesting. When defining the job, we noted that after serving "hello", the job would terminate, but the preceding output shows a PULLING_IMAGE status. This is down to how Helios manages jobs—once you've deployed to a host, Helios will do its best to keep the job running. The status you can see here is Helios going through the complete job startup process, which happens to involve ensuring the image is pulled.

Finally, we need to clear up after ourselves.

```
$ helios undeploy -a --yes netcat:v1
Undeploying netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac from >
[5a4abcb27107]
5a4abcb27107: done
$ helios remove --yes netcat:v1
Removing job netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac
netcat:v1:2067d43fc2c6f004ea27d7bb7412aff502e3cdac: done
```

We asked for the job to be removed from all nodes (terminating it if necessary, and stopping any more automatic restarts), and then we deleted the job itself, meaning it can't be deployed to any more nodes.

DISCUSSION

Helios is a simple and reliable way of deploying your containers to multiple hosts. Unlike a number of techniques we'll come to later on, there's no magic going on behind the scenes to determine appropriate locations—Helios starts containers exactly where you want them with minimal fuss.

But this simplicity comes at a cost once you move to more advanced deployment scenarios—features like resource limits, dynamic scaling, and so on are currently missing, so you may find yourself reinventing parts of tools like Kubernetes (technique 88) to achieve the behavior you want in your deployment.

11.3 Service discovery: What have we here?

This chapter's introduction referred to service discovery as the flip side of orchestration—being able to deploy your applications to hundreds of different machines is fine, but if you can't then find out which applications are located where, you won't be able to actually *use* them.

Although it's not nearly as saturated an area as orchestration, the service-discovery field still has a number of competitors. It doesn't help that they all offer slightly different feature sets.

There are two pieces of functionality that are typically desirable when it comes to service discovery: a generic key/value store and a way of retrieving service endpoints via some convenient interface (likely DNS). etcd and Zookeeper are examples of the former, whereas SkyDNS (a tool we won't go into) is an example of the latter. In fact, SkyDNS uses etcd to store the information it needs.

TECHNIQUE 85 Using Consul to discover services

etcd is a highly popular tool, but it does have one particular competitor that gets mentioned alongside it a lot: Consul. This is a little strange, because there are other tools more similar to etcd (Zookeeper has a similar feature set to etcd but is implemented in a different language), whereas Consul differentiates itself with some interesting additional features, like service discovery and health checks. In fact, if you squint, Consul might look a bit like etcd, SkyDNS, and Nagios all in one.

PROBLEM

You need to be able to distribute information to, discover services within, and monitor a collection of containers.

SOLUTION

Start a container with Consul on each Docker host to provide a service directory and configuration communication system.

Consul tries to be a generic tool for doing some important tasks required when you need to coordinate a number of independent services. These tasks can be performed by other tools, but configuring them in one place can be useful. From a high level, Consul provides the following:

- *Service configuration*—A key/value store for storing and sharing small values, like etcd and Zookeeper
- *Service discovery*—An API for registering services and a DNS endpoint for discovering them, like SkyDNS
- *Service monitoring*—An API for registering health checks, like Nagios

You can use all, some, or one of these features, as there's no tie-in. If you have existing monitoring infrastructure, there's no need to replace that with Consul.

This technique will cover the service-discovery and service-monitoring aspects of Consul, but not key/value storage. The strong similarities between etcd and Consul in this aspect make the two final techniques in chapter 9 (techniques 74 and 75) transferrable with some perusal of the Consul documentation.

Figure 11.5 shows a typical Consul setup.

The data stored in Consul is the responsibility of *server* agents. These are responsible for forming a *consensus* on the information stored—this concept is present in most distributed data-storage systems. In short, if you lose under half of your server agents, you're guaranteed to be able to recover your data (see an example of this with etcd in technique 74). Because these servers are so important and have greater resource requirements, keeping them on dedicated machines is a typical choice.

NOTE Although the commands in this technique will leave the Consul data directory (/data) inside the container, it's generally a good idea to specify this directory as a volume for at least the servers, so you can keep backups.

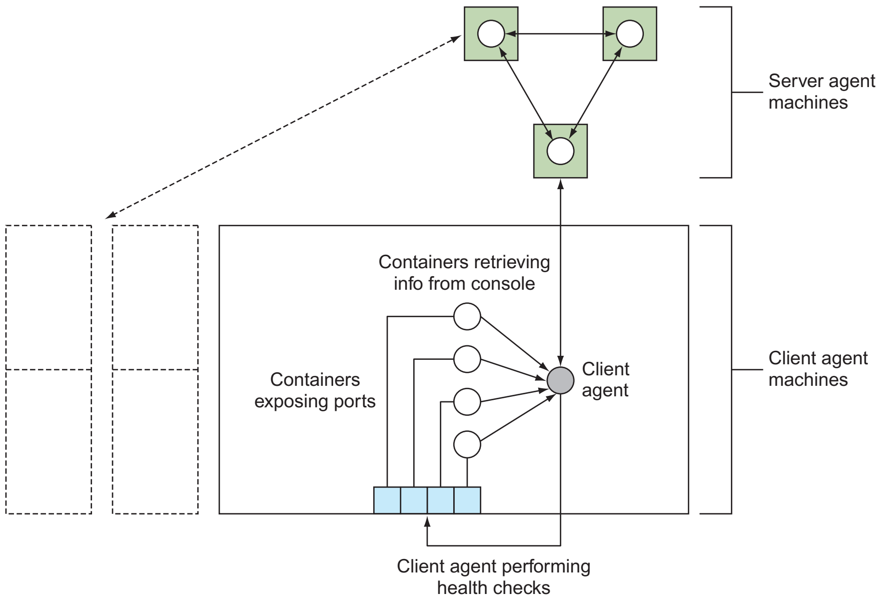
Server agent machines
Containers retrieving info from console
Containers exposing ports
Client agent
Client agent machines
Client agent performing health checks

Figure 11.5 A typical Consul setup

It's recommended that all machines under your control that may want to interact with Consul should run a client agent. These agents forward requests on to the servers and run health checks.

The first step in getting Consul running is to start a server agent:

```
c1 $ IMG=dockerinpractice/consul-server
c1 $ docker pull $IMG
[...]
c1 $ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$\|vbox.*$'
    inet 192.168.1.87/24 brd 192.168.1.255 scope global wlan0
c1 $ EXTIP1=192.168.1.87
c1 $ echo '{"ports": {"dns": 53}}' > dns.json
c1 $ docker run -d --name consul --net host \
-v $(pwd)/dns.json:/config/dns.json $IMG -bind $EXTIP1 -client $EXTIP1 \
-recursor 8.8.8.8 -recursor 8.8.4.4 -bootstrap-expect 1
88d5cb48b8b1ef9ada754f97f024a9ba691279e1a863fa95fa196539555310c1
c1 $ docker logs consul
[...]
       Client Addr: 192.168.1.87 (HTTP: 8500, HTTPS: -1, DNS: 53, RPC: 8400)
      Cluster Addr: 192.168.1.87 (LAN: 8301, WAN: 8302)
[...]
==> Log data will now stream in as it occurs:

    2015/08/14 12:35:41 [INFO] serf: EventMemberJoin: mylaptop 192.168.1.87
[...]
    2015/08/14 12:35:43 [INFO] consul: member 'mylaptop' joined, marking >
health alive
    2015/08/14 12:35:43 [INFO] agent: Synced service 'consul'
```

Because we want to use Consul as a DNS server, we've inserted a file into the folder Consul reads the configuration from to request it listen on port 53 (the registered port for the DNS protocol). We've then used a command sequence you may recognize from earlier techniques to try to find the external-facing IP address of the machine for both communicating with other agents and listening for client requests.

NOTE The IP address `0.0.0.0` is typically used to indicate that an application should listen on all available interfaces on the machine. We've deliberately not done this, because some Linux distributions have a DNS-caching daemon listening on `127.0.0.1`, which disallows listening on `0.0.0.0:53`.

There are three items of note in the preceding `docker run` command:

- We've used `--net host`. Although this can be seen as a faux pas in the Docker world, the alternative is to expose up to eight ports on the command line—it's a matter of personal preference, but we feel it's justified here. It also helps bypass a potential issue with UDP communication. If you were to go the manual route, there'd be no need to set the DNS port—you could expose the default Consul DNS port (8600) as port 53 on the host.
- The two `recursor` arguments tell Consul what DNS servers to look at if a requested address is unknown by Consul itself.
- The `-bootstrap-expect 1` argument means the Consul cluster will start operating with only one agent, which is not robust. A typical setup would set this to 3 (or more) to make sure the cluster doesn't start until the required number of servers has joined. To start the additional server agents, add a `-join` argument, as we'll discuss when we start a client.

Now let's go to a second machine, start a client agent, and add it to our cluster.

WARNING Because Consul expects to be able to listen on a particular set of ports when communicating with other agents, it's tricky to set up multiple agents on a single machine while still demonstrating how it would work in the real world. We'll use a different host now—if you decide to use an IP alias, ensure you pass a `-node newAgent`, because by default the hostname will be used, which will conflict.

```
c2 $ IMG=dockerinpractice/consul-agent
c2 $ docker pull $IMG
[...]
c2 $ EXTIP1=192.168.1.87
c2 $ ip addr | grep docker0 | grep inet
    inet 172.17.42.1/16 scope global docker0
c2 $ BRIDGEIP=172.17.42.1
c2 $ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$'
    inet 192.168.1.80/24 brd 192.168.1.255 scope global wlan0
c2 $ EXTIP2=192.168.1.80
c2 $ echo '{"ports": {"dns": 53}}' > dns.json
c2 $ docker run -d --name consul-client --net host \
-v $(pwd)/dns.json:/config/dns.json $IMG -client $BRIDGEIP -bind $EXTIP2 \
-join $EXTIP1 -recursor 8.8.8.8 -recursor 8.8.4.4
5454029b139cd28e8500922d1167286f7e4fb4b7220985ac932f8fd5b1cdef25
c2 $ docker logs consul-client
[...]
    2015/08/14 19:40:20 [INFO] serf: EventMemberJoin: mylaptop2 192.168.1.80
[...]
    2015/08/14 13:24:37 [INFO] consul: adding server mylaptop >
(Addr: 192.168.1.87:8300) (DC: dc1)
```

NOTE The images we've used are based on gliderlabs/consul-server:0.5 and gliderlabs/consul-agent:0.5, and they come with a newer version of Consul to avoid possible problems with UDP communication, indicated by the constant logging of lines like "Refuting a suspect message." When version 0.6 of the images is released, you can switch back to the images from gliderlabs.

All client services (HTTP, DNS, and so on) have been configured to listen on the Docker bridge IP address. This gives containers a known location from which they can retrieve information from Consul, and it only exposes Consul internally on the machine, forcing other machines to directly access the server agents rather than taking a slower route via a client agent to a server agent. To ensure the bridge IP address is consistent across all your hosts, you can look at the `--bip` argument to the Docker daemon.

As before, we've found the external IP address and bound cluster communication to it. The `-join` argument tells Consul where to initially look to find the cluster. Don't worry about micromanaging the cluster formation—when two agents initially meet each other, they'll *gossip*, transferring information about finding the other agents in the cluster. The final `-recursor` arguments tell Consul what upstream DNS servers to use for DNS requests that aren't trying to look up registered services.

Let's verify that the agent has connected to the server with the HTTP API on the client machine. The API call we'll use will return a list of members the client agent currently thinks are in the cluster. In large, quickly changing clusters, this may not always match the members of the cluster—there's another (slower) API call for that.

```
c2 $ curl -sSL $BRIDGEIP:8500/v1/agent/members | tr ',' '\n' | grep Name
[{"Name":"mylaptop2"
{"Name":"mylaptop"
```

Now that the Consul infrastructure is set up, it's time to see how you can register and discover services. The typical process for registration is to get your app to make an API call against the local client agent after initializing, which prompts the client agent to distribute the information to the server agents. For demonstration purposes, we'll perform the registration step manually.

```
c2 $ docker run -d --name files -p 8000:80 ubuntu:14.04.2 \
python3 -m http.server 80
96ee81148154a75bc5c8a83e3b3d11b73d738417974eed4e019b26027787e9d1
c2 $ docker inspect -f '{{.NetworkSettings.IPAddress}}' files
```

```
172.17.0.16
c2 $ /bin/echo -e 'GET / HTTP/1.0\r\n\r\n' | nc -i1 172.17.0.16 80 \
| head -n 1
HTTP/1.0 200 OK
c2 $ curl -X PUT --data-binary '{"Name": "files", "Port": 8000}' \
$BRIDGEIP:8500/v1/agent/service/register
c2 $ docker logs consul-client | tail -n 1
    2015/08/15 03:44:30 [INFO] agent: Synced service 'files'
```

Here we've set up a simple HTTP server in a container, exposing it on port 8000 on the host, and checked that it works. Then we used curl and the Consul HTTP API to register a service definition. The only thing absolutely necessary here is the name of the service—the port, along with the other fields listed in the Consul documentation, are all optional. The ID field is worth a mention—it defaults to the name of the service but must be unique across all services. If you want multiple instances of a service, you'll need to specify it.

The log line from Consul has told us that the service is synced, so we should be able to retrieve the information about it from the service DNS interface. This information comes from the server agents, so it acts as validation that the service has been accepted into the Consul catalog. You can use the `dig` command to query service DNS information and check that it's present:

Looks up the IP address of the files service from the server agent DNS. This DNS service is available to arbitrary machines not in your Consul cluster, allowing them to benefit from service discovery as well.

Looks up the IP address of the files service from the client agent DNS. If using $BRIDGEIP fails, you may wish to try with $EXTIPI.

Requests the SRV record of the files service from the client agent DNS

Starts a container configured to use the local client agent as the only DNS server

Verifies that service lookup works automatically inside the container

```
c2 $ EXTIP1=192.168.1.87
c2 $ dig @$EXTIP1 files.service.consul +short
 192.168.1.80
c2 $ BRIDGEIP=172.17.42.1
c2 $ dig @$BRIDGEIP files.service.consul +short
 192.168.1.80
c2 $ dig @$BRIDGEIP files.service.consul srv +short
 1 1 8000 mylaptop2.node.dc1.consul.
c2 $ docker run -it --dns $BRIDGEIP ubuntu:14.04.2 bash
 root@934e9c26bc7e:/# ping -c1 -q www.google.com
 PING www.google.com (216.58.210.4) 56(84) bytes of data.

--- www.google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 25.358/25.358/25.358/0.000 ms
root@934e9c26bc7e:/# ping -c1 -q files.service.consul
 PING files.service.consul (192.168.1.80) 56(84) bytes of data.

--- files.service.consul ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.062/0.062/0.062/0.000 ms
```

Verifies that lookup of external addresses still works

NOTE SRV records are a way of communicating service information by DNS, including protocol, port, and other entries. In the preceding case, you can see the port number in the response, and you've been given the canonical hostname of the machine providing the service rather than the IP address.

Advanced users may want to avoid manually setting the `--dns` argument by configuring the –dns and –bip arguments for the Docker daemon itself, but remember to override the defaults for the Consul agent, or you may end up with unexpected behavior.

The similarities between the Consul DNS service and the Docker virtual networks in technique 80 are interesting—both allow you to discover containers by a human-readable name, and Docker has the built-in ability to make this work across multiple nodes with overlay networks. The key difference is that Consul exists outside Docker and so may be easier to integrate into existing systems.

However, as mentioned at the beginning of this technique, Consul has another interesting feature we'll take a look at: health checks.

Health checking is a big topic, so we'll leave the minutiae for the comprehensive Consul documentation and look at one of the options for monitoring—a script check. This runs a command and sets the health based on the return value, with 0 for success, 1 for warning, and any other value for critical. You can register a health check when initially defining the service, or in a separate API call, as we'll do here.

```
c2 $ cat >check <<'EOF'
 #!/bin/sh
set -o errexit
set -o pipefail
```

Creates a check script verifying that the HTTP status code from the service is "200 OK". The service port is looked up from the service ID passed to the script as an argument.

```
SVC_ID="$1"
SVC_PORT=\
"$(wget -qO - 172.17.42.1:8500/v1/agent/services | jq ".$SVC_ID.Port")"
wget -qsO - "localhost:$SVC_PORT"
echo "Success!"
EOF
c2 $ cat check | docker exec -i consul-client sh -c \
'cat > /check && chmod +x /check'
```

Copies the check script into the Consul agent container

```
 c2 $ cat >health.json <<'EOF'
 {
  "Name": "filescheck",
  "ServiceID": "files",
  "Script": "/check files",
  "Interval": "10s"
}
EOF
```

Creates a health check definition to send to the Consul HTTP API. The service ID has to be specified in both the ServiceID field and the script command line.

```
c2 $ curl -X PUT --data-binary @health.json \
172.17.42.1:8500/v1/agent/check/register
```

Submits the health check JSON to the Consul agent

```
 c2 $ sleep 300
```

Waits for the check output to be communicated to the server agents

```
 c2 $ curl -sSL 172.17.42.1:8500/v1/health/service/files | \
python -m json.tool | head -n 13
```

Retrieves health check information for the check you've registered

```
 [
    {
        "Checks": [
            {
                "CheckID": "filescheck",
                "Name": "filescheck",
                "Node": "mylaptop2",
                "Notes": "",
                "Output": "/check: line 6: jq: not \
found\nConnecting to 172.17.42.1:8500 (172.17.42.1:8500)\n",
                "ServiceID": "files",
                "ServiceName": "files",
                "Status": "critical"
            },
c2 $ dig @$BRIDGEIP files.service.consul srv +short
 c2 $
```

Attempts to look up the files service, with no results

NOTE Because output from health checks can change on every execution (if it includes timestamps, for example), Consul only synchronizes check output with the server on a status change, or every five minutes (though this interval is configurable). Because statuses start as critical, there's no initial status change in this case, so you'll need to wait out the interval to get output.

We added a health check for the files service to be run every 10 seconds, but checking it shows the service as having a critical status. Because of this, Consul has automatically taken the failing endpoint out of the entries returned by DNS, leaving us with no servers. This is particularly helpful for automatically removing servers from a multiple-backend service in production.

The root cause of the error we've hit is an important one to be aware of when running Consul inside a container. All checks are also run inside the container, so, as the check script had to be copied into the container, you also need to make sure any commands you need are installed in the container. In this particular case, we're missing the `jq` command (a helpful utility for extracting information from JSON), which we can install manually, though the correct approach for production would be to add layers to the image.

```
c2 $ docker exec consul-client sh -c 'apk update && apk add jq'
fetch http://dl-4.alpinelinux.org/alpine/v3.2/main/x86_64/APKINDEX.tar.gz
v3.2.3 [http://dl-4.alpinelinux.org/alpine/v3.2/main]
OK: 5289 distinct packages available
(1/1) Installing jq (1.4-r0)
Executing busybox-1.23.2-r0.trigger
OK: 14 MiB in 28 packages
c2 $ docker exec consul-client sh -c \
'wget -qO - 172.17.42.1:8500/v1/agent/services | jq ".files.Port"'
8000
c2 $ sleep 15
c2 $ curl -sSL 172.17.42.1:8500/v1/health/service/files | \
python -m json.tool | head -n 13
[
    {
        "Checks": [
            {
```

```
        "CheckID": "filescheck",
        "Name": "filescheck",
        "Node": "mylaptop2",
        "Notes": "",
        "Output": "Success!\n",
        "ServiceID": "files",
        "ServiceName": "files",
        "Status": "passing"
    },
```

We've now installed `jq` onto the image using the Alpine Linux (see technique 57) package manager, verified that it works by manually executing the line that was previously failing in the script, and then waited for the check to rerun. It's now successful!

With script health checks, you now have a vital building block for constructing monitoring around your application. If you can express a health check as a series of commands you'd run in a terminal, you can get Consul to automatically run it—this isn't limited to HTTP status. If you find yourself wanting to check the status code returned by an HTTP endpoint, you're in luck, as this is such a common task that one of the three types of health checking in Consul is dedicated to it, and you don't need to use a script health check (we did so above for illustrative purposes).

The final type of health check, time to live, requires a deeper integration with your application. The status must be periodically set to healthy, or the check will automatically be set to failing. Combining these three types of health check gives you the power to build comprehensive monitoring on top of your system.

To round off this technique, we'll look at the optional Consul web interface that comes with the server agent image. It provides a helpful insight into the current state of your cluster. You can visit this by going to port 8500 on the external IP address of a server agent. In this case you'd want to visit `$EXTIP1:8500`. Remember that even if you're on a server agent host, `localhost` or `127.0.0.1` won't work.

DISCUSSION

We've covered a lot in this technique—Consul is a big topic! Fortunately, just as the knowledge you gained about utilizing key/value stores with etcd in technique 74 is transferable to other key/value stores (like Consul), this service-discovery knowledge is transferable to other tools offering DNS interfaces (SkyDNS being one you may come across).

The subtleties we covered related to using the host network stack and using external IP addresses are also transferable. Most containerized distributed tools requiring discovery across multiple nodes may have similar problems, and it's worth being aware of these potential issues.

TECHNIQUE 86 Automatic service registration with Registrator

The obvious downside of Consul (and any service discovery tool) so far is the overhead of having to manage the creation and deletion of service entries. If you integrate this into your applications, you'll have multiple implementations and multiple places it could go wrong.

Integration also doesn't work for applications you don't have complete control over, so you'll end up having to write wrapper scripts when starting up your database and the like.

PROBLEM

You don't want to manually manage service entries and health checks in Consul.

SOLUTION

Use Registrator.

This technique will build on top of the previous one and will assume you have a two-part Consul cluster available, as described previously. We'll also assume there are no services in it, so you may want to recreate your containers to start from scratch.

Registrator (http://gliderlabs.com/registrator/latest/) takes away much of the complexity of managing Consul services—it watches for containers to start and stop, registering services based on exposed ports and container environment variables. The easiest way to see this in action is to jump in.

Everything we do will be on the machine with the client agent. As discussed previously, no containers except the server agent should be running on the other machine.

The following commands are all you need to start up Registrator:

```
$ IMG=gliderlabs/registrator:v6
$ docker pull $IMG
[...]
$ ip addr | grep 'inet ' | grep -v 'lo$\|docker0$'
    inet 192.168.1.80/24 brd 192.168.1.255 scope global wlan0
$ EXTIP=192.168.1.80
$ ip addr | grep docker0 | grep inet
    inet 172.17.42.1/16 scope global docker0
$ BRIDGEIP=172.17.42.1
$ docker run -d --name registrator -h $(hostname)-reg \
-v /var/run/docker.sock:/tmp/docker.sock $IMG -ip $EXTIP -resync \
60 consul://$BRIDGEIP:8500 # if this fails, $EXTIP is an alternative
b3c8a04b9dfaf588e46a255ddf4e35f14a9d51199fc6f39d47340df31b019b90
$ docker logs registrator
2015/08/14 20:05:57 Starting registrator v6 ...
2015/08/14 20:05:57 Forcing host IP to 192.168.1.80
2015/08/14 20:05:58 consul: current leader  192.168.1.87:8300
2015/08/14 20:05:58 Using consul adapter: consul://172.17.42.1:8500
2015/08/14 20:05:58 Listening for Docker events ...
2015/08/14 20:05:58 Syncing services on 2 containers
2015/08/14 20:05:58 ignored: b3c8a04b9dfa no published ports
2015/08/14 20:05:58 ignored: a633e58c66b3 no published ports
```

The first couple of commands here, for pulling the image and finding the external IP address, should look familiar. This IP address is given to Registrator so it knows what IP address to advertise for the services. The Docker socket is mounted to allow Registrator to be automatically notified of container starts and stops as they happen. We've also told Registrator how it can connect to a Consul agent, and that we want all containers to be refreshed every 60 seconds. Because Registrator should automatically be notified of container changes, this final setting is helpful in mitigating the impact of Registrator possibly missing updates.

Now that Registrator is running, it's extremely easy to register a first service.

```
$ curl -sSL 172.17.42.1:8500/v1/catalog/services | python -m json.tool
{
    "consul": []
}
$ docker run -d -e "SERVICE_NAME=files" -p 8000:80 ubuntu:14.04.2 python3 \
-m http.server 80
3126a8668d7a058333d613f7995954f1919b314705589a9cd8b4e367d4092c9b
$ docker inspect 3126a8668d7a | grep 'Name.*/'
    "Name": "/evil_hopper",
$ curl -sSL 172.17.42.1:8500/v1/catalog/services | python -m json.tool
{
    "consul": [],
    "files": []
}
$ curl -sSL 172.17.42.1:8500/v1/catalog/service/files | python -m json.tool
[
    {
        "Address": "192.168.1.80",
        "Node": "mylaptop2",
        "ServiceAddress": "192.168.1.80",
        "ServiceID": "mylaptop2-reg:evil_hopper:80",
        "ServiceName": "files",
        "ServicePort": 8000,
        "ServiceTags": null
    }
]
```

The only effort we've had to put in when registering the service is passing an environment variable to tell Registrator what service name to use. By default, Registrator uses a name based on the container name component after the slash and before the tag: "mycorp.com/myteam/myimage:0.5" would have the name "myimage". Whether this is useful or you want to specify something manually will depend on your naming conventions.

The rest of the values are pretty much as you'd hope. Registrator has discovered the port being listened on, added it to Consul, and set a service ID that tries to give a hint about where you can find the container (which is why the hostname was set in the Registrator container).

DISCUSSION

Registrator is excellent at giving you a handle on a swiftly changing environment with a high churn of containers, making sure you don't need to worry about your service-creation checks being created.

In addition to service details, Registrator will pick up a number of pieces of information from environments if they're present, including tags, service names per port (if multiple), and using health checks (if you're using Consul as the data storage). All three types of Consul health checks can be enabled by specifying the check details in the environment in JSON—you can read more about this in the Consul section of the "Registrator Backends" documentation at http://gliderlabs.com/registrator/latest/user/backends/#consul, or revisit the previous technique to get a brief introduction to Consul health checks themselves.

Summary

- systemd units are useful for controlling container execution on a single machine.
- Dependencies can be expressed in systemd units to provide startup orchestration.
- Helios is a production-quality, simple, multi-host orchestration solution.
- Consul can hold information about your services, allowing dynamic service discovery.
- Registrator can automatically register container-based services into Consul.