SAMPLE CHAPTER

CoreOS IN ACTION

Running applications on Container Linux

Matt Bailey

MANNING

CoreOS in Action

by Matt Bailey

Chapter 3

brief contents

3 Expecting failure: fault tolerance in CoreOS

This chapter covers

- Monitoring and fault tolerance in CoreOS
- Getting your first complex service running
- Application architecture in the context of CoreOS

If you work in infrastructure or operations in any capacity, you'll understand the importance of monitoring systems. When the alarms go off, it's time to figure out what's happened. You might have also taken a crack at automating some of the most common fixes to problems or mitigated situations with disaster-recovery failover switches, multicasting, or a variety of other ways to react to failure. You probably also have an understanding that technology always finds a way to break. Hardware, software, connectivity, power grid—these are all things that wake us up in the middle of the night. If you've been working in operations for a while, you probably have the sense that although automating fault tolerance is possible, it's usually risky and difficult to maintain.

CoreOS tries to solve this problem; by providing generic abstractions for the state of your application distributed over a cluster, the implementation details of automating fault tolerance become much clearer and reusable. The next logical benefit of containers after abstracting the runtime from any particular machine is to allow that runtime to be portable across a network, thus decoupling any container from the failure of its host.

In this chapter, we'll expand on what you learned in chapters 1 and 2 and dive into more-complex examples of how to give your services greater resiliency and quicker failure recovery. We'll examine how to manage the ephemeral nature of application stacks and explore some high-level concepts of systems architecture and design and how they apply to CoreOS. By the end of this chapter, you'll have a good understanding of how to plan deployments of your applications to CoreOS; this will lead into chapter 4, where we'll move to production.

3.1 The current state of monitoring

If you've been in operations for any length of time, you've used some kind of monitoring system. Usually such systems look like the typical monitoring architectures shown in figures 3.1 and 3.2, or a combination.

Your monitoring system can either send out probes to gather information about a server and its services, as in figure 3.1, and/or an agent running on the server can report status to a monitoring system, as in figure 3.2. You've probably experienced the drawbacks of each approach. Probes are difficult to maintain, and they fire false positives; and agents can be just as difficult to maintain, while also adding load to your system and uncertainty around the agent's reliability. With etcd, CoreOS replaces much of the need for these systems by normalizing state information that's composed by the services.

With traditional monitoring setups, you usually assume that your monitoring system is at least as reliable as the thing it's monitoring. Sometimes you rely on third-party solutions for monitoring, and other times you end up monitoring your own monitoring system. As your infrastructure and applications grow, your monitoring

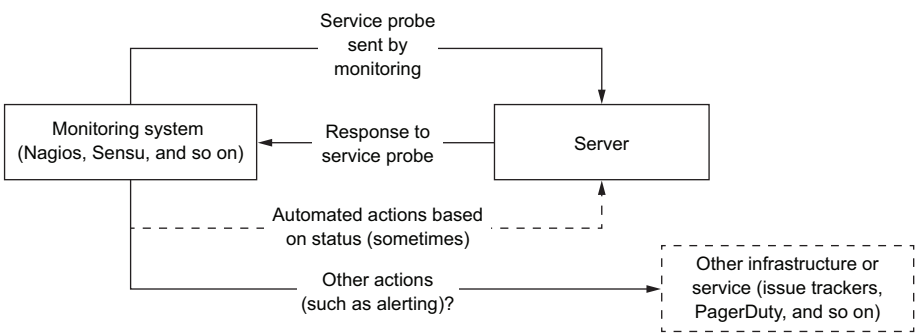

Figure 3.1 Monitoring with probes

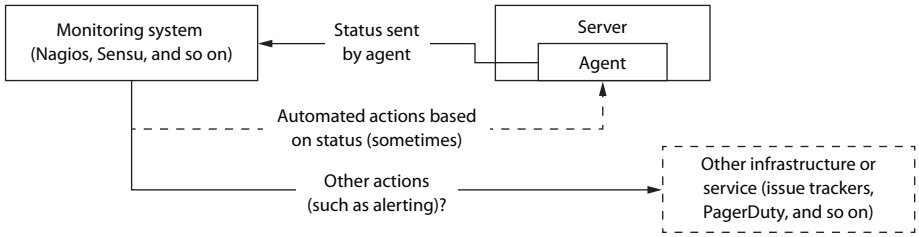

Figure 3.2 Monitoring with an agent

solutions increase in complexity right along with them; at the end of the day, monitoring tells you about the state of everything at once and usually doesn't do a great job of telling you why the state changed. If you're using things like public clouds, sometimes you can't even find out or don't care why it changed.

CoreOS lets you take a different approach to retaining observation of your live systems. To be clear, CoreOS doesn't do anything to preclude monitoring. What it *does* do is free your time to allow you to focus on monitoring what's important (your application), and not what isn't. (You're not in the business of maintaining operating systems, right?)

3.1.1 What's lacking

Consider this scenario. Your company runs a business-critical Rails application, and a cluster of Debian servers keeps it running. Maybe you've even got Chef keeping all the configurations in line. You've spent hours ensuring that log files are shipped off to a third-party log consumer.

One night, you get an alert from your monitoring system that that disks are full and your application isn't responding. Root-cause analysis time! Did the update you ran last month overwrite some of your log configs and begin writing logs to disk again? Did that new developer decide to write a new log file and not let you know? Did you miss something in your Chef config a year ago that slowly leaked data to disk where it shouldn't be? Is it a false positive? (Don't lie: you know the first thing you'd do is run `df` to see if the monitoring system was playing a trick on you.)

Finally, you find out you weren't purging your .deb files out of /var/cache often enough after you added a little automation around OS upgrades. A very small log file was being written to every day from a short cron job you added six months ago, and the combination brought everything down. At this point, you ask yourself, "What does any of this have to do with the application I'm supporting?" and "Why am I still solving the same system administration problems I was solving 10 years ago?"

Monitoring has become the tip of the iceberg—or maybe a better metaphor is a canary in a coal mine, reminding you that you missed an edge case. Can you keep up with edge cases as fast as they're created?

3.1.2 What CoreOS does differently

CoreOS takes back the responsibility of not letting your OS or its configuration be the downfall of your application:

- It's stripped down to eliminate a lot of configuration and administration problems out of the box.
- As we discussed in chapter 1, CoreOS takes advantage of containerization's ability to abstract your application from the OS, as well as as abstract it from the machine with fleet, to empower you to focus on your application and not OS internals.
- Application failures are contained, and machine failures are mitigated so that they can be handled outside of a maintenance window (or ignored in some public cloud scenarios).
- Maintenance of the OS is also done without the need for interaction.

You can forget the fear of OS upgrades for two reasons. First, the behavior of a CoreOS operating system upgrade from the perspective of your application is the same as the behavior of a machine outage: that downtime is avoided by fleet shifting around containers across the cluster to meet your specifications, regardless of the state of the cluster. And second, *because* everything is abstracted by containers, nothing in your application depends on anything in the base OS being available other than the handful of CoreOS services.

With these benefits in mind, see how figure 3.3 shows a CoreOS upgrade in progress. Although this level of OS automation might seem dangerous, the abstraction afforded by containers and fleet significantly reduces the impact. Essentially, this is CoreOS dogfooding its approach to providing fault tolerance for your applications onto the OS. The upgrade process is part of the equation of how CoreOS reduces the need for complex monitoring systems; the cluster-wide scheduling and discovery systems reveal a much more generic interface for gathering important data.

The default setting for upgrade-locking (`etcd-lock`) is to have only one machine upgrade in the cluster at a time. If the etcd cluster is in a problematic state, it won't upgrade any nodes. If you have a larger cluster, you can increase the number of nodes that can upgrade and reboot simultaneously with `locksmithctl`:

```
core@core-01 $ locksmithctl set-max 2
Old: 1
New: 2
```

NOTE Don't actually do this on your local three-node cluster! If two out of three nodes reboot at the same time, you'll lose the quorum in etcd. A quorum in etcd can tolerate up to $(N\text{-}1)/2$ failures, where N is the number of cluster members (machines).

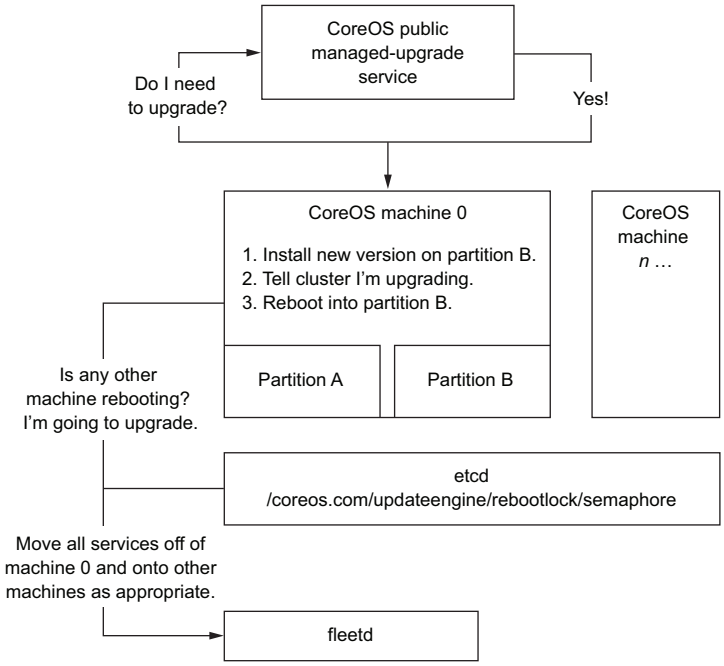

Figure 3.3 CoreOS upgrade process

Cluster upgrades

CoreOS operating system upgrades require some level of public internet access to *.release.core-os.net by default, via either an HTTP proxy or NAT. If you want more control over upgrades beyond the three release channels, CoreOS, Inc. (the company) provides a premium managed service to assist you.

Additionally, how you plan the capacity of your services should go hand in hand with how you plan your cluster and upgrade configuration. Upgrades will occur only when etcd has an available lock and has no errors (for example, another machine is down or rebooting for some reason other than an upgrade). If your services can't all live on a cluster with the performance you expect while missing two nodes, don't increase your `etcd-lock` max. But at a *minimum* you should plan for one machine outage. This isn't much different from scaling mass storage: the more redundant units, the higher your fault tolerance to some kinds of failure.

3.2 Service scheduling and discovery

In chapters 1 and 2, you learned a bit about etcd and fleet and how they provide service scheduling and discovery for your application. Together, they provide fault tolerance and composability for monitoring data within your application runtime, rather than from outside of it. We'll go a little deeper here and consider a more realistic example to illustrate how these things can fit together. We'll expand on the NGINX

example with an upstream Express example application, and we'll look at how to further use etcd in this application stack. In this example, NGINX will monitor the state of the Express application and act accordingly without the need for an outside monitoring system.

To observe how CoreOS can hedge your services against failure, you'll build out an application environment with fault tolerance built in. Then, you'll try to break it with partial failures in the cluster and observe how the fault tolerance reacts.

3.2.1 Deploying production NGINX and Express

A real-world example would involve at least a couple of tiers. We won't get into the complexities of database tiers yet (we will later!), but an application stack isn't really a stack unless some internal communication is going on. Say, for example, that you want to deploy an application that consists of some Express node services behind an instance of NGINX. Ultimately, you want your system to look like figure 3.4, which shows the simple network topology between NGINX and the Express applications behind it.

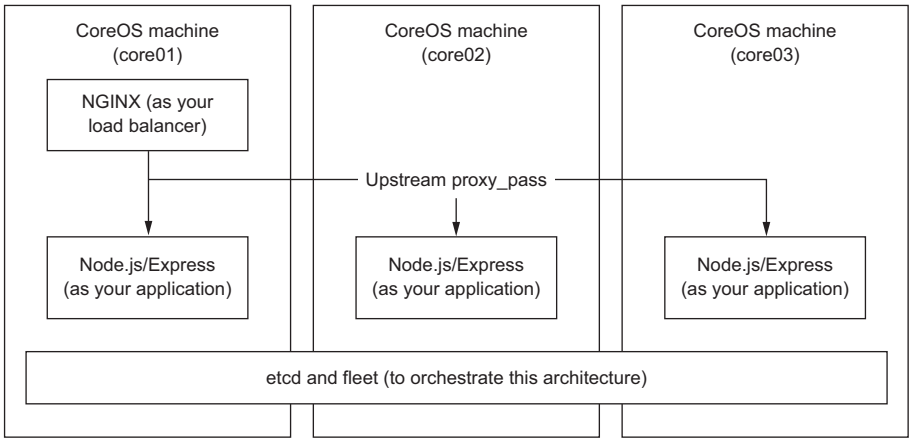

Figure 3.4 NGINX and Express stack

In this scenario, NGINX acts like a load balancer but could be performing any number of jobs (SSL termination, external reverse proxies, and so on). The next few sections set up this architecture; it's crucial for you to take away that the failure of any node becomes a non-concern as you build toward a fault-tolerant application instead of a monitoring-dependent one.

3.2.2 Using etcd for configuration

For this application stack, you'll use what you learned in chapter 2: you'll set up NGINX in a CoreOS cluster and add a fairly common back-end service. The example uses Node.js/Express mostly for simplicity, but it could be any HTTP service you want to distribute across your cluster.

I've added some significant complexity to the previous example, in the form of a new requirement to modify and deploy containers that are different from the publicly available Docker images. But I'll assume that you have a repository to which to upload custom built containers and that you're using the public, official Docker registry at https://hub.docker.com.

For the sake of the example, assume that it's okay to publish your containers to Docker's public repository. In the real world, of course, this might not be possible. There are many options for publishing private Docker images, using software-as-a-service (SaaS) products or hosting your own repository, but that's beyond the scope of this book. For further reading, check out *Docker in Action* by Jeff Nickoloff (Manning, 2016, www.manning.com/books/docker-in-action).

The Express application

Let's start with your Express instance. First you need to create a "Hello World" Express app. You don't need any experience with Node.js for this; you can paste the code from listings 3.1, 3.2, and 3.3 into files in a new directory.

Listing 3.1 code/ch3/helloworld/app.js

```
const app = require('express')()
app.get('/', (req, res) => { res.send('hello world').end() })
app.listen(3000)
```

Listing 3.2 code/ch3/helloworld/Dockerfile

```
FROM node:5-onbuild
EXPOSE 3000
```

Listing 3.3 code/ch3/helloworld/package.json

```
{
  "name": "helloworld",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4"
  }
}
```

Next, build the image and push it to the Docker hub. You can do all this on a CoreOS instance (because it has Docker running) or anywhere else you may be running Docker, such as your workstation:

```
$ cd code/ch3/helloworld
$ docker build -t mattbailey/helloworld .
Sending build context to Docker daemon 1.166 MB
...
Successfully built f8945e023a8c

$ docker login  # IF NECESSARY
$ docker push mattbailey/helloworld
```

```
The push refers to a repository [docker.io/mattbailey/helloworld]
...
latest: digest: sha256:e803[...]190e size: 12374
```

You can drop your .service files in this directory as well. It's somewhat common to keep these service files under the same source control as the project. You'll have a main service file and a sidekick.

The first service file looks at lot like what you saw with NGINX, but you reference the Docker image you published earlier.

Listing 3.4 code/ch3/helloworld/helloworld@.service

```
[Unit]
Description=Hello World Service
Requires=docker.service
After=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill helloworld
ExecStartPre=-/usr/bin/docker rm -f helloworld
ExecStartPre=/usr/bin/docker pull mattbailey/helloworld:latest
ExecStart=/usr/bin/docker run --name helloworld \
  -p 3000:3000 mattbailey/helloworld:latest
ExecStop=-/usr/bin/docker stop helloworld

[X-Fleet]
Conflicts=helloworld@*
```

What is TimeoutStartSec?

Notice that you use `TimeoutStartSec=0` in listing 3.4, to indicate that you don't want a timeout for this service. This can be helpful on slower connections or with larger Docker images that may take a while to pull, especially if you're pulling them all at the same time in three VMs on a single workstation.

You may want to tune this setting in the future depending on your use cases (you could, for example, set it from etcd), but it's easier to have no timeout while you're testing and developing services.

The sidekick also looks similar: it announces the presence of the `helloworld` service in /services/helloworld/.

Listing 3.5 code/ch3/helloworld/helloworld-sidekick@.service

```
[Unit]
Description=Register Hello World %i
BindsTo=helloworld@%i.service
After=helloworld@%i.service

[Service]
TimeoutStartSec=0
```

```
EnvironmentFile=/etc/environment
ExecStartPre=/usr/bin/etcdctl set /services/changed/helloworld 1
ExecStart=/bin/bash -c 'while true; \
  do \
    [ "`etcdctl get /services/helloworld/${COREOS_PUBLIC_IPV4}`" \
      != "server ${COREOS_PUBLIC_IPV4}:3000;" ] && \
    etcdctl set /services/changed/helloworld 1; \
    etcdctl set /services/helloworld/${COREOS_PUBLIC_IPV4} \
      \'server ${COREOS_PUBLIC_IPV4}:3000;\' \
      --ttl 60;sleep 45;done'
ExecStop=/usr/bin/etcdctl rm /services/helloworld/helloworld@%i
ExecStopPost=/usr/bin/etcdctl set /services/changed/helloworld 1

[X-Fleet]
MachineOf=helloworld@%i.service
```

Organizing etcd keys

There are no strict guidelines or preset structures for how to organize your etcd keys—doing so is completely free-form.

You will, of course, want to plan this structure much as you'd plan your infrastructure, to keep things appropriately namespaced and flexible enough to accommodate your future needs.

Now, you can fire up `helloworld` on your cluster and verify that it has started:

```
$ fleetctl start code/ch3/helloworld/helloworld@{1..3}.service
Unit helloworld@1.service inactive
Unit helloworld@2.service inactive
Unit helloworld@3.service inactive
$ fleetctl start code/ch3/helloworld/helloworld-sidekick@{1..3}.service
Unit helloworld-sidekick@1.service inactive
Unit helloworld-sidekick@2.service inactive
Unit helloworld-sidekick@3.service inactive
```

Also, verify that `helloworld` is running:

```
$ fleetctl list-units
UNIT        MACHINE       ACTIVE  SUB
helloworld-sidekick@1.service a12d26db.../172.17.8.102  active  running
helloworld-sidekick@2.service c1fc6b79.../172.17.8.103  active  running
helloworld-sidekick@3.service c37d052c.../172.17.8.101  active  running
helloworld@1.service    a12d26db.../172.17.8.102  active  running
helloworld@2.service    c1fc6b79.../172.17.8.103  active  running
helloworld@3.service    c37d052c.../172.17.8.101  active  running
$ curl 172.17.8.101:3000
hello world
$ etcdctl ls /services/helloworld/
/services/helloworld/172.17.8.101
/services/helloworld/172.17.8.103
/services/helloworld/172.17.8.102
```

The next section moves on to the NGINX configuration.

THE NGINX APPLICATION

Create a new directory for your NGINX build. You'll have three files for configuring NGINX, not including the service units. The first is a fairly simple Dockerfile using the official NGINX image as its base.

Listing 3.6 code/ch3/nginx/Dockerfile

```
FROM nginx

COPY helloworld.conf /tmp/helloworld.conf
COPY start.sh /tmp/start.sh
RUN chmod +x /tmp/start.sh

EXPOSE 80

CMD ["/tmp/start.sh"]
```

Next is a start script. You'll using Bash as the dynamic runtime configuration for simplicity, so you won't add any more dependencies to the example. But many tools are available to help you template your configuration files at runtime, such as confd (www.confd.io).

Listing 3.7 code/ch3/nginx/start.sh

```
#!/usr/bin/env bash

# Write dynamic nginx config
echo "upstream helloworld { ${UPSTREAM} }" > /etc/nginx/conf.d/default.conf

# Write rest of static config
cat /tmp/helloworld.conf >> /etc/nginx/conf.d/default.conf

# Now start nginx
nginx -g 'daemon off;'
```

Finally, here's the static NGINX config file for the reverse proxy.

Listing 3.8 code/ch3/nginx/helloworld.conf

```
server {
    listen       80;
    location / {
        proxy_pass http://helloworld;
    }
}
```

Build and push this image to your repository, just as you did the Express app:

```
$ cd code/ch3/nginx/
$ docker build -t mattbailey/helloworld-nginx .
Sending build context to Docker daemon 4.096 kB
...
Successfully built e9cfe4f5f144

$ docker push mattbailey/helloworld-nginx
```

```
The push refers to a repository [docker.io/mattbailey/helloworld-nginx]
...
latest: digest: sha256:01e4[...]81f8 size: 7848
```

Now, you can write your service files, shown in listings 3.9 and 3.10.

Listing 3.9 code/ch3/nginx/helloworld-nginx.service

```
[Unit]
Description=Hello World Nginx
Requires=docker.service
After=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill helloworld-nginx
ExecStartPre=-/usr/bin/docker rm -f helloworld-nginx
ExecStartPre=/usr/bin/docker pull mattbailey/helloworld-nginx:latest
ExecStart=/bin/sh -c /for host in `etcdctl ls /services/helloworld`; \
  do UPSTREAM=$UPSTREAM`etcdctl get $host`; \
  done; \
  docker run -t -e UPSTREAM="$UPSTREAM" \
    --name helloworld-nginx -p 80:80 mattbailey/helloworld-nginx:latest'
ExecStop=-/usr/bin/docker stop helloworld-nginx
```

Listing 3.10 code/ch3/nginx/helloworld-nginx-sidekick.service

```
[Unit]
Description=Restart Nginx On Change

[Service]
ExecStart=/usr/bin/etcdctl exec-watch \
  /services/changed/helloworld -- \
  /bin/sh -c "fleetctl stop helloworld-nginx.service; \
  fleetctl start helloworld-nginx.service"
```

Next, start your NGINX service units:

```
$ fleetctl start code/ch3/nginx/helloworld-nginx.service
Unit helloworld-nginx.service inactive
Unit helloworld-nginx.service launched on a12d26db.../172.17.8.102
$ fleetctl start code/ch3/nginx/helloworld-nginx-sidekick.service
Unit helloworld-nginx-sidekick.service inactive
Unit helloworld-nginx-sidekick.service launched on a12d26db.../172.17.8.102
```

Notice that you don't care which machine the sidekick runs on for NGINX, because it's interacting with NGINX entirely via `etcdctl` and `fleetctl`.

You should now have a setup that looks like figure 3.4. NGINX is effectively watching for changes in the topology of Express applications and is set up to adapt to those changes. Further, you did this without implementing any complex monitoring systems. You expect failure to occur, and CoreOS lets you integrate that notion into the composition of the service architecture. You need to test this notion; so, in the next section, you'll see what happens when a machine fails.

3.3 Breaking things

Now that you have a "production-like" deployment in place, it's time to try to break it! What you've built should stand up pretty well to a single machine failure. We'll look at how a machine failure affects your application and the how CoreOS can bring the cluster back together when it's restored. Simulating more complex scenarios is a little difficult on a local cluster of three machines; but as a baseline, the CoreOS cluster considers any inability to resolve a node in etcd as a machine failure and will react as if a machine is down. As mentioned in section 3.1.2, etcd can survive $(N\text{-}1)/2$ machine failures, where N is the number of machines; because etcd is the source of truth for your cluster state, your deployment of CoreOS machines (virtual or physical) should consider this rate of failure a baseline.

3.3.1 Simulating a machine failure

The most destructive kind of scenario you can simulate is a complete failure of a CoreOS machine. This scenario includes loss of network connectivity, because that's functionally equivalent to the CoreOS cluster. To simulate this, you'll have to shut down one of your machines. To make things interesting, you'll shut down the machine that's also running NGINX, which will result in an outage, but one that is mitigated by fleet. You may want to open another terminal to a machine you're not shutting down to watch what happens:

```
$ vagrant ssh core-01
core@core-01 ~ $ fleetctl journal -f helloworld-nginx.service
...
Feb 17 05:00:59 core-02 systemd[1]: Started Hello World Nginx.
```

> **When failure isn't "failure"**
>
> In some scenarios, losing a machine in your cluster is intentional and doesn't represent a fault of any kind. For example, this happens if you have CoreOS automatic OS updates enabled, or you need to shut down some infrastructure for maintenance, or you want to rebuild your AWS EC2 instance for any number of reasons. If you consider machine "faults" to be occurrences that are part of the normal lifecycle of your systems, you'll have a much easier time gaining the benefits of CoreOS.

In a different terminal from your host, have Vagrant shut down the machine where `helloworld-nginx.service` is running:

```
$ vagrant halt core-02
```

Watch on core-01 or any other machine that wasn't running `helloworld-nginx.service`:

```
...
Connection to 127.0.0.1 closed by remote host.
Error running remote command: wait: remote command exited without exit status
      or exit signal
```

```
core@core-01 ~ $ fleetctl journal -f helloworld-nginx.service
...
Feb 17 05:16:32 core-01 systemd[1]: Started Hello World Nginx.
```

You can see that the service was shut down on core-02, and then fleet moved it to core-01. You can also observe that NGINX has picked up the new upstream configuration:

```
core@core-01 ~ $ docker exec -it helloworld-
     nginx cat /etc/nginx/conf.d/default.conf
upstream helloworld { server 172.17.8.101:3000;server 172.17.8.103:3000; }
server {
    listen       80;
    location / {
        proxy_pass http://helloworld;
    }
}
```

Now that you've seen your application stack adapt to a missing machine, in the next section you'll bring the machine back to see how the cluster deals with service restoration.

3.3.2 Self-repair

Bring the machine back up, and watch everything go back to normal:

```
$ vagrant up core-02
```

Once it's booted back up, wait about 45 seconds. Then you can confirm that the machine is back in NGINX's upstream:

```
core@core-01 ~ $ docker exec -it helloworld-
     nginx cat /etc/nginx/conf.d/default.conf
upstream helloworld { server 172.17.8.101:3000;server 172.17.8.103:3000;
     server 172.17.8.102:3000; }
server {
    listen       80;
    location / {
        proxy_pass http://helloworld;
    }
}
```

The upstream is again pointing to all three of your Express applications. It took relatively little engineering to add fault tolerance to a system completely unfamiliar with that concept. Additionally, you didn't need to employ any additional tools to accomplish this, other than what is provided by CoreOS. Ultimately, building robust, self-repairing systems is always a hard problem, but CoreOS provides a generic tool set with fleet and etcd that gives you a pattern for building it into many scenarios.

Application architecture is still an important skill. And adapting your architecture to CoreOS requires some planning, as we'll discuss next.

3.4 Application architectures and CoreOS

Application architecture is a topic that could fill many volumes. This won't be the last time we discuss it in this book; but it's worth looking at it and at how it relates to the big picture, now that you've simulated things that application architects try to plan for.

First we'll look at some common pitfalls with designing applications for failure, and then we'll follow up with a discussion of what parts of the architecture you can control. Finally, we'll touch on what all this means with respect to configuration management.

3.4.1 Common pitfalls

There are some common pitfalls when it comes to running application stacks in environments where faults are common or expected, or where the scale of what you're doing statistically demands that faults will occur at some regular interval. You can probably recognize in the chapter's example that the host on which NGINX is running becomes somewhat of a single point of failure. Even though you've designed the system to tolerate that machine's failure by starting up NGINX on another instance, you still could have a gap in availability. You can resolve this in your architecture in a few ways:

- The NGINX sidekick can update a DNS entry with a short TTL if you can tolerate a minute of downtime.
- You can rely on upstream content delivery network (CDN) caching to carry you through an outage.
- You can run NGINX on two or all three machines and have a load-balancer appliance or something like AWS Elastic Load Balancer (ELB) with a health check in front of them.

Most commonly, the last option is used if you need that level of reliability. You're building enough vertical capacity into your machines to run both services at the same time, so there's little reason not to. But here's where you have to be careful. Assume that NGINX is doing something specific for a user's session. This isn't likely; but for the sake of an example, if NGINX stored some kind of state locally, that state wouldn't be shared to the other NGINX service running on the other machine. Often, you accept that users may be logged out if some part of a cluster fails, but you also wouldn't want them to be logged out by hitting a different node behind your load balancer.

The architectural choices you make, especially with respect to the software you use, have an effect on your ability to make the architecture fault tolerant with CoreOS's tools. Even the complexity of applying fault tolerance to software that supports it can be difficult. For example, before Redis 3.0 and the `redis-cluster` feature that comes with it, clustering Redis involved a separate sentinel process to elect a write master and realign the cluster. The Redis Sentinel system was designed to be applied in a fault-tolerant system like CoreOS, but making it work was a complex task. The takeaway is that you should *always* test your cluster configurations and fault scenarios in an environment like a local Vagrant cluster, where you can control conditions.

3.4.2 Greenfield and legacy systems

Sometimes you get to choose your architecture, and sometimes you don't. Dealing with legacy systems is part of every engineer's career; obviously, it will be easier to build fault tolerance into a greenfield project via CoreOS than to build it into a legacy stack. You may find that it's impossible to reach certain levels of reliability in some systems that you could with others. You can, however, mitigate some of the risk with the patterns CoreOS provides.

Mostly you'll run into issues with legacy services that store some kind of state and have no way to distribute it. Of those, the single most annoying problem is the "undistributable" state being stored on the local filesystem. If the data that's being stored isn't important, the only downside is that you can only run the service on one machine; you can still rely on fleet to move it around. If the data *is* important, and you can't change how it works, you'll have to implement distributed storage. We'll go into detail about your options in section 4.5.

3.4.3 Configuration management

If you're dealing with greenfield applications, your approach to configuration management should assume that the application configuration is split between configuration that needs to understand the runtime environment (such as database IPs) and configuration that's stateless (such as a database driver). The former should be managed with etcd, and the latter should be managed with your container build process. With that in mind, you'll no longer need complex configuration-management systems, and your software environment will become much more repeatable and understandable.

3.5 Summary

- Follow the sidekick pattern to build complex application environments with service discovery.
- Use service discovery to implement fault tolerance and self-healing capabilities.
- Design scenarios in which you can simulate failures that you might see in a production environment, so you can test your cluster implementations.
- Application architectures are important in planning your CoreOS deployments and always require review.

LINUX/SOFTWARE DEVELOPMENT

CoreOS IN ACTION

Matt Bailey

Traditional Linux server distributions include every component required for anything you might be hosting, most of which you don't need if you've containerized your apps and services. CoreOS Container Linux is a bare-bones distro with only the essential bits needed to run containers like Docker. Container Linux is a fast, secure base layer for any container-centric distributed application, including microservices. And say goodbye to patch scheduling; when Container Linux needs an update, you just hot-swap the whole OS.

CoreOS in Action is a clear tutorial for deploying container-based systems on CoreOS Container Linux. Inside, you'll follow along with examples that teach you to set up CoreOS on both private and cloud systems, and to practice common sense monitoring and upgrade techniques with real code. You'll also explore important container-aware application designs, including microservices, web, and Big Data examples with real-world use cases to put your learning into perspective.

What's Inside

- Handling scaling and failures gracefully
- Container-driven application designs
- Cloud, on-premises, and hybrid deployment
- Smart logging and backup practices

Written for readers familiar with Linux and the basics of Docker.

Matt Bailey has 15 years of experience on everything from large-scale computing cluster architecture to front-end programming.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/coreos-in-action

"A useful compass to guide you through the complex paths of CoreOS and microservices."
—Marco Zuppone
Gemalto SafeNet

"A very practical introduction with realistic deployment scenarios."
—Michael Bright
Hewlett-Packard Enterprise

"Great source, carefully crafted ... offers compelling, in-depth insight."
—Antonis Tsaltas
Huawei Technologies

"Everything you need to get started, from the basic building blocks to advanced architectures."
—Thomas Peklak
Emakina CEE, Austria

ISBN-13: 978-1-61729-374-0
ISBN-10: 1-61729-374-1
54499
9 781617 293740

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]