Docker in Production

LESSONS FROM THE TRENCHES

Written by: Joe Johnston, Antoni Batchelli, Justin Cormack
John Fiedler, Milos Gajdos

BLEEDING EDGE PRESS

Docker in Production

Lessons from the Trenches

Joe Johnston, Antoni Batchelli, Justin Cormack, John Fiedler, Milos Gajdos

Docker in Production

ISBN 9781939902184
Published by: Bleeding Edge Press, Santa Rosa, CA 95404
Title: Docker in Production
Authors: Joe Johnston, Antoni Batchelli, Justin Cormack, John Fiedler, Milos Gajdos
Editor: Troy Mott
Copy Editor: Christina Rudloff
Cover Design: Bob Herbstman
Website: bleedingedgepress.com

Table of Contents

Preface

Docker is the new sliced bread of infrastructure. Few emerging technologies compare to how fast it swept the DevOps and infrastructure scenes. In less than two years, Google, Amazon, Microsoft, IBM, and nearly every cloud provider announced support for running Docker containers. Dozens of Docker related startups were funded by venture capital in 2014 and early 2015. Docker, Inc., the company behind the namesake open source technology, was valued at about $1 billion USD during their Series D funding round in Q1 2015.

Companies large and small are converting their apps to run inside containers with an eye towards service oriented architectures (SOA) and microservices. Attend any DevOps meet-up from San Francisco to Berlin or peruse the hottest company engineering blogs, and it appears the ops leaders of the world now run on Docker in the cloud.

No doubt, containers are here to stay as crucial building blocks for application packaging and infrastructure automation. But there is one thorny question that nagged this book's authors and colleagues to the point of motivating another Docker book.

Who is This Book For?

Readers with intermediate to advanced DevOps and ops backgrounds will likely gain the most from this book. Previous experience with both the basics of running servers in production as well as creating and managing containers is highly recommended.

Many books and blog posts already cover individual topics related to installing and running Docker, but few resources exist to weave together the myriad and sometimes forehead-to-wall-thumping concerns of running Docker in production. Fear not, if you enjoyed the movie Inception, you will feel right at home running containers in virtual machines on servers in the cloud.

This book will give you a solid understanding of the building blocks and concerns of architecting and running Docker-based infrastructure in production.

Who is Actually Using Docker in Production?

Or more poignantly, how do you navigate the hype to successfully address real world production issues with Docker? This book sets out to answer these questions through a mix of

interviews, end-to-end production examples from real companies, and referable topic chapters from leading DevOps experts. Although this book contains useful examples, it is not a copy-and-paste "how-to" reference. Rather, it focuses on the practical theories and experience necessary to evaluate, derisk and operate bleeding-edge technology in production environments.

As authors, we hope the knowledge contained in this book will outlive the code snippets by providing a solid decision tree for teams evaluating how and when to adopt Docker related technologies into their DevOps stacks.

Running Docker in production gives companies several new options to run and manage server-side software. There are many readily available use cases on how to use Docker, but few companies have publicly shared their full-stack production experiences. This book is a compilation of several examples of how the authors run Docker in production as well as a select group of companies kind enough to contribute their experience.

Why Docker?

The underlying container technology used by Docker has been around for many years, even before **dotCloud**, the Platform-as-a-Service startup, pivoted to become Docker as we now know it. Before dotCloud, many notable companies like **Heroku** and **Iron.io** were running large scale container clusters in production for added performance benefits over virtual machines. Running software in containers instead of virtual machines gave these companies the ability to spin up and down instances in seconds instead of minutes, as well as run more instances on fewer machines.

So why did Docker take off if the technology wasn't new? Mainly, ease of use. Docker created a unified way to package, run, and maintain containers from convenient CLI and HTTP API tools. This simplification lowered the barrier to entry to the point where it became feasible--and fun--to package applications and their runtime environments into self-contained images rather than into configuration management and deployment systems like Chef, Puppet, and Capistrano.

Fundamentally, Docker changed the interface between developer and DevOps teams by providing a unified means of packaging the application and runtime environment into one simple Dockerfile. This radically simplified the communication requirements and boundary of responsibilities between devs and DevOps.

Before Docker, epic battles raged within companies between devs and ops. Devs wanted to move fast, integrate the latest software and dependencies, and deploy continuously. Ops were on call and needed to ensure things remained stable. They were the gatekeepers of what ran in production. If ops was not comfortable with a new dependency or requirement, they often ended up in the obstinate position of restricting developers to older software to ensure bad code didn't take down an entire server.

In one fell swoop, Docker changed the roll of DevOps from a "mostly say no" to a "yes, if it runs in Docker" position where bad code only crashes the container, leaving other serv-

ices unaffected on the same server. In this paradigm, DevOps are effectively responsible for providing a PaaS to developers, and developers are responsible for making sure their code runs as expected. Many teams are now adding developers to PagerDuty to monitor their own code in production, leaving DevOps and ops to focus on platform uptime and security.

Development vs. Production

For most teams, the adoption of Docker is being driven by developers wanting faster iterations and release cycles. This is great for development, but for production, running multiple Docker containers per host can pose security challenges, which we cover in chapter 10 on Security. In fact, almost all conversations about running Docker in production are dominated by two concerns that separate development environments from production: 1) orchestration and 2) security.

Some teams try to mirror development and production environments as much as possible. This approach is ideal but often not practical due to the amount of custom tooling required or the complexity of simulating cloud services (like AWS) in development.

To simplify the scope of this book, we cover use cases for deploying code but leave the exercise of determining the best development setup to the reader. As a general rule, always try to keep production and development environments as similar as possible and use a continuous integration / continuous deliver (CI/CD) system for best results.

What We Mean by *Production*

Production means different things to different teams. In this book, we refer to production as the environment that runs code for real customers. This is in contrast to development, staging, and testing environments where downtime is not noticed by customers.

Sometimes Docker is used in production for containers that receive public network traffic, and sometimes it is used for asynchronous, background jobs that process workloads from a queue. Either way, the primary difference between running Docker in production vs. any other environment is the additional attention that must be given to security and stability.

A motivating driver for writing this book was the lack of clear distinction between actual production and other envs in Docker documentation and blog posts. We wagered that four out of five Docker blog posts would recant (or at least revise) their recommendations after attempting to run in production for six months. Why? Because most blog posts start with idealistic examples powered by the latest, greatest tools that often get abandoned (or postponed) in favor of simpler methods once the first edge case turns into a showstopper. This is a reflection on the state of the Docker technology ecosystem more than it is a flaw of tech bloggers.

Bottom line, production is hard. Docker makes the work flow from development to production much easier to manage, but it also complicates security and orchestration (see chapter 4 for more on orchestration).

To save you time, here are the cliff notes of this book.

All teams running Docker in production are making one or more concessions on traditional security best practices. If code running inside a container can not be fully trusted, a one-to-one container to virtual machine topology is used. The benefits of running Docker in production outweigh security and orchestration issues for many teams. If you run into a tooling issue, wait a month or two for the Docker community to fix it rather than wasting time patching someone else's tool. Keep your Docker setup as minimal as possible. Automate everything. Lastly, you probably need full-blown orchestration (Mesos, Kubernetes, etc.) a lot less than you think.

Batteries Included vs. Composable Tools

A common mantra in the Docker community is "batteries included but removable." This refers to monolithic binaries with many features bundled in as opposed to the traditional Unix philosophy of smaller, single purpose, pipeable binaries.

The monolithic approach is driven by two main factors: 1) desire to make Docker easy to use out of the box, 2) golang's lack of dynamic linking. Docker and most related tools are written in Google's **Go programming language**, which was designed to ease writing and deploying highly concurrent code. While Go is a fantastic language, its use in the Docker ecosystem has caused delays in arriving at a pluggable architecture where tools can be easily swapped out for alternatives.

If you are coming from a Unix sysadmin background, your best bet is to get comfortable compiling your own stripped down version of the docker daemon to meet your production requirements. If you are coming from a dev background, expect to wait until Q3/Q4 of 2015 before Docker plugins are a reality. In the meantime, expect tools within the Docker ecosystem to have significant overlap and be mutually exclusive in some cases.

In other words, half of your job of getting Docker to run in production will be deciding on which tools make the most sense for your stack. As with all things DevOps, start with the simplest solution and add complexity only when absolutely required.

As of May, 2015, Docker, Inc., released **Compose**, **Machine**, and **Swarm** that compete with similar tools within the Docker ecosystem. All of these tools are optional and should be evaluated on merit rather than assumption that the tools provided by Docker, Inc., are the best solution.

Another key piece of advice in navigating the Docker ecosystem is to evaluate each open source tool's funding source and business objective. Docker, Inc., and **CoreOS** are frequently releasing tools at the moment to compete for mind and market share. It is best to wait a few months after a new tool is released to see how the community responds rather than switch to the latest, greatest tool just because it seems cool.

What Not to Dockerize

Last but not least, expect to not run everything inside a Docker container. Heroku-style **12 factor** apps are the easiest to Dockerize since they do not maintain state. In an ideal microservices environment, containers can start and stop within milliseconds without impacting the health of the cluster or state of the application.

There are startups like **ClusterHQ** working on Dockerizing databases and stateful apps, but for the time being, you will likely want to continue running databases directly in VMs or bare metal due to orchestration and performance reasons.

Any app that requires dynamic resizing of CPU and memory requirements is not yet a good fit for Docker. There is work being done to allow for dynamic resizing, but it is unclear when this will become available for general production use. At the moment, resizing a container's CPU and memory limitations requires stopping and restarting the container.

Also, apps that require high network throughput are best optimized without Docker due to Docker's use of iptables to provide NAT from the host IP to container IPs. It is possible to disable Docker's NAT and improve network performance, but this is an advanced use case with few examples of teams doing this in production.

Authors

As authors, our primary goal was to organize and distribute our knowledge as expediently as possible to make it useful to the community. The container and Docker infrastructure scene is evolving so fast, there was little time for a traditional print book.

This book was written over the course of a few months by a team of five authors with extensive experience in production infrastructure and DevOps. The content is timely, but care was also given to ensure the concepts are able to stand the test of time.

Joe Johnston is a full-stack developer, entrepreneur, and advisor to startups in San Francisco. He co-founded Airstack, a microservices infrastructure startup, as well as California Labs and Connect.Me. **@joejohnston**

John Fiedler is the Director of Engineering Operations at RelateIQ. His team focuses on Docker based solutions to power their SaaS infrastructure and developer operations. **@johnfielder**

Justin Cormack is a consultant especially interested in the opportunities for innovation made available by open source software, the cloud, and distributed systems. He is currently working on unikernels. You can find him on **github**. **@justincormack**

Antoni Batchelli is the Vice President of Engineering at **PeerSpace** and co-founder of **PalletOps**, an infrastructure automation consultancy. When he is not thinking about mixing functional programming languages with infrastructure he is thinking about helping engineering teams build awesome software. **@tbatchelli**

Milos Gajdos is an independent consultant, Infrastructure Tsar at Infrahackers Ltd., helping companies understand Linux container technology better and implement container based infrastructures. He occasionally blogs about **containers**. **@milosgajdos**

Technical Reviewers

We would like to the thank the following technical reviewers for their early feedback and careful critiques: Mika Turunen, Xavier Bruhiere, and Felix Rabe.

1 Getting Started

The first task of setting up a Docker production system is to understand the terminology in a way that helps visualize how components fit together. As with any rapidly evolving technology ecosystem, it's safe to expect over ambitious marketing, incomplete documentation, and outdated blog posts that lead to a bit of confusion about what tools do what job.

Rather than attempting to provide a unified thesaurus for all things Docker, we'll instead define terms and concepts in this chapter that remain consistent throughout the book. Often, our definitions are compatible with the ecosystem at large, but don't be too surprised if you come across a blog post that uses terms differently.

In this chapter, we'll introduce the core concepts of running Docker in production, and containers in general, without actually picking specific technologies. In subsequent chapters, we'll cover real-world production use cases with details on specific components and vendors.

Terminology

Let's take a look at the Docker terminology we use in this book.

Image vs. Container

- Image is the filesystem snapshot or tarball.
- Container is what we call an image when it is run.

Containers vs. Virtual Machines

- VMs hold complete OS and application snapshots.
- VMs run their own kernel.
- VMs can run OSs other than Linux.
- Containers only hold the application, although the concept of an application can extend to an entire Linux distro.

- Containers share the host kernel.
- Containers can only run Linux, but each container can contain a different distro and still run on the same host.

CI/CD: Continuous Integration / Continuous Delivery

System for automatically building new images and deploying them whenever application new code is committed or upon some other trigger.

Host Management

The process for setting up--provisioning--a physical server or virtual machine so that it's ready to run Docker containers.

Orchestration

This term means many different things in the Docker ecosystem. Typically, it encompasses scheduling and cluster management but sometimes also includes host management.

In this book we use *orchestration* as a loose umbrella term that encompasses the process of scheduling containers, managing clusters, linking containers (discovery), and routing network traffic. Or in other words, orchestration is the controller process that decides where containers should run and how to let the cluster know about the available services.

Scheduling

This is deciding which containers can run on which hosts given resource constraints like CPU, memory, and IO.

Discovery

The process of how a container exposes a service to the cluster and discovers how to find and communicate with other services. A simple use case is a web app container discovering how to connect to the database service.

Docker documentation refers to *linking* containers, but production grade systems often use a more sophisticated discovery mechanism.

Configuration Management

Configuration management is often used to refer to pre-Docker automation tools like Chef and Puppet. Most DevOps teams are moving to Docker to eliminate many of the complications of configuration management systems.

In many of the examples in this book, configuration management tools are only used to provision hosts with Docker and very little else.

Development to Production

This book focuses on Docker in production, or non-development environments, which means we will spend very little time on configuring and running Docker in development. But since all servers run code, it is worth a brief discussion on how to think about application code in a Docker versus a non-Docker system.

Unlike traditional configuration management systems like Chef, Puppet, and Ansible, Docker is best used when application code is pre-packaged into a Docker image. The image typically contains all of the application code as well as any runtime dependencies and system requirements. Configuration files containing database credentials and other secrets are often added to the image at runtime rather than being built into the image.

Some teams choose to manually build Docker images on dev machines and push them to image repositories that are used to pull images down onto production hosts. This is the simple use case. It works, but it is not ideal due to workflow and security concerns.

A more common production example is to use a CI/CD system to automatically build new images whenever application code or Dockerfiles change.

Multiple Ways to Use Docker

Over the years, technology has changed significantly from physical servers to virtual servers to clouds with platform-as-a-service (PaaS) environments. Docker images can be used in current environments without heavy lifting or with completely new architectures. It is not necessary to immediately migrate from a monolithic application to a service oriented architecture to use Docker. There are many use cases that allow for Docker to be integrated at different levels.

A few common Docker uses:

- Replacing code deployment systems like Capistrano with image-based deployment.
- Safely running legacy and new apps on the same server.
- Migrating to service oriented architecture over time with one toolchain.
- Managing horizontal scalability and elasticity in the cloud or on bare metal.
- Ensuring consistency across multiple environments, from development to staging to production.

- Simplifying developer machine setup and consistency.

Migrating an app's background workers to a Docker cluster while leaving the web servers and database servers alone is a common example of how to get started with Docker. Another example is migrating parts of an app's REST API to run in Docker with a Nginx proxy in front to route traffic between legacy and Docker clusters. Using techniques like these allows teams to seamlessly migrate from a monolithic to a service oriented architecture over time.

Today's applications often require dozens of third-party libraries to accelerate feature development or connect to third-party SaaS and database services. Each of these libraries introduces the possibility of bugs or dependency versioning hell. Then add in frequent library changes and it all creates substantial pressure to deploy working code consistently without the failure on infrastructure.

Docker's golden image mentality allows teams to deploy working code--either monolithic, service oriented, or hybrid---in a way that is testable, repeatable, documented, and consistent for every deployment due to bundling code and dependencies in the same image. Once an image is built, it can be deployed to any number of servers running the Docker daemon.

Another common Docker use case is deploying a single container across multiple environments, following a typical code path from development to staging to production. A container allows for a consistent, testable environment throughout this code path.

As a developer, the Docker model allows for debugging the exact same code in production on a developer laptop. A developer can easily download, run, and debug the problematic production image without needing to first modify the local development environment.

What to Expect

Running Docker containers in production is difficult but achievable. More and more companies are starting to run Docker in production everyday. As with all infrastructure, start small and migrate over time.

Why is Docker in production difficult?

A production environment will need bulletproof deployment, health checks, minimal or zero downtime, the ability to recover from failure (rollback), a way to centrally store logs, a way to profile or instrument the app, and a way to aggregate metrics for monitoring. Newer technologies like Docker are fun to use but will take time to perfect.

Docker is extremely useful for portability, consistency, and packaging services that require many dependencies. Most teams are forging ahead with Docker due to one or more pain points:

- Lots of different dependencies for different parts of an app.

- Support of legacy applications with old dependencies.
- Workflow issues between devs and DevOps.

Out of the teams we interviewed for this book, there was a common tale of caution around trying to adopt Docker in one fell swoop within an organization. Even if the ops team is fully ready to adopt Docker, keep in mind that transitioning to Docker often means pushing the burden of managing dependencies to developers. While many developers are begging for this self-reliance since it allows them to iterate faster, not every developer is capable or interested in adding this to their list of responsibilities. It takes time to migrate company culture to support a good Docker workflow.

In the next chapter we will go over the Docker stack.

2 The Stack

Every production Docker setup includes a few basic architectural components that are universal to running server clusters--both containerized and traditional. In many ways, it is easiest to initially think about building and running containers in the same way you are currently building and running virtual machines but with a new set of tools and techniques.

1. Build and snapshot an image.
2. Upload the image to repository.
3. Download the image to a host.
4. Run the image as a container.
5. Connect the container to other services.
6. Route traffic to the container.
7. Ship container logs somewhere.
8. Monitor the container.

Unlike VMs, containers provide more flexibility by separating hosts (bare metal or VM) from applications services. This allows for intuitive improvements in building and provisioning flows, but it comes with a bit of added overhead due to the additional nested layer of containers.

The typical Docker stack will include components to address each of the following concerns:

- Build system
- Image repository
- Host management
- Configuration management
- Deployment
- Orchestration
- Logging
- Monitoring

Build System

- How do images get built and pushed to the image repo?
- Where do Dockerfiles live?

There are two common ways to build Docker images:

1. Manually build on a developer laptop and push to a repo.
2. Automatically build with a CI/CD system upon a code push.

The ideal production Docker environments will use a CI/CD (Configuration Integration / Continuous Deployment) system like Jenkins or Codeship to automatically build images when code is pushed. Once the container is built, it is sent to an image repo where the automated test system can download and run it.

Image Repository

- Where are Docker images stored?

The current state of Docker image repos is less than reliable, but getting better every month. Docker's **hosted image repo hub** is notoriously unreliable, requiring additional retries and failsafe measures. Most teams will likely want to run their own image repo on their own infrastructure to minimize network transfer costs and latencies.

Host Management

- How are hosts provisioned?
- How are hosts upgraded?

Since Docker images contain the app and dependencies, host management systems typically just need to spin up new servers, configure access and firewalls, and install the Docker daemon.

Services like Amazon's **EC2 Container Service** eliminate the need for traditional host managment.

Configuration Management

- How do you define clusters of containers?
- How do you handle run time configuration for hosts and containers?
- How do you manage keys and secrets?

As a general rule, avoid traditional configuration management as much as possible. It is added complexity that often breaks. Use tools like **Ansible**, **SaltStack**, **Chef** or **Puppet** only

to provision hosts with the Docker daemon. Try to get rid of reliance on your old configuration management systems as much as possible and move toward self-configured containers using the discovery and clustering techniques in this book.

Deployment

- How do you get the container onto the host?

There are two basic methods of image deployment:

1. **Push** - deployment or orchestration system pushes an image to the relevant hosts.
2. **Pull** - image is pulled from image repo in advance or on demand.

Orchestration

- How do you organize containers into clusters?
- What servers do you run the containers on?
- How do you schedule server resources?
- How do you run containers?
- How do you route traffic to containers?
- How do you enable containers to expose and discover services?

Orchestration = duct tape. At least most of the time.

There are many early stage, full-featured container orchestration systems like **Docker Swarm**, **Kubernetes**, **Mesos**, and **Flynn**. These are often overkill for most teams due to the added complexity of debugging when something goes wrong in production. Deciding on what tools to use for orchestration is often the hardest part of getting up and running with Docker.

In the next chapter we cover a minimalistic approach to building Docker systems that Peerspace took.

3 Example – Bare Bones Environment

Container usage in production has been associated with large companies deploying thousands of containers on a similarly large number of hosts. You don't need to be building such large systems in order to levergage containers, in fact it is quite the contrary. It is smaller teams that can benefit the most from containers, making it so that building and deploying services is easy, repeatable and scalable.

This chapter describes a minimalistic approach to building systems that Peerspace, one of such smaller companies, took. This minimalistic approach allowed them to boostrap a new market in a short time and with limited resources, all the while keeping a high development velocity.

Peerspace set out to build their systems in a way that would be both easy to develop on and stable in production. These two goals are usually contradictory, since the large amounts of change that come with high development velocity in turn generate a great deal of change on how systems are built and configured. As most any experienced system administrator would agree, such a rate of change leads to instability.

Docker seemed a great fit from the start, given that it is developer friendly and it also favors agile approaches to building and operating systems. But even though Docker simplifies some aspects of development and systems configurations, it is at times over-simplistic. Striking the right balance between ease of development and robust operations is not trivial.

Keeping the Pieces Simple

Peerspace's approach to achieving the goals of developer velocity and stable production environments consists of embracing simplicity. In this case, simple means that each piece of the system--container--has one goal and one goal only. This goal is that the same processes, such as log collection, are done in the same way everywhere, and that the way in which pieces connect together is defined explicitly and statically--you can look at a configuration file.

Such a simple system makes it is easy for developers to build on different parts of the system concurrently and independently, knowing that the containers they're building will

fit together. Also, when problems appear in production, the same simplicity makes it quite straightforward to troubleshoot and resolve these issues.

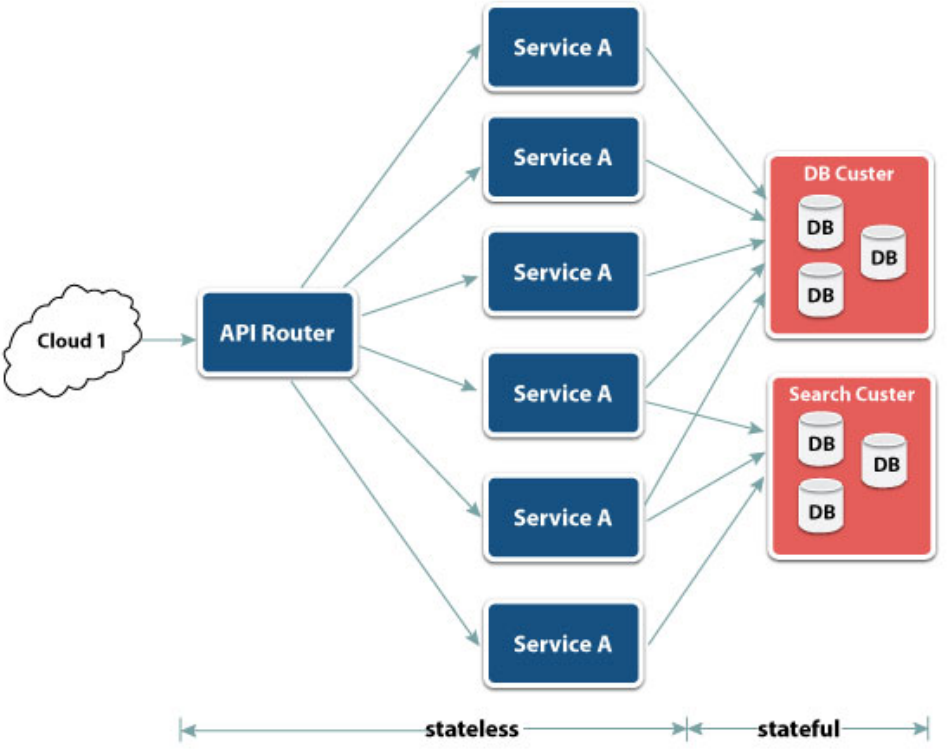
Keeping the system simple over time requires a great deal of thought, compromise, and tenacity, but in the end this simplicity pays off.

PeerSpace's system is comprised of 20 odd mircroservices, some of which are backed up by a MongoDB database and/or an ElasticSearch search engine. The system was designed with the following guidelines:

1. Favor stateless services. This is probably the biggest decision in simplifying PeerSpace's production environment: most of their services are stateless. Stateless services do not keep any information that should be persisted, except for temporary information that is needed to process the current ongoing requests. The advantage of stateless services is that they can be easily destroyed, restarted, replicated and scaled, all without regard of handling any data. Stateless services are also easier to write.
2. Favor static configuration. The configuration of all hosts and services is static: once a configuration is pushed to the servers, this configuration will remain in effect until a new one is explicitly pushed. This is in contraposition to systems that are dynamically configured, where the actual configuration of the system is generated in real time and can autonomously change based on factors such as available hosts and incoming load. Static configurations are easier to understand and troubleshoot, although dynamic systems scale better and can have interesting properties like the ability to heal in front of certain failures.
3. Favor network layout is also static: if a service is to be found in one host, it will always be found in that host until a new configuration is decided and committed.
4. Treat stateless and stateful services differently. Although most of PeerSpace's services are stateless, they use MongoDB and ElasticSearch to persist data. These two types of services are very different in nature and should be treated accordingly. For example, while you can easily move a stateless service from one host to another by just starting the new service and then stopping the old one, doing so with a database requires to also move the data. Moving this data can take a long time, require the service to be stopped while the migration is taking place, or device methods to perform an online migration. In our field it is common to refer to such stateless services as cattle--nameless, easy to replace and scale--and stateful services as pets--unique, named, need upkeep, and hard to scale. Fortunately, in the case of Peerspace, as in any farm, their number of cattle largely outnumber their pets.

These design principles above are the foundation of the simplicity of Peerspace's systems. Separating stateful from stateless services allows for the different treatment of services that are essentially very different, thus this treatment is optimized and as simple as possible in each case. Running stateless services with static configuration allows for the procedures required to operate the systems to be very straightforward: most of the times

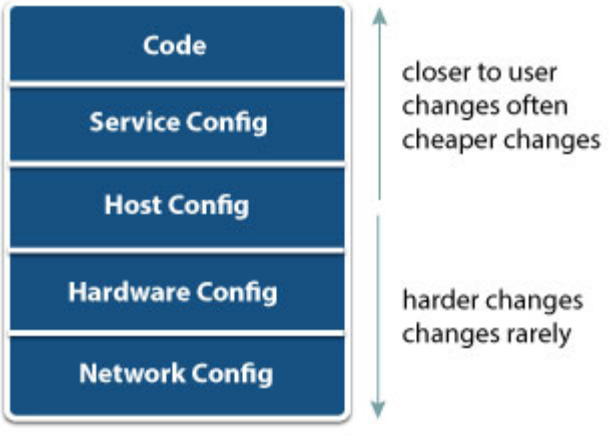
they are reduced to copying files and restarting containers, with no regard of other considerations like dependencies on third-party systems.



The proof of whether these design guidelines lead to a simplified system depends on whether or not operating the system is equally simple.

Keeping The Processes Simple

When designing their operational processes, PeerSpace used the assumption, based upon observation, that the layers of their infrastructure closer to the hardware are the ones that change less often, whereas the ones closer to the end user are the ones that change most often.



According to this observation, the number of servers used in a production environment rarely change, usually due to either scaling issues or hardware failure. These server's configuration might change more often, usually for reasons related to performance patches, OS bug fixes, or security issues.

The number and kind of services that run on the above servers changes more often. This usually means moving services around, adding new kinds of services, or running operations on data. Other changes at this level can be related to newer versions deployed that require reconfiguration, or changes in third-party services. These changes are still not very common.

Most of the changes that take place in such infrastructure are related to pushing new versions of the many services. On any given day PeerSpace can perform many deployments of newer versions of their services. Most often pushing one of these new versions is simply replacing the current with new ones running a newer image. Sometimes, the same image is used but the configuration parameters change.

PeerSpace processes are built to make the most frequent changes the easiest and simplest to perform, even if this might have made infrastructure changes harder (it hasn't).

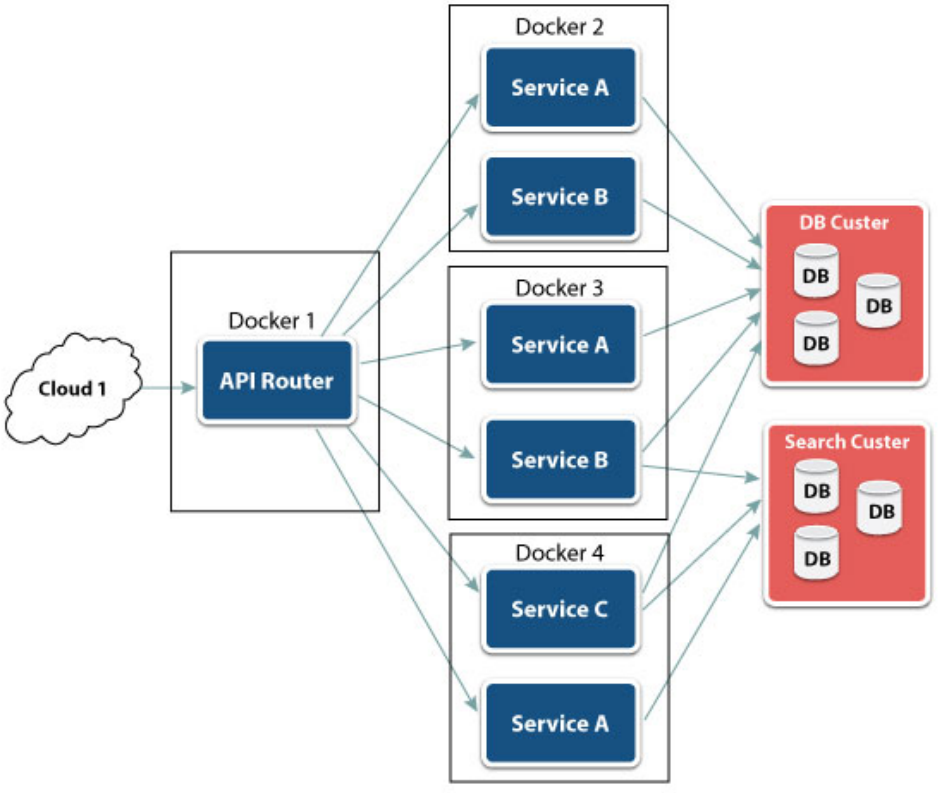
Systems in Detail

PeerSpace runs three production-like clusters: integration, staging and production. Each cluster contains the same amount of services and they are all configured in the same way, except for their raw capacity (CPU, RAM, etc.). Developers also run full or partial clusters on their computers.

Each cluster is composed by:

1. A number of docker hosts running CentOS 7, using **systemd** as the system supervisor.
2. A MongoDB server or replica set.
3. An ElasticSearch server or cluster.

The MongoDB and/or the ElasticSearch servers might be dockerized on some environments and not dockerized on others. They are also shared by multiple environments. In production, and for operational and performance reasons, these data services are not dockerized.

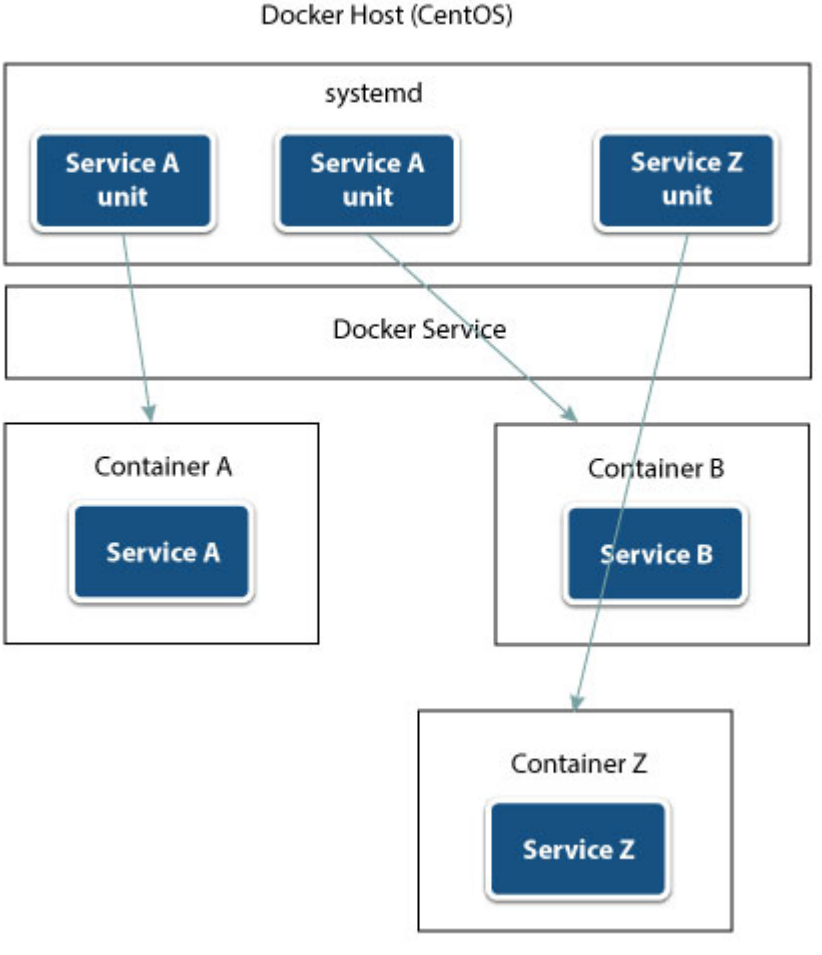


Each docker hosts runs a static set of services, and each of these services is built following the same pattern:

- All of their configuration is set via environment variables. This includes the addresses (and ports) of other services.
- They don't write any data to disk.
- They send their logs to stdout.
- Their lifecycle is managed by systemd and defined in a systemd unit file.

Leveraging systemd

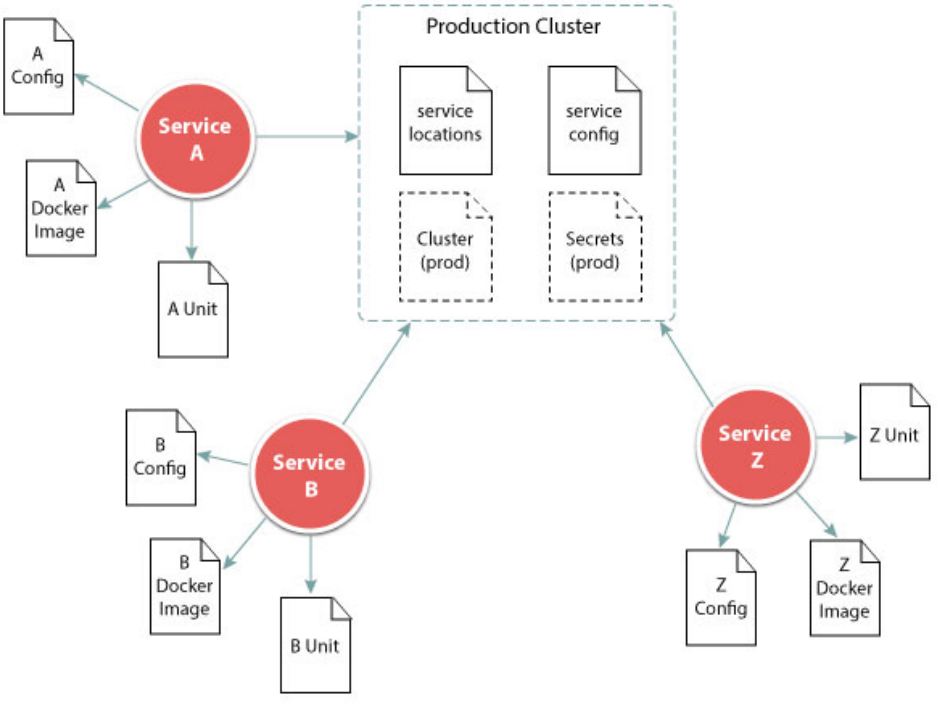
Each service is managed by systemd. Systemd is a service supervisor loosely based on OSX's launchd and, among other things, uses plain data files named units to define each service's lifecycle--as opposed to other more traditional supervisors that use shell scripts for such matters.



PeerSpace's services only have the Docker process as their sole runtime dependency. Systemd's dependency management is only used to ensure that Docker is running, but not to ensure that their own services are started in the right order. The services are built in such a way that they can be started in any order.

Each of the services is composed by:

1. A container image.
2. A systemd unit file.
3. An environment variable file specific for this container.
4. A set of shared environment variable files for global configuration parameters.



All units follow the same structure. Before the service starts, there are a set of files that are loaded for their environment variables:

```
EnvironmentFile=/usr/etc/service-locations.env
EnvironmentFile=/usr/etc/service-config.env
EnvironmentFile=/usr/etc/cluster.env
EnvironmentFile=/usr/etc/secrets.env
EnvironmentFile=/usr/etc/%n.env
```

This ensures that each service loads a set of common environment files (`service-locations.env`, `service-config.env`, `cluster.env` and `secrets.env`) plus one that is specific to this particular service: `%n.env`, where at runtime `%n` is replaced by the full name of the unit. For example, `docker-search.service` is replaced for a service unit named `docker-search`.

Next are entries to ensure the container is properly removed before we start a new one:

```
ExecStartPre=-/bin/docker kill %n
ExecStartPre=-/bin/docker rm -f %n
```

Containers are named after the unit's full name, using `%n`. Naming the containers after a variable makes the unit file a bit more generic and portable. Pre-pending the path to the docker binary with `-` prevents the unit from failing to start if the command fails; you need to ignore potential failures because these commands will fail if there is no pre-exisiting container, which is a legal situation.

The main entry in the unit is `ExecStart`, where it instructs systemd how to start the container. There is quite a bit going on here, but let's highlight the most important bits:

```
ExecStart=/bin/docker \
          run \
          -p "${APP_PORT}:${APP_PORT}" \
          -e "APP_PORT=${APP_PORT}"  \
          -e "SERVICE_C_HOST=${SERVICE_C_HOST}" \
          -e "SERVICE_D_HOST=${SERIVCE_D_HOST}" \
          -e "SERVICE_M_HOST=${SERVICE_M_HOST}" \
          --add-host docker01:${DOCKER01_IP} \
          --add-host docker02:${DOCKER02_IP} \
          --volume /usr/local/docker-data/%n/db:/data/data \
          --volume /usr/local/docker-data/%n/logs:/data/logs \
          --name %n \
          ${IMAGE_NAME}:${IMAGE_TAG}
```

1. Use the environment variables loaded by `EnvironmentFile` to configure the container (succh as the ports exported with `-p`).
2. Add the address of the other hosts in the cluster in the container's `/etc/hosts` (`--add-host`).
3. Map some volumes for logs and data. This is mostly a honey pot, so you can check those directories to ensure no one is writing on them.
4. The image itself (name and version) comes from the environment variables loaded by `/usr/etc/%n.env`, so in this example case it would map to `/usr/etc/docker-search.service.env`.

To finalize, here are some entries to define how to stop the container and other lifecycle concerns:

```
ExecStop=-/bin/docker stop %n
Restart=on-failure
RestartSec=1s
TimeoutStartSec=120
TimeoutStopSec=30
```

Cluster-wide, common and local configurations

PeerSpace breaks out the configuration of their clusters into two types of files: environment variable files and systemd units. We've already talked about the units and how they load the environment variable files, so let's see what's in the environment files.

The environment variables are broken into different files, mostly because of how these files are to be changed, or not, across clusters, but also for other operational reasons:

- `service-locations.env`: the host names of all services in the cluster. This is often the same across clusters, but doesn't have to be.
- `service-config.env`: config related to the services themselves. This *should* be the same across clusters if they're running the compatible versions of the services.
- `secrets.env`: the secret keys. This file is handled differently than the others because of its content, and is different on each cluster.
- `cluster.env`: everything that is different across clusters goes here, such as which db-prefix to use, whether test or production, external address, and more. The most relevant information in this file is the ip addresses of all of the hosts that belong to the cluster.

The following are all the files in some example cluster.This is the `cluster.env` file:

```
CLUSTER_ID=alpha
CLUSTER_TYPE="test"
DOCKER01_IP=x.x.x.226
DOCKER02_IP=x.x.x.144
EXTERNAL_ADDRESS=https://somethingorother.com
LOG_STORE_HOST=x.x.x.201
LOG_STORE_PORT=9200
MONGODB_PREFIX=alpha
MONGODB_HOST_01=x.x.x.177
MONGODB_HOST_02=x.x.x.299
MONGODB_REPLICA_SET_ID=rs001
```

And this is the `service-locations.env`:

```
SERVICE_A_HOST=docker01
SERVICE_B_HOST=docker03
CLIENTLOG_HOST=docker02
SERIVCE_D_HOST=docker01
...
SERVICE_Y_HOST=docker03
SERVICE_Z_HOST=docker01
```

Each systemd unit contains references to the other hosts in the cluster, and these references come from environment variables. These variables containing service host names are threaded into the docker command to make them available to the container process.

This is done using `-e`, for example: `-e "SERVICE_D_HOST=${SERIVCE_D_HOST}"`.

IP address of the docker hosts are also injected into the container by `--add-host docker01:${DOCKER01_IP}`. This allows you to spread the containers across different number of hosts by only changing those two files and keeping the units intact.

Deploying services

Changes done at the container level or at configuration level are done in three steps: first make the changes on our config repository (git), second copy the config files to a staging area on the hosts (ssh), and third make the configuration changes effective by running a script on the host that deploys each service individually. This approach provides versionable configuration, pushing a coherent configuration at once, and a flexible way to make this pushed configuration effective.

When you need to make changes on a set of services, first make the changes on git and commit them. Then, run a script that pushes this configuration to a staging area within each of the hosts. Once this configuration is pushed, run a script on each of the hosts to deploy or re-deploy a subset of all of the containers on the host. This script does the following for each of the listed service:

1. Copies the config files from the staging are into their final place:
    - systemd unit
    - shared config files
    - this service's config file
    - secrets (decrypted)
2. Downloads the image if necessary (the image is defined in the service's own config file).
3. Reloads systemd's configuration so that the new unit is picked up.
4. Restarts the systemd unit corresponding to the container.

PeerSpace has two deployment workflows, and understanding them might help clarify their deploy processes: one for development and the other to push to production, with the latter being a superset of the former.

During development, they deploy ad-hoc builds to their integration servers by:

1. Creating a new container image with the latest codebase.
2. Pushing this image to the image repository.
3. Run the deploy script on the host running the container for this image.

Development systemd units are tracking the latest version of the image, so as long as the configuration does not change, it is sufficient to push the image and redeploy.

Production-like servers (prod and staging) are similar to the development ones configuration-wise, but the one main difference is that container images in production are all tagged with a version number instead of `latest`. Here is the process of deploying a released image to a production-like container:

1. Run the release script on the repo for the container image. This script will tag the git repo with the new version number and build and push the image with this version number.
2. Update the per-service environment variable file to refer to this new image tag.
3. Push the new configuration for the host(s).
4. Run the deploy script on the host(s) running the container for this image.

When services move from development to production they usually do it in batches (usually every two weeks). When pushing a release to production, the config files used in development for that release are copied to the production directory. Most of these files are copied verbatim, as they are abstracted from the concrete aspects of the cluster (ips, number of hosts, and more), but `cluster.env` and `secrets.env` are usually different on each cluster and these are not updated while releasing. Usually, all newer versions of the services are pushed at once.

Support services

PeerSpace uses a set of services to support their own services. These services include:

- log aggregation: a combination of fluentd + kibana, and **docker-gen** . Docker-gen allows creating and recreating a configuration file based on the containers running on the host. Docker-gen generates a fluentd entry for each running container that sends the logs to kibana. This works well and is easy to debug.
- Monitoring: Datadog, a SaaS monitoring service. The datadog agent is running a container, and it is used to monitor performance metrics as well as API usage and business events. Datadog provides extensive support to tags, allowing you to tag every single event in multiple ways, and this is done by fluentd. This extensive tagging allows you to slice and dice the data in multiple ways after the data is collected (such as same service across clusters, all docker services, all API endpoints using a certain release, and more).

Discussion

This system results in a very explicit configuration of all of the hosts and services, which allow all developers to easily understand the system's configuration and also to be able work on different parts of the system without interference. Each developer can push to an integration cluster at any time, and to production with minimal coordination.

Since the configuration of each cluster is kept on git, it is easy to track changes in configuration and troubleshoot a cluster when there are configuration issues.

Because of the way the configuration is pushed, once a new configuration is set in place, this configuration does not change. This static configuration provides you with a great deal of stability.

Also, the way the services are written--configured by environment variables, logging to console, stateless, and more--makes them very amenable to be used later as-is by cluster managers like Mesos or Kubernetes.

Of course, these tradeoffs come at a price. One of the most obvious downsides is that configuration is somewhat tedious, repetitive and error prone. There is a great deal of automation that we could implement to generate these configuration files.

Changes on global configuration might require restarting more than one container. Currently it is up to the developer to restart the right containers. In production, you can usually do a rolling restart when pushing many changes, which is not ideal. This is definitely a weak point, but so far it is manageable.

Future

There are a few extensions to this system that are being considered. One of them is enabling zero downtime deployments using a reverse proxy. This would also allow Peerspace to scale horizontally each of the services.

Another direction is to generate all of the configuration files from a higher level description of the cluster. This option has the potential of computing which containers need to be restarted after configuration changes.

When considering these future directions, Peerspace is also weighing the possibility of using Mesos or Kubernetes, as they argue that adding any more complexity to their deployment scripts would be stretching this simplistic model a bit too much.

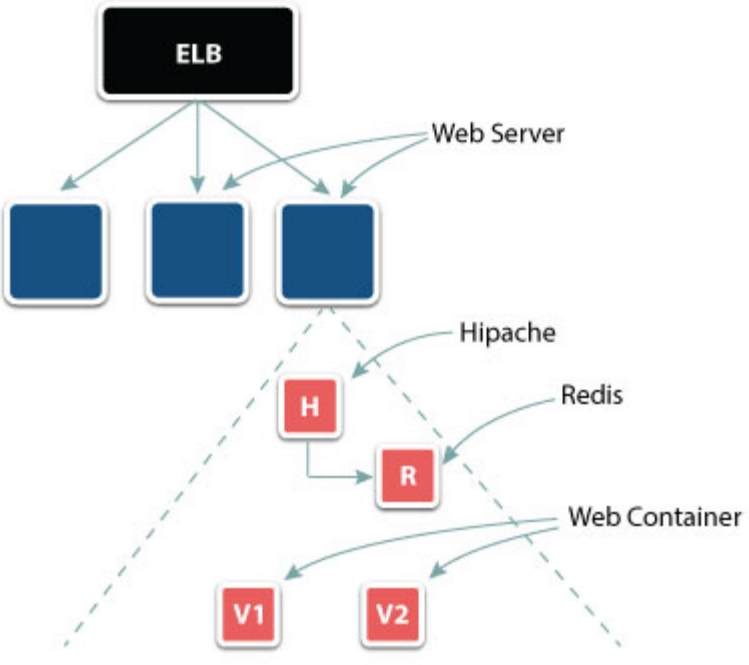
Summary

Although this chapter has covered a radically minimalistic approach to Docker, we hope it provides the foundation to “thinking in Docker,” an ability that we think will pay off as your read the rest of the book, regardless of whether you decide to try a bare bones approach or you decide to try your luck with a cluster management system.

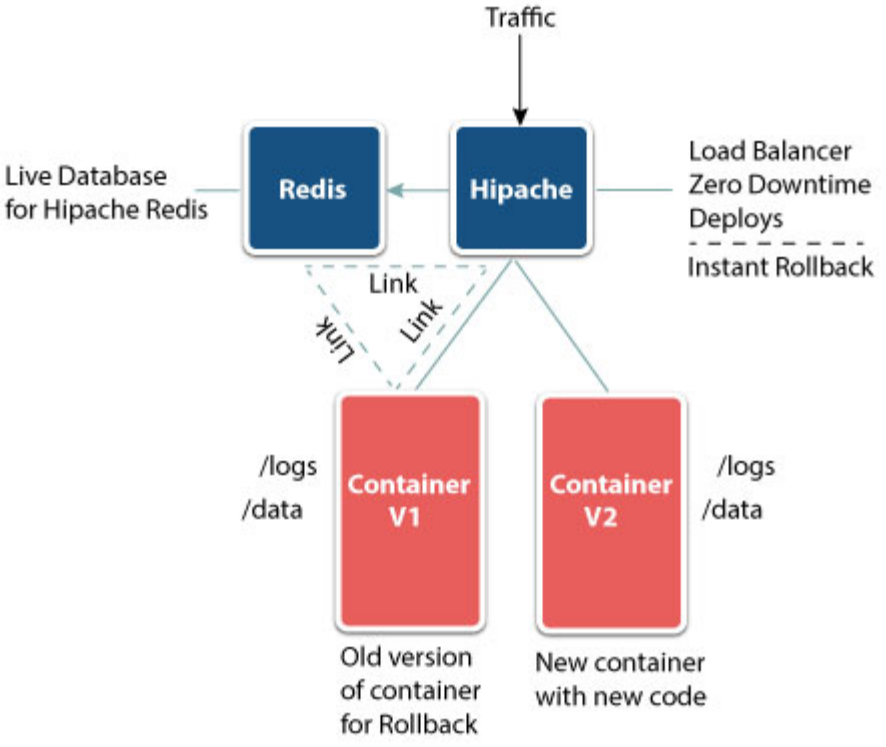
Of course there are many other ways to approach Docker, and the next chapter walks through a real live production web server environment that has been running at RelateIQ for over a year now with Docker.

4 Example – Web Environment

Most companies we've seen have been successful using Docker by running a low container to host ratio of 1-2 containers to a single host machine. In other words, you don't need to run Apache Mesos or Kubernetes to be successful running Docker in production. In this example we'll walk through a real live production web server environment that has been running at RelateIQ for over a year now with Docker. For example, this environment uses Docker on standard Amazon Web Services instances running Ubuntu to power their production CRM web application. The reason Docker was used initially was to provide zero downtime deployments for their customers due to the ability to spin up and down containers quickly, provide dependency isolation between web versions, and use instant rollbacks. Here is a high level image of the environment.



Believe it or not this web environment has the following: stable and zero downtime deploys, rollbacks, centralized logging, monitoring, and a way to profile a JVM. All of this is powered by orchestrating Docker images through bash scripts. Let's look closer at a single machine.



The web server is running on a single AWS server with four containers running on Docker. Some of the containers are linked to provide communication to other containers on the Docker bridge. It has multiple ports exposed to the host in order to provide HTTP and JVM monitoring for profiling. It uses an Amazon ELB load balancer (where it does health checks). All of the containers store their logs to the host so existing logging solutions still work (SumoLogic), and there is a simple bash orchestration script to deploy and setup new versions of the web service.

Let's look into some of the specifics to help understand some of the main areas that many companies have questions on when running Docker in production.

Orchestration

When you break down orchestration there are essentially two things happening. One is how you get the server installed with Docker and ready to run the containers, and the other is how to get the container up and running on the server.

Getting Docker on the server ready to run containers

The server is deployed on AWS using a standard base Ubuntu AMI. The host is setup using a standard configuration management system in **Chef**. The setup is very traditional to many environments today. After the server is launched Chef will run and setup the ssh users, ssh keys, then install basic packages with Chef's package installer (such as iostat), install and configure the monitoring agent (Datadog in this scenario), raid some ephemeral drives for data or log storage, install and configure the logging agent (SumoLogic), install the latest version of Docker, and lastly create the bash setup script and configure it to run in a cron job.

After Chef runs on the server the host is now setup to run any containers that are needed on the machine. It's also configured with monitoring and logging software for future debugging. This environment can run any type of container service, and is not different than most server environments run today even in physical environments. Now that Docker is installed and the host is ready with the core operations tools, let's move on to getting the containers on the host to run the web application.

Getting the containers running

Most companies that started running Docker early on typically used bash scripts to setup the containers. This environment is no different by using a cron job setup to run a bash script every five minutes in order to do all the orchestration of the containers. The core function of the script is programed to setup the correct containers and pull down the latest web server image. Lets dive deeper into snippits of the script that they use.

This script does the following:

1. Check if the containers are running (typically they are, this is just in case it's a new machine).
2. If they are not running, then deploy hipache and redis containers and link them together.
3. Pull the latest web server container.
4. Wait for the web server to health check before adding it to the load balancer.
5. Once successful, send a message to the mini load balancer hipache on the server (in this case a redis-cli command using netcat) with the random port it received from docker and the ip address.
6. Keep the old container running so that it's possible to rollback if needed.
7. Clean up old images.

Here are snippits from their script (some lines removed for readability):

```
#!/bin/bash

# check for hipache container
```

```
STATE=$(docker inspect hipache | jq ".[0].State.Running")
if [[ "$STATE" != "true" ]]; then
    set +e
    docker rm hipache > /dev/null 2>&1
    set -e
    mkdir -p /logs/hipache/
    docker run -p 80:80 -p 6379:6379 --name hipache -v /logs/
hipache:/logs -d repo.com/hipache
    echo "$(date +"%Y-%m-%d %H:%M:%S %Z") lpush frontend:* de-
fault"
    sleep 5
    (echo -en "lpush frontend:* default\r\n"; sleep 1) | nc lo-
calhost 6379
fi

#pull the latest image
IMAGE_ID=$(docker images | grep ${IMAGE_NAME} | grep $RE-
MOTE_VERSION | head -n 1 | awk '{print $3}')
if [ -z $IMAGE_ID ]; then
  docker pull $DOCKER_IMAGE_NAME
fi

echo $REMOTE_VERSION > $VERSION_FILE

#launch a new one
echo "$(date +"%Y-%m-%d %H:%M:%S %Z") launching $DOCKER_IM-
AGE_NAME, logging to $LOG_DIR"
mkdir -p $LOG_DIR
NEW_WEBAPP_ID="abcdefghijklmnopqrstuvwxyz"
MAX_TIMEOUT=5
set +e
until [ $MAX_TIMEOUT -le 0 ] || NEW_WEBAPP_ID=$(docker run -P -
h $(hostname) --link hipache:hipache $(dockerParameters
$BRANCH) -d -v $LOG_DIR:/logs $DOCKER_IMAGE_NAME); do
    echo -n "."
    sleep 1
    let MAX_TIMEOUT-=1
done
set -e

#check to see if web app container started
NEW_WEBAPP_IP_ADDR=$(docker inspect $NEW_WEBAPP_ID | jq '.
[0].NetworkSettings.IPAddress' -r)
if [ -z "$NEW_WEBAPP_IP_ADDR" -o "$NEW_WEBAPP_IP_ADDR" =
"null" ]; then
    echo "$(date +"%Y-%m-%d %H:%M:%S %Z") no new webapp ip,
failed to start"
    # send_deploy_message $HOSTNAME $BRANCH $IMAGE_NAME "error"
    send_webhook $HOSTNAME $BRANCH $BUILD_ID $BUILD_NUMBER
"failure"
```

```
    exit 1
fi

echo -n "$(date +"%Y-%m-%d %H:%M:%S %Z") new instance $NEW_WE-
BAPP_ID starting, on ip $NEW_WEBAPP_IP_ADDR"
# 5 minutes
MAX_TIMEOUT=300
HEALTH_RC=1
set +e
until [ $HEALTH_RC == 0 ]; do
    if [ $MAX_TIMEOUT -le 0 ]; then
        echo "$(date +"%Y-%m-%d %H:%M:%S %Z") failed to be
healthy within 5 minutes, killing and exiting..."
        docker kill $NEW_WEBAPP_ID
        docker rm $NEW_WEBAPP_ID
        # send_deploy_message $HOSTNAME $BRANCH $IMAGE_NAME "er-
ror"
        send_webhook $HOSTNAME $BRANCH $BUILD_ID $BUILD_NUMBER
"failure"
        exit 1
    fi

    ${SCRIPT_HOME}/health.sh $NEW_WEBAPP_IP_ADDR
    HEALTH_RC=$?
    echo -n "."
    sleep 5
    let MAX_TIMEOUT-=5
done
set -e
echo

# add myself as a backend to redis
(echo -en "rpush frontend:* http://${NEW_WEBAPP_IP_ADDR}:${WE-
BAPP_PORT}\r\n"; sleep 1) | nc localhost 6379
# ensure i am first backend to redis
(echo -en "lset frontend:* 1 http://${NEW_WEBAPP_IP_ADDR}:${WE-
BAPP_PORT}\r\n"; sleep 1) | nc localhost 6379
# remove all but 1 backend to redis
(echo -en "ltrim frontend:* 0 1\r\n"; sleep 1) | nc localhost
6379
```

As you can see, most of the script does some very basic bash stuff. With some bash scripting experience any system administrator or operations engineer can do the same type of orchestration. Orchestration for containers can be simple, but it does require some iteration and over time the scripts will become more robust. In a failure scenario the script is also setup appropriately so it doesn't bring the new container online if the container doesn't pass the health checks. As new technology around Docker comes out, such as sys-

tems like Apache Mesos and Kubernetes, it should replace the need for bash scripts to do orchestration. Lets jump into some other areas on how this environment works.

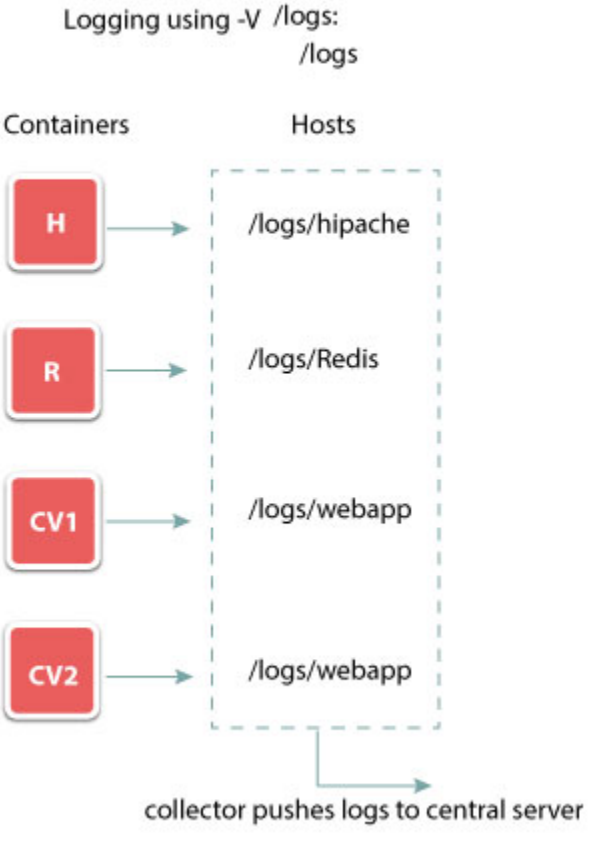
Networking

The networking for a single host running Docker run and a container is easy enough once you get the hang of it. Docker exposes ports in the container to the host through the Docker run command. The ports exposed on the server are port 80 (ssl is terminated on the load balancer) that the load balancer listens on, a profiling port for Java profiling, a port for redis to switch the load balancer backend, and a port for the web server itself (deep dive in later chapters). The load balancer siting outside of the server only monitors port 80. On the host itself the web server launches with a random port that the hipache proxy will forward requests onto from port 80.

Data storage

Since this is a web service, there typically isn't a lot you need for storage. Sometimes you will need to store logs and maybe some type of file cache or static content to load from. For this case we use the hosts storage and not the container. The reason why we store the data on the host is simple. If the container dies we still want to be able to troubleshoot what happened. Typically services log to a file path. In this case we have the Docker container map to the hosts file system and redirect the persistent logs from within the container to the host for future log analysis. This is easily done with the volume -v command in Docker run.

Logging



Container logs are systematic based on the service. For instance, we use /logs/redis, /logs/hipache, and /logs/webserver/. An important note here is the web server logs errors and request logs based on the date time stamp of the requests. When a container logs it will look like this: /logs/webserver/2015-03-01.request.log. It will auto append to the same file if it exists. If another container or even more come up it will automatically log to the same file due to the append. Log rotate is installed with Chef to keep the logs from growing out of control.

In a production environment you will typically have a centralized log server so the logs on the server are only temporary until they are picked up from the collector. Since each of the containers log to the host there is no need to use a whole new logging technology for

Docker. An environment not running Docker will most likely log the same exact way allowing for operations to keep their existing monitoring framework in place. In this environment the logs are easily shipped to the central log server (Splunk, Sumologic, and Loggly) as new log files get created or appended to for analysis.

Monitoring

The important note here is that the load balancer is monitoring for the uptime of the server and will automatically send the next request to another available web service if needed. The host is monitored via the monitoring agent on the host with a docker plug-in (Datadog in this case). The monitoring in this example is a full stack monitor. This agent monitors for host usage like CPU, memory, disk IO, JVM monitoring, and number of running containers. The application metrics in this environment are sent via StatsD to a central collector. Metrics like web hits, application query speed, and function specific latency metrics.

This environment uses a JVM profiling tool called Yourkit to monitor what's going on in the heap. The usage of this allows for the ops team or developers to connect to the host with their profiling tool to figure out deep issues with the call stack of the application. One downside is you need to have a single port per container and it can't be the same if two containers are running on the host at the same time. So it requires a quick ssh or tools to check this port. Newer technologies are able to monitor this such as New Relic and Sysdig (mentioned in the ecosystem).

No worries about new dependencies

Since all of the application dependencies are stored within the container image, the operations team now only has to manage the dependencies for their server management aspects. This simplifies the Chef configuration management framework and amount of scripts used to keep the environment up to date.

Zero downtime

This web service environment can provide zero downtime deployment. The zero downtime deployment works using hipache with a realtime web query engine backed by redis, which is a perfect database to use since it's single threaded. Hipache will redirect the HTTP sessions to the top most server in the database list. When a new container comes online and a command is sent to the redis server to update the list, the new container receives all the new hits. The session state is stored in a backend database so the containers can stay ephemeral and clients don't lose state.

Service rollbacks

Leveraging dockers image store on the server and its speed of container start time this environment can easily roll back old code if necessary. Since the machine has multiple containers stored on the host it's easy to spin up the old container and swap out the new (bad deploy) container with a similar script or another orchestration.

Conclusion

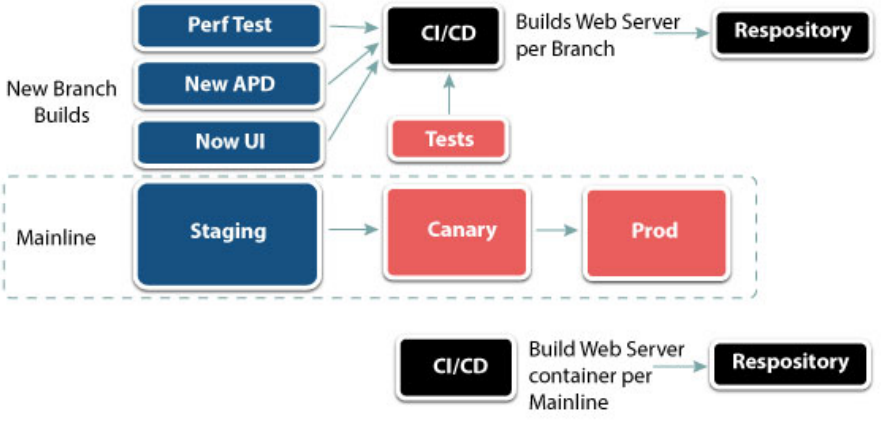
RelateIQ has been running the setup described in this chapter for over a year now in production with great success. The team took standard operations tools and applied it to Docker to create a very feature full web orchestration layer. This has allowed the team to experiment with newer technology without introducing major underlining foundational changes. They were also able to combine Docker with their current infrastructure monitoring and logging solutions, making it easy to run in production. If you would like to read more about this environment they have **blogged** and discussed their environment in several **talks**.

In the next chapter we will look at how RelateIQ is using a web environment per branch with Docker fully orchestrated using AWS Beanstalk.

Example – Beanstalk Environment 5

In most software companies today there are several infrastructure environments. These are typically a three-tier structure with a test, staging, and production environment. Some companies have pre-prod or even canary environments in addition to these, but those are special cases. These different environments provide isolation for the life cycle of new code or even infrastructure components. The environments are typically made up of at least a web server tier for application logic and presentation and a database tier. Over the last 10 years or so these environments have been fairly static within companies. We found another great Docker environment example from RelateIQ. They are doing something interesting that could break up the standard environment model.

RelateIQ has a web environment per branch fully orchestrated using AWS Beanstalk. This is a new type of infrastructure environment using Docker technology through their CI/CD infrastructure. They have essentially disconnected the web tier from the data tier. This turns the typical three tier model into only data stores. This might be a little hard to imagine at first, so let's see it visually.



This image shows three different branches with a PerfTest, a New APD, and a New UI branch. Each branch gets its own purpose-built web container through the CI/CD system. After its built and tested it will then be pushed to the repository. This allows a web environment per branch for every developer or team that needs one. It also allows you to think differently on how to use containers and rethink your environments all together. One of the huge advantages of using this model in a SaaS company is the power to see changes quicker. Think about how product managers and designers can leverage these types of environments if they are used in a continuous delivery model.

As an example, RelateIQ in the summer of 2014 completely redesigned their entire web application from CSS to a new look and feel using this new model with Docker. They were able to run A/B testing with side-by-side web servers to compare the old version to the new version. As developers would commit code within 15-30 minutes, the designers and product managers were able to see the newly reflected changes in an isolated environment. The web server also had a way to redirect the backend from the staging data servers to production data servers using an environment variable. With a quick restart of the Docker container they were able to see their newly reflected changes against production data in a matter of minutes. When RelateIQ was about to launch their newly designed site they switched the containers to the production data instead of staging. This allowed the developers to make sure the data and new UI matched up properly.

Process to build containers

In this environment, RelateIQ uses Teamcity to build and deploy their applications. They were able to use VCS triggers to monitor for specific branch names off of their github repository to perform automatic builds. For instance, they used docker-<branch>. If a branch was created in their repository starting with “docker-” they would automatically build this branch’s own Docker container. To get started quickly, most developers create their branch right off of staging. Once they create their new branch, Teamcity would build the container and push it to a local repository. They use the Beanstalk service from Amazon Web Services to deploy and update their containers.

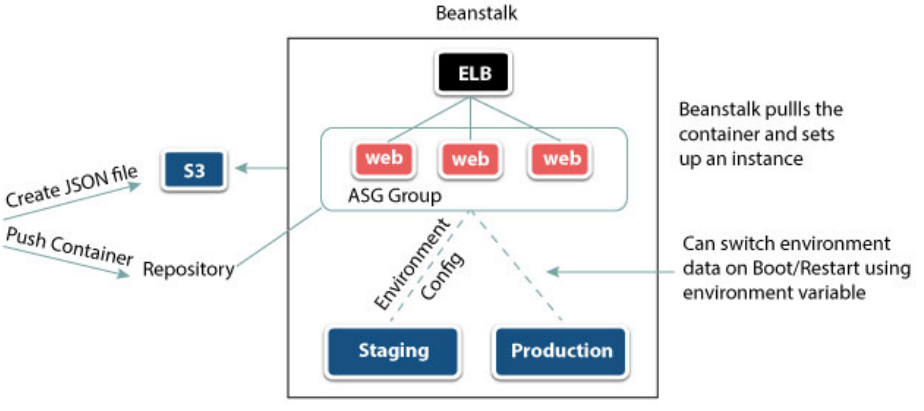
Process to deploy/update containers

“AWS Elastic Beanstalk is an easy-to-use service for deploying and scaling web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS.

You can simply upload your code and Elastic Beanstalk automatically handles the deployment, from capacity provisioning, load balancing, and auto-scaling, to application health monitoring. At the same time, you retain full control over the AWS resources powering your application and can access the underlying resources at any time.” **Sourced from Amazon**

Beanstalk will automatically deploy a Load balancer, setup auto scaling groups, provision the number of instances/servers based on the setting, pull down and run the Docker containers, provide health monitoring, and secure the servers with security groups. This gets a company really far with infrastructure if they are just starting out. When combining this with a container per web service it becomes a really useful environment for a SaaS company.

The deployment can be performed in a couple of ways, from Elastic Beanstalk to using an S3 bucket JSON file or an API call to the service itself. In this example they were using the S3 bucket JSON file created within their Teamcity configuration. A build step would write a new file and push the changes to the S3 bucket. The file contains where the container is, ports that need to be open, and the name of the container. When the new file is uploaded, the Elastic Beanstalk environment will automatically spin up another server, pull the container and set it up on the machine, then tear down the old container and server when its health checks pass. This essentially creates a zero downtime deployment to the new service.



Using Elastic Beanstalk shows that its only a matter of time before infrastructure providers really start to own running containers just like they did running virtual infrastructure.

Logging

Logging from the Elastic Beanstalk containers is fully automated, just like the rest of the infrastructure. You are able to pull logs through their GUI tool with the options of the full logs on the server or just the last 100 lines from Standard Output. These two options are only useful for troubleshooting purposes so they also offer a new option to tail the logs to an S3 bucket. Using a centralized logging service that can consume logs from S3 services

allows a company to integrate these logs directly into their current logging solution. There are a couple of notes around their logs. The names of the logs are not based on the container name it self. The name is created from a random generated service name that only pertains as a unique ID to the service. It can be troublesome to track down which unique ID belongs to which Elastic Beanstalk service.

Monitoring

All services that are provided through Amazon Web Services typically have their own cloud monitoring solution built in. Elastic Beanstalk is also supported. The monitoring you get, however, is very basic. You'll get the ELB metrics and server metrics but nothing from the container itself. This is due to the fact that the Elastic Beanstalk service has a 1:1 container per server ratio. This means network, CPU, disk, and memory metrics are typically the metrics from the container running on the host. If something goes rouge on the machine, though, you'll have to SSH into the service to troubleshoot or deploy a new version.

Security

Beanstalk provides automated firewall port security with their security groups along with IAM roles to secure user access. Since Beanstalk uses a low container to host ratio, just like a normal application and server environment, it is easy to isolate containers from other containers. Beanstalk templates also provide the ability to deploy new environments consistently across new deployments making security easy to change across a wide number of hosts.

Summary

By using Docker and an automated infrastructure with Elastic Beanstalk from Amazon Web Services, RelateIQ was able to provide a web environment for each of their front end engineers in a scalable template environment. The environments are super easy to create, and with a little orchestration from a CI/CD system, its completely automated. One note: if you're thinking of trying this in your own infrastructure, this environment comes with a lot of moving parts. The logging could be better since unique ID's from Beanstalk environments can be a pain, and at the time of writing you can only have a single container running on the service (soon multiple containers will be supported). Just keep in mind that when using Docker you can make your environment extremely flexible and provide new innovative ways to drive faster development.

In the next chapter we will dive into the topic of security with Docker.

6 Security

Security is always a difficult area. Striking the correct balance between paranoia and getting things into production, and getting a development team to think about security are the hallmarks of a good security professional.

Docker has always had a difficult relationship with security, as the model of doing usability first and then tagging security on later was always going to lead to criticism. In addition it does not have a single strong security model and response, but instead it relies on many layers, some of which may not be there yet, or may not work for your application. The **official security documentation** is also very minimal, so users need more support.

This reflects the Linux security approach, given how there are lots of options at the bazaar. These options are not all container specific, but containers have added more tools, leaving an even more complex security environment than most non containerized environments face.

Threat models

The security assessment of containers, or a microservice architecture in general, requires understanding threat models. These models depend on the individual circumstances, but many have a lot in common.

If you are going to run untrusted code, as a service provider might, or run a platform as a service, where people may deliberately upload hostile code, then your requirements are different from a small team developing and hosting an application. A large enterprise is in a different position, where regulatory requirements mean that certain forms of isolation are mandated.

If you are a service provider, currently virtualization is the most mature technology for isolating hostile code. This is not to say there have been no security issues, but the number is low and the attack surface smaller. That does not mean that containers are not also part of the solution; in particular if you are providing a language runtime, such as `ruby` or `Node.js` then many of the techniques below are entirely applicable. Containers are an extended form of isolation and can be used to reduce attack surface further, as a layered protection.

Google, in the Borg **paper** explains how they have a very different architecture for trusted internal jobs where "We use a Linux chroot jail as the primary security isolation mechanism between multiple tasks on the same machine," a weaker form of security than containers, versus: "VMs and security sandboxing techniques are used to run external software by Google's AppEngine (GAE) and Google Compute Engine (GCE). We run each hosted VM in a KVM process that runs as a Borg task."

It is the large and small companies looking at the security of a microservice architecture built around containers that will concern us most in this chapter.

Security is a complex area, and needs evaluating in the context of the applications you are running. There are no magic bullets, and defense in depth is key, which is why this chapter will cover many different ways to make your applications more secure. Many of them may eventually be baked into the tools you use, but it is still important to understand what they will and will not protect against, and whether a more specific solution can be used instead of a generic one.

Containers and security

Linux containers are not a monolithic entity, unlike for example FreeBSD's **jail**, which has a single system call to create and configure a container. Rather they are a set of facilities that further increase process isolation over and above the traditional Unix mechanisms of user ids and permissions.

Containers are essentially built from namespaces, cgroups and capabilities.

The core of containers in Linux is a series of "namespaces", modelled on ideas from the Plan 9 operating system. A process namespace hides all of the processes outside the namespace, giving you a new set of process IDs including a new init (pid 1). A network namespace hides the system's network interfaces and replaces them with a new set. The security aspect here is that if an item does not have a name you can reference, you cannot interact with it, so there is isolation.

Not everything in the system is namespaced; there remains a lot of global state. Clocks, for example, do not have a namespace, so if a container sets the system time it affects everything running on the kernel. Most of these can only be affected by a process running as root.

Another issue is that the Linux kernel interface is huge, and there are bugs hiding in it. There are over 300 system calls, and many thousands of miscellaneous `ioctl` operations, and a bug in validating user input on one of these can lead to a kernel exploit.

We do have a lot of ways to mitigate these risks, however, which we will cover below.

Kernel updates

The most basic recommendation is to keep your kernel up to date with security fixes. It is not always clear what changes are security issues, as many bugs may be exploitable, but

no one has found an exploit yet. This means you will need regular reboots of your container host machines, and hence also to restart your all of your containers.

Clearly you do not want all of your cluster rebooting at the same moment, thus taking all services down and losing quorum on distributed systems, so managing this needs some thought. CoreOS machines that detect that they are in a cluster, because they are running `etcd`, will **take a reboot lock** in `etcd` so that only one machine at a time reboots. Other systems will need a similar mechanism for staggered reboots.

Container updates

You must keep the host kernel and the host OS updated, and keeping the running containers patched for security updates is a key requirement.

If you are running "fat" containers with a whole host OS in them, such as RHEL or Ubuntu, then it is pretty simple to keep them updated. You can just run the same tooling as you would for a virtual machine. This takes no advantage of a container based workflow, but at least it is a well understood problem.

The situation people are worried about is if Docker is used as a way to get developers to put random containers that no one really knows what is inside them into production. Clearly this is not what you want to happen. Containers need to be reproducibly built from scratch, and must be rebuilt if there are security issues in the components if they are not being updated at runtime.

The closest way to do this to traditional practices is to take a traditional distro, configure it with tools like Puppet, and then use that as a base Docker image.

The microservices route is to make the container only contain a statically linked binary, as produced for example by Go, so the build process is simply to rebuild the application with updated dependencies. Then the upgrade problem becomes a build time dependency management problem. A Java application may also be similar to this.

Between these extremes there are a lot of other models; the important thing is to have a model, and ideally to have tests that can test build artifacts. For example, after the `bash` **shellshock bug**, you want to be able to inspect the containers you have in production and test if they contain bash and are vulnerable.

suid and guid binaries

Unix has long had a rather poorly designed privilege escalation mechanism where a file can be marked as `suid` or `guid`, in which case it runs the program as the owner (or group) of the file rather than the user running the program. Usually this is used to run programs as root that need special privileges. If programs are well written they will drop this root status as soon as possible, before parsing any user input and having done as little as possible. If they do not, there is a risk that they can be subverted.

Typical binaries that may be suid root include `su`, `sudo`, `mount` and `ping`. Most of these are not needed inside a container, so they can be removed, the suid bit removed, or the container root mounted with the `nosuid` option so they are ignored. This is something your security test suite can test for.

Note that we have not yet seen many distributions specifically designed to run inside containers, which would address this type of issue. Current distributions assume that these basic commands are needed. Some lightweight container base systems use the small **Busybox** core tools, which do not implement suid programs securely, as they do not drop privileges, and this should never be run with suid enabled.

Commands like this:

```
find / -xdev -perm -4000 -a -type f -print
find / -xdev -perm -2000 -a -type f -print
```

will look on a system for all of the suid and guid files.

root in containers

Containers should be designed so that nothing in them needs root privileges. In particular you should not be using `docker run --privileged ...`, which will run a container with full root access, and the ability to do anything that the host can do.

Capabilities (see below) are one way of giving a subset of root's capabilities to a process if necessary.

User namespaces (see below) are intended to provide a magic root-but-not-root unicorn, allowing root usage. We discuss this magic in more detail below.

Unfortunately, many existing containers do require root access, often for not very good reasons that just need fixing. An example is the **docker registry**, which creates lock files in a directory owned by root, unless you disable search, but the issue is still open.

Capabilities

Linux has some finer grained permissions for the capabilities that root has, and these can be given individually to a container. The **man page for capabilities(7)** lists which capabilities correspond to which actions. For example:

```
docker run --cap-add=NET_ADMIN ubuntu sh -c "ip link eth0 down"
```

will take the `eth0` interface down in a container with the `NET_ADMIN` permission only, which is the minimum needed to do that. This should allow running binaries that might otherwise be suid or which would otherwise need the whole container to run as root, but in

general it should be avoided, and containers should run with no capabilities for maximal security.

seccomp

While capabilities restrict the kind of actions that can be taken, seccomp filters can completely remove the ability to use specified system calls, or calls with certain arguments.

The difficulty with this is knowing which calls your application may need to use. You can take traces, but you need 100% code coverage, which is difficult. Your code may change and the calls used may change. So for general purpose use cases the easiest strategy is to blacklist system calls that are essentially administrative, and not generally used by applications, or are largely obsolete. About 25% of calls fall into these categories.

At the time of writing only the Docker lxc backend has hooks for running a seccomp filter, not the default **libcontainer** backend. There are sample filters in the **contrib directory in the docker repo**. It would also be possible to get an application to set its own filters.

Kernel security frameworks

Linux supports several kernel security frameworks, the best known being SELinux, designed by the NSA, which ships with RedHat Linux. There is also the similar AppArmor, which ships with Ubuntu.

SeLinux is a framework for implementing a **mandatory access control** policy. It is important to note that it is just a framework, and the actual policy must be defined. Very few people define policies, and it is a complex and poorly documented process. So most people use the vendor provided policies if that; indeed the most popular Google search completion for SELinux is still “disable”, despite the existence of a **coloring book** explaining it.

Defining security policies that actually lock things down is difficult if you do not work in an organization with a long running structure like the US Department of Defense where these originated. They could be used in principle, however, to isolate access to different types of data, for PCI compliance, HR data, or personal information. Unfortunately, the tooling to support these uses is rather lacking.

We would recommend however not disabling vendor policies if possible, and understanding how to label items to allow access. The vendor policies are better than no policy. Policies for containers are relatively new, and may not always work well.

Docker supports SELinux, since version 1.3, although it is off by default. `docker --selinux-enabled` will enable it and then options such as `--security-opt="label:user:USER"` can set the user, role, type and labels when containers are run.

Resource limits and cgroups

The kernel cgroup facility was created by Google for running applications at scale in its Borg scheduler, a precursor to Kubernetes.

A cgroup limits the resources allocated to a group of processes, typically a container. There is a large and complex set of cgroup controllers, but the important ones relate to restricting CPU time, memory and storage.

The simplest restrictions are those on memory and CPU access. Memory usage can be set with `docker run -m 128m`. You can set which CPUs a container can run on with `docker run --cpuset=0-3` and allocate shares of CPU time with `docker-run -cpu-shares=512`.

The important point with these is to stop an application affecting others running on the same host by using up all the memory, IO nadwidth or CPU time.

Depending on how you set up containers there may well still be some interference. For example, cache will be shared unless you completely allocate CPUs, and IO is contended if you share IO devices such as network or disks. How much this matters depends upon workloads and how much you oversubscribe resources, but usually this is a throughput rather than a security issue, although **side channel attacks** are possible.

Docker 1.6 adds the ability to attach a container to an existing cgroup with the `–cgroup-parent` option. This means you can manage cgroups externally from Docker with other tools, and then select which cgroup to add containers to. This lets you use all of the controls for cgroups, whether or not these are exposed in the docker command line.

ulimit

Docker 1.6 introduces the ability to control `ulimit` per container. This is an old Unix facility to control resources on a per process basis, a somewhat weaker concept than cgroups which apply limits to a group of processes. Keep in mind that `ulimit` will also let you configure maximum numbers of processes. For many purposes `ulimit` may be simpler than using cgoups for resource control, and more familiar to sysadmins.

Previously containers would inherit the `ulimit` from the docker process, for which the limits were generally set fairly high. Now you can for example use

```
docker -d --default-ulimit nproc=1024:2048
```

to set the default ulimit for the number of processes that can be created to a 1024 soft limit and 2048 hard limit. The soft limit is the limit that will be enforced, but a process can increase the soft limit up to the hard limit.

You can then overwrite the limits on a per container level, for example:

```
docker run -d --ulimit nproc=2048:4096 httpd
```

This will increase the process ulimit for just the httpd container.

User namespaces

User namespaces were added later than the other namespaces in the Linux kernel and are somewhat more complicated.

The idea is like other forms of namespacing, but for user ids (uid) and group ids (gid). In particular, inside a container that is in a user namespace the root user, uid 0, can be mapped to a different, unprivileged user in the host.

This means that in the host system, the container root user is simply a normal user, and cannot do anything special. So in what way is it root? It is root with respect to resources that only belong in its container, such as the container network interfaces; so it can reconfigure the container network interface for example, or bind to port 80.

This introduces more complexities, as uids are stored on filesystems to assign permissions to files, and therefore they will mean different things depending on the namespace. This also meant there was a long delay from the time the feature was introduced until it was production ready, not helped by some major security holes being found in the implementation. These delays, for example, mean that it missed the deadline for RHEL 7.0, and there is not yet direct support from Docker, although it is expected soon and there is some support in the lxc driver.

Another less visible advantage of user namespaces is that namespaces can be created without root permission at all. This allows the Docker daemon to internally reduce the amount of code that needs to run as root.

The most basic feature, just to introduce a root user in a container that is not root in the host system, covered in **pull 12648** has missed the deadline for Docker 1.7, and is now being targeted for Docker 1.8, due to conflicts between the user namespace code and the libnetwork code. More complex abilities of user namespaces are even further off.

Image verification

Docker 1.3 introduced the beginning of a roadmap toward **verification of Docker images**. This is just the beginning; the aim is to have a full model along the lines of Linux package managers, where you have a set of keys you trust, which may include trusted vendors as well as signatures from your organization, and to not allow running unsigned images.

The current implementation is just a beginning, as it only warns on signature failure and does not block the install of unsigned packages, so it does not really offer any security benefits yet. But it is the beginning of a roadmap; the **signed images issue** is a good place to see what is planned and follow the implementation.

Running the docker daemon securely

By default the docker daemon is only accessible from a local Unix domain socket, which means that access can be controlled locally via the permissions on the socket, and no remote access is possible.

Access to the docker daemon gives full root access on the computer, as you can run a docker container as root to run any command on the host, so protecting access is important.

If you force docker to bind to a tcp port for remote control (rather than for example control over ssh), using the `-H` option, then you need to control access with iptables and SSL. This is not a recommended behavior for most use cases.

Monitoring

Monitoring containers is important in order to know what is going on, including finding security related issues. We have a chapter on monitoring, so start there in designing your monitoring strategy.

Devices

If your containers need access to device nodes that provide access to hardware or virtual devices, use the `--device` option to pass through exactly the device that is needed and set permissions.

For example `docker run --device=/dev/snd:/dev/snd:r ...` will pass through the `/dev/snd` audio device, making it read only in the container.

Device nodes are part of the attack surface, as they allow `ioctl` access, and there may be bugs in the underlying kernel driver, especially for unusual devices. So a policy of providing devices only as needed and with minimal permissions is best.

Mount points

Docker, when it uses the default `libcontainer` driver, is careful to mount necessary virtual file systems with read-only permissions. If you use the lxc driver, you need to do this yourself. Write access to filesystems such as `/sys` and `/proc/bus` can lead to host compromise if there is root access in the container.

ssh

Do not run ssh in your containers. Get used to managing them from the host. Not only does this simplify your containers, but it removes a whole level of complexity of access. Get used to the tools that Docker provides to let you see what is going on in a container when you need to.

Simplified containers are much easier to manage, and entering containers from the host is easy; Docker effectively ends up managing the processes very well, and ssh is not necessary and adds complexity.

Secret distribution

Services need keys in order to access other services, such as keys to access AWS, or to validate whether they can join clusters or access resources. Key management is hard, and is not yet a solved problem but there are better and worse ways, and useful tools are starting to appear rapidly.

Keys should be distributed only on an as-needed basis, so if they are discovered, the least possible access is compromised. Keys should be rotated regularly so a one off breach becomes time limited. Keys should not be checked into source code, as updating keys should not require a new deployment, and they will end up in a public github repo.

Being able to audit access to keys is also a desirable goal, so usage can be tracked.

Services that are appearing include **Keywhiz from Square**, and **Vault** from Hashicorp. Kubernetes also has an excellent **design document on secret management**, which will be the basis for a secret management framework.

Location

If your services running on one host or virtual machine are all services that have the same level of access to the same data, then you can potentially worry less about isolation. This is after all no worse than a monolithic application after all, where there is no real isolation of components.

While microservices do allow you to build a more secure architecture with high levels of privilege separation, doing this for applications not having access to sensitive data is not a priority. You want to concentrate your security efforts where they will bring the most benefit.

User facing services, which face untrusted input, are clearly a weak point and should be isolated from any access to important data. PCI compliance related endpoints should not run on the same hosts as other services, and should be isolated to their own cluster to reduce the audit boundary.

In the next chapter we will learn all about building images in Docker.

7 Building Images

All containers run off of an image, and consequently, dominating the art of building images is essential when building your Docker infrastructure.

How you build your images will determine how fast the containers can be deployed, how easy is to get the logs from the container, how much you can configure them, and how safe they are. Although the first concern when building an image is that the containers that run it work as expected, all of these listed factors become very important in production.

Before we get down to the business of building images, we need to understand a few aspects of how they are implemented.

Not your father's images

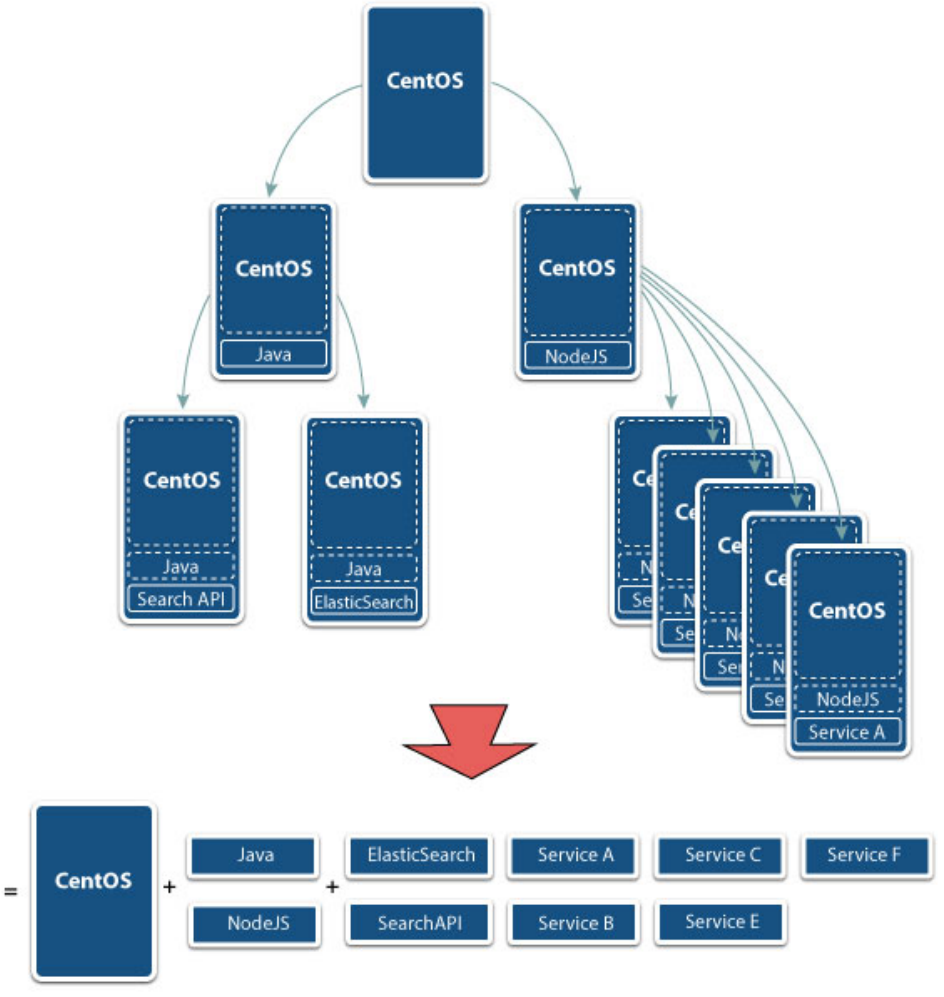
Although on the surface Docker images are not much different from the ones used for Virtual Machines (VM), they are implemented quite differently. VM images provide full filesystem virtualization: the filesystem in the image can be completely different from the host filesystem where the image resides. VM images are usually implemented as volumes that live as large files on the host operating system. Once a volume is allocated for the VM, the guest operating system in the VM will create and format one or more paritions on this volume. The VM hypervisor will present these files to the guest operating system as a raw disk.

This approach to virtualized filesystems provides a great deal of isolation and flexibility, but it can be inefficient. For example, you lose efficiency when you run multiple VMs using the same image or you need slightly different images that all stem from the same base image. The standard approach to cloning a VM has been to create a new copy of the file system for the new image so the filesystem in both images can evolve independently. This approach is costly both in disk space consumed and the time that it takes to create such copy, and because of this VM vendors have relied on Copy-on-Write techniques to make the use of images more efficient in these scenarios where the plain copy won't work well.

Copy on Write and Efficient Image Storage and Distribution

Copy-on-write (CoW) is a technique that saves time and space when creating and running many processes start from the same baseline data. In the case of virtualization we use CoW so that if, for example, 20 Virtual Machines (VMs) need to use the same base image you don't need to create 20 copies of such image (one per VM). Instead, all VMs can start from the same image file, which results in much faster startup times and a big save in terms of the disk space required to run all VMs. CoW makes the guest Operating System (OS) in each VM believe that they are making changes to the file system in the base image independently, and it does so by providing each VM an overlay on top of the shared base image that can be modified independently of the other VMs. Whenever the OS attempts to make a change to the file system, it does so on this overlay, leaving the base image intact.

When the OS wants to make changes to the file system, the VM, behind the scenes, copies the disk sector or sectors to be edited onto the overlay and presents these copies to the guest OS as if they were the original ones. Then, the hypervisor allows the guest operating system to modify the copies in the overlay, leaving the original sectors in the base image intact. From there on, the original shared copy of these sectors are not visible to this VM anymore, only the copies in the overlay. The hypervisor provides the guest OS the 'illusion' of a file system that is the result of merging the overlay and the base image, as if it were a single volume.



Docker images are natively based on CoW techniques, and unlike standard VMs, Docker's images are not full virtualized; they are built right on top of the host's file system. Whether this approach provides any peformance advantages over full virtualization is up for debate and depends very much on the use case. For example, CoW in the VM world is usually sector-based, that is, only the file disk sectors that change on the base image are copied and edited on the overlay, whereas with Docker the full file is copied and edited, so if only a portion of a large file changes, the whole file needs to be copied. On the other hand, with Docker images there is no file system translation needed between the guest and the host operating systems. What's important for our discussion of building images in

Docker is that Docker takes the CoW approach a bit further by making it very easy to stack many CoW overlays to create an image or a family of related images.

Docker leverage of Copy-on-Write

Docker uses CoW for two main purposes. The first one is to allow you to build images interactively, adding one layer at a time. The second has much deeper implications and has to do with the storage and delivery of images. Most often, when we build our systems, we do so in a way that all services are based on the same small set of operating systems, even sharing some basic configuration. Container images in such setups only differ from each other on the *last mile* of their configuration, this last mile containing only what makes this image different from the others, and sometimes containers use the same exact images. In these scenarios CoW is very effective in saving time and space.

Docker also uses CoW to run each container on an overlay and never directly on the image. The original image is used in Read-Only mode, and any changes that the container might make to the file system are performed solely on the overlay. You might read the the Docker literature that Docker images are immutable, and this is exactly what this means: once an image is created, the image never gets modified, so all you can do is build new images off of it.

What makes Docker's usage of overlays really powerful is how these overlays can be shared across hosts. Each overlay contains a reference to their base image, which is in turn another overlay. Each overlay has a unique ID, and can optionally have a name and a version. Docker images name overlays stored and shared in image repositories. When a container is deployed, Docker checks the image required by the container that is already present in the local repository. If it doesn't exist locally, Docker checks the image repository for this image and pulls references to all of the overlays that are part of the image and determines which ones of those layers are already on them locally, downloading the missing overlays.

This approach can reduce the space required to hold all of the images needed in a host, and can dramatically reduce the download times for new images. For example, in a scenario running 10 containers off of 10 images that all stem from the same base CentOS 7 image, the host only needs to download the base image once and then each of the 10 different overlays, as opposed to downloading 10 images each containing a full copy of CentOS 7. Similarly, downloading an updated image would require only downloading the last few overlays.

We will discuss in more detail how you can leverage these features later in this chapter, but first let's look at the primary aspect of building an image: making it work.

Image building fundamentals

At the most basic level, building a container image (image from here on) can be done in two ways. The first method is to start a container from a base image (ubuntu-14.04), run a series of commands inside the container, such as install package and edit configuration files, and once the image is at the desired state, save it.

Let's see how it works. In one terminal you start a container running `/bin/bash` interactively, using a base image of `ubuntu`. Once at the shell inside the container, we create a file in the root directory named `docker-was-here`. This operation should not modify the base image, instead, the new file is created in this container's filesystem overlay:

```
[~] —docker run -ti ubuntu /bin/bash
root@4621ac608b25:/# pwd
/
root@4621ac608b25:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc
root  run  sbin  srv  sys  tmp  usr  var
root@4621ac608b25:/# touch docker-was-here
root@4621ac608b25:/#
```

Now, in a second terminal we create a new image based on what is currently on the previous container, with ID `4621ac608b25` in the previous example.

```
 docker commit 4621ac608b25 my-new-image
6aeffe57ec698e0e5d618bd7b8202adad5c6a826694b26cb95448dda788d4ed8
```

Finally, we start a new container on this second terminal, this time using our newly created image `my-new-image`. We can verify that the image contains our very own `docker-was-here` file.

```
$ docker run -ti my-new-image /bin/bash
root@50d33db925e4:/# ls
bin  boot  dev  docker-was-here  etc  home  lib  lib64  media
mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@50d33db925e4:/# ls -la
total 72
drwxr-xr-x  32 root root 4096 May  1 03:33 .
drwxr-xr-x  32 root root 4096 May  1 03:33 ..
-rwxr-xr-x   1 root root    0 May  1 03:33 .dockerenv
-rwxr-xr-x   1 root root    0 May  1 03:33 .dockerinit
drwxr-xr-x   2 root root 4096 Mar 20 05:22 bin
drwxr-xr-x   2 root root 4096 Apr 10  2014 boot
drwxr-xr-x   5 root root  380 May  1 03:33 dev
-rw-r--r--   1 root root    0 May  1 03:31 docker-was-here
drwxr-xr-x  64 root root 4096 May  1 03:33 etc
                        ....
```

```
drwxr-xr-x  12 root root 4096 Apr 21 22:18 var
root@50d33db925e4:/#
```

Although this interactive method for building build images is quite straightforward, it does not lend itself to reproducibility and automation. For production settings it is very desirable to automate the building of images in such a way that it can be easily reproduced. Docker provides a method to do just this, based on a file named `Dockerfile`.

A `Dockerfile` contains a series of instructions for Docker to run in a container in order to generate an image. These instructions can be divided into two groups: the ones that change the filesystem of the image and the ones that change the metadata of the image. An example of an instruction that changes the filesystem would be ADD, which will write it into the image's file system files from a remote location as defined by a URL, or RUN, that will run a command on the image. An example of the latter would be CMD, which sets up the default command with its parameters to run when the container process starts.

When using `docker build`, Docker starts a temporary container with the base image indicated by the Dockerfile instruction FROM, and then runs each instruction in the context of the container. For each instruction, Docker creates an intermediate image. This is to make it easy for the person building the image to do so incrementally: when you change or add an instruction in the Dockerfile, Docker knows that there haven't been any changes in the previous instructions and so it uses the image built after running the last image.

As an example, this is the `Dockerfile` to build the same image that we previously built interactively, then create a new directory, and inside this directory, create a file named Dockerfile with the following:

```
FROM ubuntu
MAINTAINER Me Myself and I
RUN touch /docker-was-here
```

Next we tell Docker to build the image `my-new-image` using this dockerfile:

```
$ docker build -t my-new-image .
```

Docker by default looks the `Dockerfile` in the current directory. If you are using a different name or the dockerfile is somewhere else, use `-f` to tell Docker about the path to the dockerfile to use:

```
$ docker build -t my-new-image -f my-other-dockerfile .
```

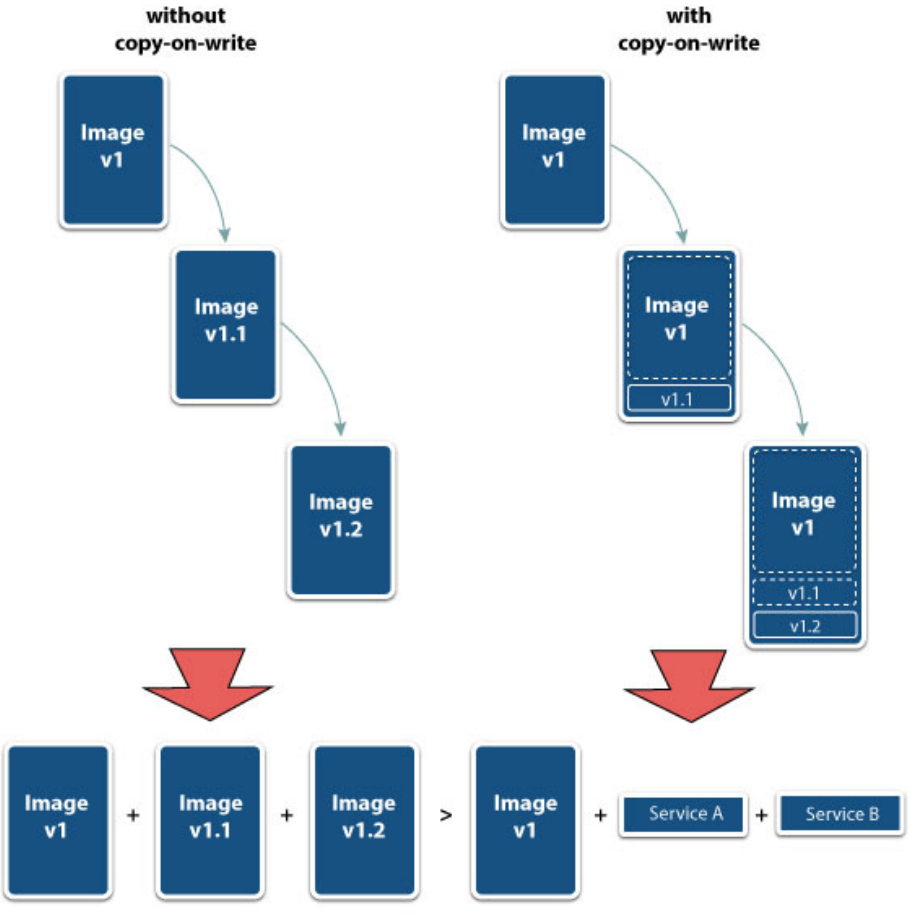
Layered File Systems and Preserving Space

As discussed in the previous section, Docker images present a layered architecture where images consist of a stack of filesystem overlays. Each layer is a set of additions, modifica-

tions and deletions of files from the previous layer. When a layer adds a file, a new file is created. When a layer removes a file, the file is marked as deleted, but notice that it is still contained in the previous layer(s). When a layer modifies a file, depending on the storage driver that Docker is running (more on this later in the storage chapter), either the whole file is recreated in the new layer, or only some of the disk sectors in this file are replaced by new ones in the new layer. Either way, the old file is left untouched in the previous layer(s) and the new layer contains new changes to it. At each layer there is an image that is the result of sequentially overlaying previous layers on top of the base image. At the top is the resulting image, which again, is the result of all of the previous layers.

When building Docker images, you usually start with an existing base image, which may already consist of many layers. Docker runs each instruction in the `dockerfile` in order, and at the end of each instruction Docker generates a new layer with the filesystem changes resulting from running the instruction. This is a great feature because it allows for the incremental development of images without having to wait for all of the instructions to run every time. For example, if Docker fails to build an image because the 10th instruction contains an error, then when you try again to build it after fixing the failing instruction, Docker will not run again the previous nine instructions. Instead, it will start at the last correct build layer, and then resume the build starting from the previously failing instruction. This is a great time saver, since some instructions might run commands that take a while to complete.

This layered architecture is also advantageous during deployment because when deploying a new image, chances are some of the deeper layers are already on the host, so only new layers need to be sent over the network. When running many containers off of the same or similar images, this feature will greatly reduce the time and space it takes to do so.



This layered architecture also comes with some caveats that you need to consider in real life scenarios. One is that images never shrink. If an image with all of its layers is 500MB in the filesystem, any custom image that extends it will be at least 500MB in the filesystem, even if upper layers remove files present in lower layers.

Image size matters especially when installing these images on hosts, both because the time it takes for the image to be downloaded to the host depends on it size, but also because the larger the image the more disk space is needed on the host. Size also matters greatly during development, as for a new developer, for someone starting with a new set of Docker hosts or even for Continuous Integration / Continuous Deployment servers, it can take a large amount of time to download all the images for the containers needed during development. This can be frustrating considering that Docker is meant to speed up development processes.

NOTE: Making images smaller is all the more important if you are deploying a microservices architecture, and most of what follows in this section will not make much of a difference if you are deploying large VM-like containers, as you probably are using a full fledged operating system.

START SMALL

Preserving space in images requires starting with the smallest possible base image. At the extreme, you can start with an empty file system and deploy your OS in there, but this approach is probably not the most widely used.

The next options are micro-distributions like **busybox** or **alpine**. Busybox is a mere 2.5MB and is a bare bones distro initially created for embedded applications. It contains the bare minimum of Unix utilities to get you going, but you also can create your own busybox distro with extra (or removed) commands. Busybox is often good enough of an image to support running statically-compiled binaries, such as a process written in Go.

Alpine is based on Busybox and extends it by adding a security-focused kernel build and a package manager named `apk`, and is based around **Musl libc**, a lighter and potentially faster version of libc. Alpine can be used as a general purpose Linux distribution for your containers, but you will have to work harder to get it configured to your needs: although Alpine provides a package manager, the list of available packages is much smaller than full distros like Debian or CentOS. The advantage of Alpine in front of full fledged distros is that it has many less moving parts, and therefore it's both smaller and simpler to understand, and as a corollary, it is also easier to secure.

The next option is using one of the container-optimized versions of mainstream Linux distributions, like Ubuntu or CentOS. These containers usually run slimmed-down versions of the full distro, with anything desktop removed and with a configuration that is optimized for production servers. These images are usually a few hundreds of MBs, for example Ubuntu 14.04 is ~190MB and CentOS 7 is ~215MB, but they provide a full fledged operating system. This is by far the easiest place to start when building a custom image, since their support of packaged services is excellent and there is plenty of documentation and howtos around the web to look for inspiration. The Docker registry is full of these images, most of them supported directly by Docker Inc.

It is always a good idea to standardize to one particular image version and use it across all of the containers possible. If all of the containers in a host have the same base image, once the first container is downloaded the rest of the containers will download much faster since they don't have to re-download the base image layers. Notice though that this only works when you download an image **after** the base image is already downloaded. At the time of writing this, if you download many images in parallel when the base image is not yet downloaded, the base image will be downloaded once per image since it doesn't exist yet in the local repository when the image downloads started.

Keeping images small

The next step after choosing a small base image is to keep the image small after running your Dockerfile.

Each time a command in the Dockerfile is run, a new image layer is generated. When a layer is generated, a new minimum image size is set: even if you remove files in the next command in the Docker file, no space will be released and the image size on the once on host file system won't shrink.

For this reason, how you organize your commands in the Dockerfile will have an impact in the final image size. An example of this is installing a package via the package manager: when you invoke the package manager, its indices get updated, it downloads some packages to its cache directory, and then it expands packages on a staging area before the files in the packages are finally put in their final locations on the file system. If you run the package installation command as is, the cached package files that will never be used will be part of the image forever. But if you remove them as part of the same installation command, it will be as if these files never existed.

For example, this is how you can install Scala and perform the clean up in a single step.

```
RUN curl -o /tmp/scala-2.10.2.tgz http://www.scala-lang.org/
files/archive/scala-2.10.2.tgz \
    && tar xzf /tmp/scala-2.10.2.tgz -C /usr/share/ \
    && ln -s /usr/share/scala-2.10.2 /usr/share/scala \
    && for i in scala scalc fsc scaladoc scalap; do ln -s /usr/
share/scala/bin/${i} /usr/bin/${i}; done \
    && rm -f /tmp/scala-2.10.2.tgz
```

In the example above, if you were to run the above commands independendtly like this:

```
RUN curl -o /tmp/scala-2.10.2.tgz http://www.scala-lang.org/
files/archive/scala-2.10.2.tgz
RUN tar xzf /tmp/scala-2.10.2.tgz -C /usr/share/
RUN ln -s /usr/share/scala-2.10.2 /usr/share/scala
RUN for i in scala scalc fsc scaladoc scalap; do ln -s /usr/
share/scala/bin/${i} /usr/bin/${i}; done
RUN rm -f /tmp/scala-2.10.2.tgz
```

the next images would carry the `.tgz` file, even though after the last command the file would not be visible in the filesystem.

Making images reusable

There are a two ways to configure the process or processes running inside of a container. One method is passing the configuration to processes inside the container through environment variables. Another is by mounting configuration files and/or directories into the

container. Both methods take place at the container startup time. Both methods are useful and have their applications, but they are also quite different in nature.

CONFIGURING VIA ENVIRONMENT VARIABLES

When Docker starts up a container, it can forward environment variables to the container process, which will in turn be forwarded to the process running inside the container. Let's see how it works. Start a container and run a command in the shell to print the value of the environment variable MY_VAR:

```
$ docker run -e "MY_VAR=docker-was-here" --rm busybox /bin/sh -
c 'echo "my variable is $MY_VAR"'

my variable is
```

The environment variable is not predefined in the container, therefore there is no value for it. Now run the same command inside the container, but this time we are passing an environment variable to the container via Docker:

```
$ docker run -e "MY_VAR=docker-was-here" --rm busybox /bin/sh -
c 'echo "my variable is $MY_VAR"'

my variable is docker-was-here
```

Ideally the process running inside the container will be fully configurable via environment variables. Sometimes we containerize services that take their configuration via configuration files. We will discuss how to handle these scenarios in the next section, but for now we focus on the straightforward use of environment variables.

Using environment variables provides a great deal of isolation between the process and its configuration, and is considered to be the better approach in **12 factor**, a manifesto for building service-based applications. Docker encourages this pattern by implementing an option of passing these environment variables to the container at start time.

The benefit of this separation is that you can use the same image regardless of how you compute the configuration under which you want to run your container. When the containerized process takes all of its configuration via the environment, all of the configuration responsibilities belong to the process that calls Docker to start the container. This pattern allows for a great deal of flexibility since the configuration can come hardcoded in the script starting the container, from a file, from some distributed configuration service, or even from a scheduler.

Making an image configurable via environment variables when the process is not

Sometimes you need to wrap a service that is not configurable via environment variables. The most common scenario is a process that reads its configuration from one or more configuration files, such as nginx.

USING TEMPLATE FILES

There is a widely used pattern to handle these scenarios: use an entry-point script that takes the environment variables and generates the configuration files on the file system, then calls the actual process, which will read those newly generated configuration files at startup time.

Let's see an example. This is how you can build a container that sends push notifications to iOS and Android phones using **node-pusherver**. For this example, we create a shell script named `entrypoint.sh` that will be added inside the container in the `dockefile`:

```
from node:0.10

RUN npm install node-pushserver -g \
    && npm install debug -g

ADD entrypoint.sh /entrypoint.sh
ADD config.json.template /config.json.template
ADD cert-dev.pem /cert-dev.pem
ADD key-dev.pem /key-dev.pem

ENV APP_PORT 8000
ENV CERT_PATH /cert-dev.pem
ENV KEY_PATH /key-dev.pem
ENV GATEWAY_ADDRESS gateway.push.apple.com
ENV FEEDBACK_ADDRESS feedback.push.apple.com

CMD ["/entrypoint.sh"]
```

This `dockerfile` has many default values for environment variables. As you can see, these environment variables determine, from the service's own port to MongoDB's host/port and also the location of the needed certificates and even the Apple servers to use (they can be different in development vs. production). Finally, the process that the container will run is our own `entrypoint.sh`, which looks like this:

```
#!/bin/sh

# render a template configuration file
# expand variables + preserve formatting
```

```
render_template() {
  eval "echo \"$(cat $1)\""
}

## Refuse to start if there is no monogodb prefix
[ -z "MONGODB_CONNECT_URL" ] && echo "ERROR: you need to speci-
fy MONGODB_CONNECT_URL" && exit -1

## escape quotes so that they're not removed by rendering
cat /config.json.template | sed s/\"/\\\\\"/g > /
config.json.escaped
## Render the template
render_template /config.json.escaped > /config.json
cat /config.json
/usr/local/bin/pushserver -c /config.json
```

There are a few things to note in this script file. First, we define a function `render_template` that takes a single variable containing the contents of a file and returns the same contents, but with the environment variables in it expanded.

Next we have some gatekeeping for failing fast if there is not some key configuration present. In this case, we are requiring that the caller provides an environment variable named `MONGODB_CONNECT_URL`, for which there is no default.

Finally we have the part that does the generation of the configuration file from a template. The template looks like this:

```
{
    "webPort": ${APP_PORT},

    "mongodbUrl": "${MONGODB_CONNECT_URL}",
     "apn": {
        "connection": {
            "gateway": "${GATEWAY_ADDRESS}",
            "cert": "${CERT_PATH}",
            "key": "${KEY_PATH}"
        },
        "feedback": {
            "address": "${FEEDBACK_ADDRESS}",
            "cert": "${CERT_PATH}",
            "key": "${KEY_PATH}",
            "interval": 43200,
            "batchFeedback": true
        }
    }
}
```

We escape the double quotes because the super-simple rendering engine `render_template` will remove them otherwise. And then, we call `render_template`,

which takes the file with the escaped double quotes and generates the final configuration file. This is how it looks:

```
{
    "webPort": 8300,

    "mongodbUrl": "mongodb://10.54.199.197/staging-
pushserver,mongodb://10.54.199.209?replicaSet=rs0&readPrefer-
ence=primaryPreferred",
     "apn": {
        "connection": {
            "gateway": "gateway.push.apple.com",
            "cert": "/certs/apn-cert.pem",
            "key": "/certs/apn-key.pem"
        },
        "feedback": {
            "address": "feedback.push.apple.com",
            "cert": "/certs/apn-cert.pem",
            "key": "/certs/apn-key.pem",
            "interval": 43200,
            "batchFeedback": true
        }
    }
}
```

Finally, this script calls the actual service via `/usr/local/bin/pushserver -c /config.json`, which loads our newly generated `config.json` file.

MOUNTING THE CONFIGURATION FILES

Notice that the previous configuration file that we generated also loads two certificates, and even though we could pass those as environment variables too, like with `echo ${CERT} > /certs/apn-cert.pem`, in this instance we are supplying them as mounted files. This is an alternative method for handling these containers that only take files for their configuration.

When starting a container you can mount local directories and files into the container filesystem, and this happens before the container process is started. With this in mind, an alternative way to configure the container above is to run the script that generates the configuration file **before** starting the container, and then mounting the file inside the container. The drawback of this approach is that you need to find a proper place on the host where to write these configuration files, potentially a different version for each container, and then properly clean up these files when containers are destroyed. Since there is no added benefit to customizing configuration files outside the container vs. inside the container, the latter options is preferable since it contains the config files inside the container.

Make images that reconfigure themselves when Docker changes

Sometimes we need some containers to be aware of other containers on the same host, and provide services to them. An example of this is a container that provides log collection services, which sends all the other containers' logs to some log aggregator like **Kibana**. Other needs can be related to the monitoring of such containers.

These kinds of containers need access to the host's Docker process so they can communicate with it and query about existing containers and their configuration.

Before discussing how to implement such containers, let's discuss the kinds of problems these containers can solve using a logging example: we want to send the logs from all containers to some log aggregation service, and we want to do it from within a container. This requires us to run a log collection process like **logstash** or **fluentd**, and configure it so that it picks up the logs from each container. Docker usually stores the logs for each container in its own directory, following this pattern: `/var/log/docker/containers/$CONTAINER_ID/$CONTAINER_ID-json.log` -- in this case for json logging.

One solution is to build our own log collector that understands this layout and can talk to Docker to query about existing containers.

Most of the time though you won't want to code your own log collector services, and instead configure an existing one. This means that the configuration for this log collector will change when new containers are added or removed from the host. Fortunately, there are already some tools to rebuild configuration files based on information coming from the host's Docker server.

One such tool is **docker-gen**. This tool uses provided templates to generate configuration files based on container information provided by Docker. The templating language that it offers is powerful enough for most tasks, and the way it operates is that it can either *watch* or *poll* the Docker process for changes in the containers (added, removed, etc.) and will regenerate the configuration files from the templates any time there is a change.

In our example, we wanted to regenerate the log collector configuration so that the logs for all containers are properly parsed, tagged, and sent to whatever log aggregator we use.

Let's see how this works with a real world example using *fluendD* as a log collector and ElasticSearch/Kibana as the log aggregator.

First we need to create our log collector container:

```
FROM phusion/baseimage

# Set correct environment variables.
ENV HOME /root

# Use baseimage-docker's init system.
CMD ["/sbin/my_init"]

RUN apt-get update && apt-get -y upgrade \
    && apt-get install -y curl build-essential ruby ruby-dev
wget libcurl4-openssl-dev \
```

```
    && gem install fluentd --no-ri --no-rdoc \
    && gem install fluent-plugin-elasticsearch --no-ri --no-
rdoc \
    && gem install fluent-plugin-record-reformer --no-ri --no-
rdoc

ADD . /app
WORKDIR /app
RUN wget https://github.com/jwilder/docker-gen/releases/down-
load/0.3.6/docker-gen-linux-amd64-0.3.6.tar.gz \
    && tar xvzf docker-gen-linux-amd64-0.3.6.tar.gz \
    && mkdir /etc/service/dockergen

ADD fluentd.sh /etc/service/fluentd/run
ADD dockergen.sh /etc/service/dockergen/run
```

The relevant parts of this Dockerfile is that we install fluentd, and then we install some plugins for fluentd. The first to send the logs to ElasticSearch, and the other named `record-reformer` that let's us transform and tag the logs before sending them over to ElasticSearch. Finally, we install dockergen.

Since we need to run both docker-gen and fluentd on the same container, we need some sort of service supervisor. In this case our image is based off of `phusion/base-image`, which is a *dockerized* and stripped down version of Ubuntu. One of the customizations that this image offers over stock Ubuntu is that it uses `runit` as the process supervisor. In the last two lines of our Dockerfile, we have two `ADD` directives that add the run scripts for both docker-gen and fluentd. Once the container starts up, these two scripts will be run and both docker-gen and fluentd will be running and supervised.

The startup script for docker-gen starts docker-gen with the following settings: it will watch for changes in the containers run by the docker host, and if there is any change, it will regenerate the file `/etc/fluent.conf` from the template `/app/templates/fluentd.conf.tmpl`. Once it is done, it will run `sv force-restart fluentd`, which forces a restart on fluentd (via runit), which in turn results in fluentd reloading the new configuration. Here is the startup file for docker-gen:

```
#!/bin/sh

exec /app/docker-gen \
     -watch \
     -notify "sv force-restart fluentd" \
     /app/templates/fluentd.conf.tmpl \
     /etc/fluent.conf
```

The startup script for fluentd is a bit more straightforward, as it just starts fluentd using the configuration file `/etc/fluent.conf` that docker-gen generates:

```
#!/bin/sh

exec /usr/local/bin/fluentd -c /etc/fluent.conf -v
`
```

The next thing we need are the template files that docker-gen uses to generate the configuration for FluentD. This is a non-trivial example used in a real world scenario:

```
## File input
## read docker logs with tag=docker.container

{{range $key, $value := .}}
<source>
  type tail
  format json
  time_key time
  time_format %Y-%m-%dT%T.%LZ
  path /var/lib/docker/containers/{{ $value.ID }}/
{{ $value.ID }}-json.log
  pos_file /var/lib/docker/containers/{{ $value.ID }}/
{{ $value.ID }}-json.log.pos
  tag docker.container.{{ $value.Name }}
  rotate_wait 5
  read_from_head true
</source>
{{end}}

{{range $key, $value := .}}
<match docker.container.{{ $value.Name }}>
   type record_reformer
   renew_record false
   enable_ruby false
   tag ps.{{ $value.Name }}
   <record>
     hostname {{ $.Env.HOSTNAME }}
     cluster_id {{ $.Env.CLUSTER_ID }}
     container_name {{ $value.Name }}
     image_name {{ $value.Image.Repository }}
     image_tag {{ $value.Image.Tag }}
   </record>
</match>
{{end}}

{{range $key, $value := .}}
<match ps.{{ $value.Name }}>
  type elasticsearch
  host {{ $.Env.ELASTIC_SEARCH_HOST }}
  port {{ $.Env.ELASTIC_SEARCH_PORT }}
  index_name fluentd
```

```
    type_name {{ $value.Name }}
    logstash_format true
    buffer_type memory
    flush_interval 3
    retry_limit 17
    retry_wait 1.0
    num_threads 1
  </match>
  {{end}}
```

There is a lot going on here, but focus on the key aspects of this template. For each container, we are generating 3 entries, one of type `source` and two of type `match`. The way we generate these entries in this example is by iterating over each container (`{{range ...}}`) and building the entry (e.g. `<source ...> ... </source>`) for each of them. Inside a `range` block, we can access the data for the current container with the variable `$value`, which is a dictionary that contains all of the information that Docker has about the container.

For example, in the `source` entry, we are telling fluentd where to find the log files for each container. The logs for each container are located inside `/var/lib/docker/containers`, inside a directory named after the container's ID, and in a file also named after the container's ID. This is done by:

```
path /var/lib/docker/containers/{{ $value.ID }}/{{ $value.ID }}-
json.log
```

We are also tagging the logs coming from this source with the container name, which will be useful later on for filtering:

```
tag docker.container.{{ $value.Name }}
```

The rest of the entries in the template follow the same structure. The first `match` entry is used to rewrite the log entry and add some extra information on it, such as the cluster name and the host name. Both of these values come from environment variables:

```
hostname {{ $.Env.HOSTNAME }}
cluster_id {{ $.Env.CLUSTER_ID }}
```

We also add other useful information coming from Docker: the name of the image that the container is running and the tag for the image. This way, we can filter the logs later on by the image name and even by the version of this image. This is quite helpful when different versions of the same image are displaying different error rates, don't you agree? These are the lines that add these tags:

```
image_name {{ $value.Image.Repository }}
image_tag {{ $value.Image.Tag }}
```

The final `match` entry sends the log entries to ElasticSearch, and its address and port also come from environment variables.

Now, with this setup, every time a new container is created or one is destroyed, the file `/etc/fluent.conf` is regenerated with the above 3 entries for each container. This way we get the logs from all of the containers to ElasticSearch properly timestamped and tagged.

Trust and Images

A common and well founded concern when using Docker images is how trustworthy they are. Docker and other container providers are working hard to provide a high level of trust on the images that you download and run. This trust comes at two levels. One is whether the image itself is trustworthy, and that the image is authored by a trusted developer, such as Docker Inc. or Red Hat. The other level is ensuring that the image downloaded is actually the image you intended to download. At the time of writing this, Docker still hadn't provided an end-to-end chain of trust, although some pieces are there already.

To protect ourselves from running images that contain malware or other risks, the best we can do now is to build the images by ourselves. Most of the images available are open source and need the Dockerfiles to generate them. Instead of downloading the image, copy the Dockerfile, inspect it, and then build the image from this Dockerfile. To be totally certain that the image is not compromised, you need to follow the image layers all the way to the base operating system. That is, if an image is based off of an image that is in turn based on a known OS image, then you need to verify and build the first two.

Make your images immutable

Although the container filesystems are writable, it is always better to treat them as read only and only write to them at startup time in the cases in which we need to generate config files. The main reasons for wanting to treat the container filesystems as read only are that these filesystems are slower than the host's filesystem, but also because the data can be easily lost when the container is destroyed. Obviously, if a container runs a database, you need to write the data somewhere, and in this case you can either use the container's own filesystem, or write on a host-mounted volume.

But chances are most of your containers don't need to write to the file system at all, since they don't hold data. In most of these cases, the processes still write to the filesystem to generate logs.

A common pattern among container practitioners is to write the logs to the process' standard output instead of logging to the file system. This way, you rely on Docker's own log collection to pick up those logs and you don't need to write to the container's filesystem anymore. Then on each host you run a log collector process that picks up these Docker-generated logs and sends them off to a central logging server for archival, analysis

and querying. This pattern is very prevalent when running microservices architectures where the number of different containers is large and dynamic.

When building images for 3rd party services, sometimes we cannot tell the service to output the logs to the standard output. For example, most web servers don't do that. But there is a relatively easy way to achieve the same behavior: linking the log file to standard output.

For example, `nginx` only writes the logs to log files in the filesystem. In this case, we instruct nginx to continue to do so:

```
access_log  /var/log/nginx/access.log  main;
```

But in the Dockerfile, we link `/dev/stdout` to this file, so when nginx is writing to `access.log`, it is instead writing to the container's standard output:

```
RUN ln -sf /dev/stdout /var/log/nginx/access.log
```

Summary

Understanding Docker's use of overlay file systems and how to build images that are nimble, configurable, reusable and that play well with the ecosystem provides a good foundation for building an efficient Docker infrastructure. As we've shown, sometimes we need to go a little out of our way to make Docker-friendly images using software designed with different configuration paradigms and runtimes in mind, but this extra effort pays off in the long run and it is relatively small compared to the time and headaches it will save us.

One of the key pieces of the Docker infrastructure is the image repository. The next chapter covers this topic in detail.

8 Storing Docker Images

If you've used Docker in development or production already, you will know that storing images is one of the easier things to accomplish with Docker images. Docker since day one has had a central repository that allows you to easily store images. This central model has made pushing and pulling images trival while allowing engineers to make their code and services extremly portable. There are three different ways to store Docker images: public, private, or save/load. Each of these options have their pro's and con's so it really depends on your type of environment, what works best, and company security requirements.

For open source or public projects it's suggested that you use the public repository. If you need higher security and better performance the private registry is suggested. If you need something custom then save/load is the way to go. A company should also consider the quantity and size of the images being stored. Image sizes are typically several hundred megabytes in storage so you'll need to make sure your repository will be performant while provisioning new containers.

By default when you use Docker to pull or push a image you'll be using the Docker Hub unless you specify your local repo. For instance, if you use this command you'll push directly from Docker Hub.

```
- docker push redis
```

If you specify a namespace before the image name then you can redirect to an internal private repository. For instance, this command will push from a local repo if repo.domain.com is hosted internally.

```
- docker push repo.domain.com/redis
```

Let's dive into each of the ways to store images.

Getting up and running with storing Docker images

The most common way to store Docker images for development and public use is to use the default repository in hub.docker.com (Docker Hub). You can think of Docker Hub as a

github for Docker images. It provides a great way to see many different types of images, so you can star or favorite the best ones, see the contents of the Dockerfile, and even see descriptions or comments about what the image does. It is very easy to get up and running with Docker hub. All you need to do is create an account and you'll get a single free repository to start using right away. After your account is setup you can push to Docker Hub by taking a newly built image by typing.

```
- docker push -t newrepository/webimage
```

Pulling an image from Docker Hub is also very easy.

```
- docker pull newrepository/webimage
```

When using the Public repository you need to be aware of several security settings. When using Docker Hub, images can be easily made public or private. If you're storing source code, security keys, or environment details within your image you should be very cautious of making the image public. If you have images that contain sensitive information Docker Hub does provide the ability to secure images by making the repository private (only accessible via the administrator or collaborators account). It also provides authentication to protect who can make modifications to your repository and the ability to assign collaborators to your repository.

If you're curious about the technology behind Docker Hub here is a link to a **Meetup presentation** done by Jérôme Petazzoni.

If you're just getting started with Docker try these **official images**.

Automated builds

Docker Hub offers a not very well known but rather great feature that allows you to get your container images built automatically by Docker Hub's servers. To set up the automated build point the Docker Hub repository to a Github or Bitbucket repository or a path in a repository that has a Dockerfile in it.

Once you've set up the automated build Docker Hub will then automatically build the new container image every time you make a change in the configured source code repository. The newly built container image is then pushed to the registry, marked as an automated build and made available for download.

Benefits:

- less work for you or your infrastructure
- automated builds have the base dockerlibrary images automatically updated with security patches \o/
- event driven approach (via Webhooks) - the images reflect latest up to date version of the application code - this is especially useful for the open source projects

Private repository

Another common way to store Docker images is the use of a private registry. Docker provides an open source server to store your images located **here**. The private registry allows companies to securely store their images behind their firewalls and VPN's to make sure their code and image repository is secure. It's also very easy to get the private registry up and running. Here is a link to **get started**.

When Docker first started out the private registry was part of the main code in Docker. Since the Docker registry is an extremely important part of its ecosystem they decided to split the private registry as its own product. Docker has since moved the code to its own branch and even made it a downloadable Docker image. The private registry has seen better days over the past year. If you started out early with Docker you know the private registry has gone from very unstable to now stable with many new improvements and releases.

Now that the private repository is stable many companies have started to use their own internal registry due to performance improvements and higher security requirements. If you're just getting started you should consider some architecture decisions if you're looking to run your own private registry. You will need to consider the network bandwidth, login credentials, ssl security, monitoring, and disk storage requirements.

Benefits:

- Speed - Having your repository inside your own network makes the push and pull of an image much faster.
- Security

Scaling the Private registry

When companies first start using the private registry they quickly install it on a server and they're are off and running. Companies soon realize images take up quite a bit of storage, consume a ton of network bandwidth, and will be exposed to the many file system issues that ship with the different types of file systems Docker uses (see the Data Storage chapter). If you're planning on running the private Docker registry in your own network you'll need to treat it like a first class server.

When running your own internal private registry. It is recommened to use network based storage and load balance the repository servers for redundancy. Let's take a look at a live production private repository.

This environment has a load balancer, two repository web servers setup in an autoscaling group, and uses S3 for backend storage. The environment consists of 526751 total objects in the repository. Objects are made up of the images and tags that get stored in the repository. The total size of this environment in bytes is 2678702030780 which comes out to 2.43TB of used storage.

This environment gets consistent bursts of network traffic that can spike up to 1Gbps, but are typically around 300mbps. Here are some images that capture two weeks of network throughput in and out of the web servers.

It's not that hard to scale the private registry but you should consider the storge and networking throughput as you continue to use Docker long term. This we hope gives you a good idea of the some of the metrics around this environment for when you play to implement your own. Let's dive into some deeper areas.

It is also good to check out the administrator's guide that Docker supplies located **here** or the deployment guide located **here**.

S3

We've seen how most of the companies running a private registry use S3 to store their images. Docker Hub also uses S3 to store their images. The big advantage here is nearly unlimited storage and the ease of administration. If you're using an AWS infrastructure the speed will also be very good since it will be localy routed to the S3 service. This also makes your private respository server immutable since the persistence storage will be S3 in this configuration. Allowing you to autoscale and load balance the web portion of your registry. You can use the S3 storage driver when configurating the private registry in the settings.

LOCAL STORAGE

Local storage is another common way to run the private registry. If you prefer this option or don't have an Amazon account you can store the registry files on a local mount point, but we recommend using a NFS or NAS based mount point so you can scale the reads, writes, and capacity requirements as you continue to store new images. Using a network-based storage will also allow you to scale the web portions of your registry. The size requirements can get out of control fast so make sure to plan accordingly.

Load balancing the registry

A single Docker repository server might not be enough over time due to the network bandwidth requirements. Docker was smart and decided to provide a plugable storage driver architecture in the private registry. This enables you to store the images within a network based scalable file store (such as S3) so you can load balance web portion of repository. This enables companies to put the registry images behind a load balancer of your choice. By putting your repository behind a web load balancer you can easily start to scale the network bandwidth and reduce a single point of failure of a single repositiory server.

Maintenance

Over time you'll no longer use older images and tags. Currently there is no auto pruning options in the Docker repository so operational best practices recommend you clean out unused tags and images on some regular interval. You are not able to delete images or tags via the **Docker API** (as of 1.6), so this will require sshing into the machine and cleaning out old images through the Docker CLI commands.

You will also need to keep in mind the Docker repository is an extremly active project so you'll need to keep an eye out for upgrades and new features. Please check the lastest documentation for information on the proper ways to upgrade the Docker repository.

Making your private repository secure

The Docker private registry is simple to secure on the network since you only need to allow port 5000. However, it is easy to reconfigure the registry on port 80 or 443 since it is a standard web server. We recommend following your companies best practice for configuring your firewalls to block access to only the ports needed.

SSL

You can protect your images from man in the middle attacks with the use of SSL certificates. There are multiple ways to achieve having your registry secure the transmission with an SSL certificate. You can use the built in nginx server, configure TLS, or another popular option is to off-load the certificate on a load balancer. If you're using AWS for instance, you can configure an SSL certificate on an Elastic Load Balancer (ELB) and then transmit in http to registry web servers. Installing an SSL certificate is trivial on the Docker registry. Here is a **link** to get you started.

Authentication

Authentication is important to protect your registry from getting invalid or insecure images uploaded. It also secures the intellectual property of your source code or information provided in your images. For high secure environments this will be an important factor to consider. There are currently two authentication providers provided in the private registry, silly and token.

Token based authentication is really the only secure option to choose from. Token based authentication is a well established authentication paradigm with a high degree of security that many companies use today. Silly authentication is as insecure as its name implies. Silly authentication just checks for the existence of the Authorization header in the HTTP request. If you don't supply a header it will still authenticate.

Save/Load

There is one other way to store images by using the save/load feature that Docker implemented in their pre 1.0 version. Some companies adopted the **Dogestry** pattern due the instability of the private registry in the early days and have stayed with it. Today this is the least preferred way to move images around, but it is certainly possible to use if it fits your enviornment. You can use the save/load feature by using the built in Docker commands. An example is to do the following:

- docker build redis
- docker save redis > /tmp/redis_docker_save.tar
- copy the image to your remote server (or use it locally)
- docker load < /tmp/redis_docker_save.tar

By using the save/load commands you can have as much flexiblity as needed. It is possible to save the image as a tar and upload it to your repository or a network share to be used centrally. One thing to note, you can use the Export command in Docker. It is slightly different than Save. The Export command flattens the image, which means it loses the history and meta-data. It can keep the image smaller in size. Keep this in mind if you use the save/load methods when moving images around.

Minimizing your image sizes

Docker images can get large in size depending on the dependencies used to build the image. Let's say for instance you use Ubuntu as a base image and then you use apt-get to update any libraries and then install a package such as nginx. Apt-get will install a bunch of cached libraries and dependencies that are unnecessary to use after the container is built. A common pattern is to remove the cached files and directories minimizing your image. If you find yourself needing to minimize Docker images, you can use a great community project called **docker-squash**. Here is a quick example of how to use it.

```
- docker save &lt;image id&gt; | sudo docker-squash -t newtag |
docker load
```

We ran docker-squash against a public image in mesosphere/marathon image and was able to reduce the size by 11%. The initial size when we pulled down the image was 831.7MB. After running docker-squash on the image we created a new one of 736.2MB. The space improvements can add up over time and also save network bandwidth if you're looking to improve the storage and performance in your repository.

If you would like to read more on compacting images we also suggest this **blog post**.

Other Image repository solutions

As the Docker echo system grows you should also keep in mind other image repositories as you explore new environments to run Docker.

- **Artifactory**
- **Quay**
- **Dogestry** Updated by New Relic.
- **Google Container Repository**
- **Docker on Azure**

In the next chapter we cover how to use CI/CD systems with Docker images.

9 CI/CD

Now that you are familiar with building and storing containers, we wanted to briefly talk about using CI/CD systems with Docker images. Many companies today have adopted Dev/Ops practices and use an automatic build system such as Jenkins, TravisCI, or Teamcity to automatically build their code. When code is automatically built it's simple to add the code to a container within the Docker build process. If you're thinking of running Docker in production we highly suggest building your images with an automatic continuous integration and continuous deployment (CI/CD) build system. The ease of Docker's build and push, while combined with a CI/CD system, might be one of the most powerful Dev/Ops service deployment methods to date.

Docker in its essence is an application delivery framework. Even Docker's website motto is build, ship, and run. If you are able to automate the build and ship pieces of your code delivery process, then you're most likely able to deliver products to your customers faster. Developers like to ship code early and often. Companies, however, ship products. Docker allows the ability to package an entire product into a container providing the ability to not just ship code early and often but as an entire product. Over the next few years we'll see more and more products delivered as a container image instead of a downloaded msi, jar, or zip file. If you are a company that needs to deliver code faster to customers via a SaaS, or you're delivering your product as a container, you'll want to start building Docker images with your build system.

If your build system already builds your code and artifacts, the next step is to get your build system building Docker images. Let's imagine a web container that runs Jetty and uses a jar from Java code to run its application. A CI/CD system will build the code, package it up, then deploy the final jar in an artifact location such as Artifactory, a file system, AWS S3, or store it within its own internal system. At this point, when you build a Docker image you'll just need to add the JAR file by using `ADD code.jar /jetty/bin/code.jar` within the Dockerfile. When the container is running, Jetty just needs to be configured to use that JAR in /jetty/bin/code.jar to load your application. If you want to look at an example here is one from **Rally Soft**.

CICD systems are not only great for building and packaging code, but are also fantastic at building Docker images. A Docker image just requires the Dockerfile (the code) and a build command to compile. Once it's built you just need to ship the package to a reposito-

ry. A CI/CD system will automate the entire process for you on each commit to Github if it's setup to do so. This will allow your developer's or infrastructure operation's team to deploy new components, infrastructure, or applications in an automated deployment system.

Building Docker images with a CI/CD system requires talking to a Docker daemon. A simple approach is to just install Docker on your build agents, which can then build and then ship the images. Another option you might have heard about is a term called Docker in Docker (aka **DIND**. You can read more about it here in this **blog post**. But in short, DIND allows you to send build commands to another Docker daemon to do builds for you. Regardless of the option you choose, you'll need an automated way to build Docker images and add your code to it during the build process.

Build systems typically work in steps. Let's break down a couple of possible builds with using Docker.

Building a Docker image

1. Pull the latest Dockerfile code from Github.com
2. Run the `docker build -t repo.com/image .` to build the image
3. Ship the image to the repository with `docker push repo.com/image`

Building a Docker image with code

1. Pull the latest Java code from Github.com
2. Compile and test the Java code with Maven and set the output as code.jar
3. Pull the latest Dockerfile code from Github.com (The Dockerfile has the ADD `code.jar /jetty/bin/code.jar` command)
4. Run the `docker build -t repo.com/image .` to build the image
5. Ship the image to the repository with `docker push repo.com/image`

Building a Docker image with code and integration tests

1. Pull the latest Java code from Github.com
2. Compile and test the Java code with Maven and set the output as code.jar
3. Pull the latest Dockerfile code from Github.com (The Dockerfile has the ADD `code.jar /jetty/bin/code.jar` command)
4. Run the `docker build -t repo.com/image .` to build the image
5. Spin up a test Docker infrastructure
6. Run full end to end integration tests
7. Spin down the test Docker infrastructure
8. Ship the image to the repository with `docker push repo.com/image`

These examples are not that complex, but most of the time it doesn't need to be. Docker images are super easy to build and ship. The hardest part is understanding Docker and getting your build system to automatically run the Docker commands. Automating this

process will only enhance your engineering outcomes. Let's cover a couple more topics around using an automated CI/CD system to build your Docker images.

Let everyone just build and push containers!

So you've got your CI/CD system automatically building images and you're now running Docker in production. You've probably at this point realized that your infrastructure can run almost any container you throw at it. Why not just have developers build and push a Docker image to production after they write new code? Or just put that in the build system and automatically deploy containers to production? This sounds like a fantastic world in theory but it's not in reality. An enterprise environment will quickly get out of control, especially if you unleash every developer to build their own web server container with whatever code in a Docker image and then hand that image to the operations team to run. The operations team will throw up their hands and walk out of the building. They will quickly realize that they have lost control and have no idea what's in the containers. If they do stick around they might start asking some questions like: Does it run Jetty? Does it run Nginx? Does it run Apache? Is the version of the web server the latest? Is it configured with the latest security hardening best practices? Does it have SSH in it? How does it log? Does it use a standard logging format? On and On the questions will pursue. Letting everyone build and push containers is an idea that some teams get, but it often quickly turns into a terrible practice. We highly suggest taking the higher route and working on standards sooner rather than later.

Here are some standards we've come to recognize teams using Docker in production.

Build all images with a build system

If operations or developer teams consistently push images from their workstation, some information or best practices will be disregarded. Consistency is key to scaling environments and spreading knowledge. Standards should be well thought out on how you're going to build, add to, use, and run Docker containers. Building all of your Docker images in a central system provides documentation. It's just like infrastructure as code but for building, compiling, and packaging your images. It also allows you to start standardizing on where containers get pushed, what images to base from, code verifications, and much much more.

Over time your security teams will find out what you're doing and want to know more. New approaches to security have been involving security teams more in the build process. So give your security team access and let them review what's being built. They can also leverage tools to test for vulnerable packages, insecure configurations, and even bad practices such as including secrets in code.

Suggest or don't allow the use of non standard practices

Does your Docker image run Apache, Jetty, Nginx, or all three? Does one Docker image run Gwizard and another run Dropwizard? You should be asking these questions constantly if you've not created an engineering best practice around containers. At some point you or your operations team will be called into to debug an image thats been long forgotten about. Having a consistent standard in services or packages you use within your images will go a long way in the success of your engineering practices. If your engineering team prefers to run Nginx with Jetty, then use those services to deliver your web services. If your team prefers to use Dropwizard over Gwizard then use those packages. The faster your team standardizes on what you put in your images the better your success will be.

Use a standard base image

If you're following the first and second standard you've probably come to the realization that a default base Docker image is a good idea. Let's say your team uses Python 2.6 as a code language. If you have a new developer on the team pull the latest base image of Python, but it was 3.0, you'll be bound to run into issues. Setting a practice of inheriting your images from a standard base image helps a lot. Some teams have started to have their operations team build base images for them that they then just inherit off of. An example is if your operations team creates a base Ubuntu image with the appropriate configuration for logging, setup proper security, and installed the correct version of Python 2.6. The only thing you're development team needs to do is to use the base image using FROM and ADD their code into the image during build, like this example:

```
FROM company.com/python_base:2015_02

ADD code.py /code.py

CMD [ "python", "./code.py" ]
```

This drastically simplifies the image creation process along with leaving a lot of the details to whomever owns and maintains the base image. Providing a standard base image will allow your operations team to always know what's running in the containers (most of the time), it simplifies the build process, allows developers to focus on code, and helps to scale Docker images on your infrastructure since it creates consistency.

Integration testing with Docker

Docker allows you not to only package your application in an image and ship it, but also full pieces of infrastructure. Let's say you run integration tests within your production

builds and you have a static environment with static data. You'll most likely build your code and then run the integration tests against the static environment. In a world with Docker you can turn that static environment into a dynamic one by integrating Docker images in your test infrastructure. Docker is very fast to spin up and down along with the ease of making golden images could power your test infrastructure on demand. When a build is finished you can run the integration infrastructure on the same agent or another host system (calling the Docker API) and then power it back down when finished. This may even save the company money to keep the static infrastructure up.

To this date we've not seen a plethora of innovation in the CI/CD test space with Docker, but there are new projects starting to emerge. We've seen Docker run Selenium browser tests inside of images and even provide full end-to-end testing for companies, but there are no standards we've seen yet. We look forward to seeing more come out of this space soon. If you're curious about Docker and Selenium check out this github project **Docker Selenium** .

Summary

Dev/Ops is all about a culture of delivering code and infrastructure as one team. When your engineering team practices Dev/Ops and is able to use a single system to configure and deploy your Docker images to your infrastructure you've begun to enter nirvana. Recently we've seen Jenkins go all in on Docker, along with Docker at DockerCon 15, and even the Microsoft Visual Studio team has shown off the power of building and deploying applications and infrastructure with Docker images. We hope to see more and more companies starting to use Docker within their CI/CD environments and build more best practices.

Configuration management is important with Docker and is covered in the next chapter.

10 Configuration Management

Configuration management (CM) has become a widespread tool in the infrastructure toolshed over the last decade, with platforms like Chef, Puppet, and cfengine becoming common place in most server rooms. CM tooling became ever more important as infrastructure became more dynamic with the adoption of cloud infrastructure providers like Amazon or Rackspace. The goal of CM tools is to standardize and automate the tasks of configuring new servers and update the configuration of existing servers. CM is used for tasks that range from adding new users to creating complex computer clusters that run Big Data services. These platforms need to deal with a large variety of operating systems' families and versions, services and use cases. Over time these tools have gained more criticality in the datacenter, and the kinds of automation builds with them have become more and more complex. In part, the lure of containers has been fueled by their promise to remove the complexity from configuration management.

Configuration Management versus Containers

Compared to bare iron servers or Virtual Machines (VMs), the configuration management of containerized infrastructure is much simpler. When looking at configuration management from the perspective of how often configuration changes take place, you realize that there are three different layers of configuration changes, each with different levels of entropy.

At the top level there is the number of hosts that are part of the system, along with their capacity, configuration, and how they relate to each other. This configurtion seldom changes except for hardware/VM failures, and usually the change is one of replacing broken servers with functioning ones.

The second level is the configuration of hosts themselves. This includes installing packages, applying patches, and writing configuration files. This changes more often than the previous layer.

Finally, the level that changes more often is the one related to the applications that are run on the host. This includes bug fixes, optimizations, new features, new versions, new configuration, and other changes. This is where most of the entropy of modern infrastructure exists.

The use of containers makes this difference in rate of change between these three layers more extreme. Whereas all hosts that run containers look the same and they rarely need to change, the containers that are run on each host change much more often.

Since a configuration manager's forté is setting up hosts, these mangers have little work to do in this new containerized world. Their role thus changes from configuring every aspect of a system to only configuring the infrastructure where these services will run, which ranges from setting up Docker hosts to building Mesos clusters.

This work of setting up the container infrastructure is so much more static, generic and repetitive that you must ponder whether using a traditional configuration manager is even a cost worth bearing. These configuration managers are complex and are built to do much more than what is required now because of Docker, and with this complexity comes a learning curve, a cost of operation and a cost of customization.

Also, traditional configuration management has become less mission-critical because most of the configuration changes needed are happening at the Docker container level. If there are no base infrastructure changes, you don't need a configuration manager to push new versions of the services or to push the new configuration of those services.

Configuration Management for Containers

Even though Configuration Management (CM) has become less central in containerized environments, it is still relevant, and chances are you are not going to work in a fully containerized environment that still requires CM, so integrating containers into it just makes sense.

There are three areas where CM help Docker users:

1. Set up and maintain Docker hosts. This covers from provisioning new hardware with the base operating system to installing and configuring the Docker process, and keeping these hosts up-to-date with security patches. This allows users to quickly setup new hosts to increase capacity and also manage a fleet of Docker hosts' containers in a centralized manner.
2. Management of Docker containers and Docker images. This covers everything from the lifecycle of images (creating them, pushing them, etc.) to running and managing containers that run these images.
3. Building images. Although Docker provides for a simple way of building images using Dockerfiles, there already exists a large selection of CM formulas to install and configure most of the software you would run in a container. You can use the mechanisms that install these kinds of software on VMs to install it on a Docker container, bypassing Dockerfiles.

Of the three functionalities provided by CM tooling, the one that is more pervasive among Docker users is the first one--setting up Docker hosts. The two others might not be compelling enough for users who don't already have an existing CM set up, since they offer little over what Docker already offers.

In this section we'll explore how some of the most used CM systems that support Docker.

Chef

Chef provides Docker the host installation and management of images and containers via the **docker-chef cookbook**. This cookbook is quite complete in its support for the different Docker host configuration options (storage drivers, daemon startup options, etc.), and also in its support of host operating systems.

Installation of Docker in the host is as easy as adding the following to your cookbook:

```
include_recipe 'docker'
```

The management of images and containers is equally straightforward and nicely maps to docker commands, such as pulling an image is done via a resource like this:

```
docker_image 'nginx'
```

This will install the latest nginx image. If instead you want to pull a different version of the image you do so with:

```
docker_image 'nginx' do
  tag '1.9.1'
end
```

Running containers off of images is not any more difficult. For example, this is how you'd start a container from the previous image, opening port 80 and mounting a local directory to `/www`:

```
docker_container 'nginx' do
  detach true
  port '80:80'
  volume '/mnt/docker:/www'
end
```

Finally, you can operate containers in the same way you would do with Docker. This, for example, would be run to commit the current container as a new image named `my-company/nginx:my-new-version`:

```
docker_container 'nginx' do
  repository 'my-company'
  tag 'my-new-version'
  action :commit
end
```

There is much more functionality in this cookbook that maps to existing Docker functionality like push, cp, export, and more, but the gist is always the same and it maps neatly the functionality that the Docker binary offers.

Chef also offers functionality to build images using their standard configuration infrastructure. What they provide is a **Chef Container**, a Docker image that has the Chef agent installed and configures images to run a full operating system with *runit* as the process supervisor. Containers running this image will connect to a Chef server and configure themselves with the predetermined cookbooks. This offers a nice transition from the VM/bare metal world to Docker, allowing you to use the same cookbooks across VMs, containers and bare metal servers.

There are many other cookbooks that the Chef community provides to automate the configuration of other pieces in the Docker ecosystem, including service discover (etcd, consul, etc.), schedulers and resource managers like Mesos and Kubernetes. They even provide a docker-api gem to allow Chef recipes to talk directly to the Docker process via its API.

In summary, if you are already a Chef user, you probably want to explore what Chef offersin the realm of containers. Chef can offer a straightforward migration path from your current infrastructure to containers, and once in Docker world, the use of Chef will depend mostly on environmental factors (like having other users in the company) and how deeply you embrace the Docker philosophy of moving away from VMs into single process containers.

Ansible

Similarly to Chef, Ansible provides support for Docker that ranges from host configuration to image and container management.

Ansible's approach to configuration management fits well with Docker in that they're both quite simple and direct. Whereas Chef and Puppet have a master/server architecture (offering standalone side-projects) where an agent running inside the machine (and container) pulls changes in the configuration needed, Ansible offers a much more direct approach where configuration is pushed from a remote machine on to the target machine via SSH. This approach fits a bit better with Docker than the agent model.

Installing Docker on a host with Ansible is straightforward. Although Ansible does not provide an official playbook for this matter, you can find some inspiration from the **docker.ubuntu** role from Paul Durivage that installs Docker on Ubuntu. For example, to install Docker so it listens on port 7890 you add this to your Ansible playbook:

```
- name: Install Docker on Port
  hosts: all
  roles:
    - role: angstwad.docker_ubuntu
      docker_opts: "-H tcp://0.0.0.0:7890"
      kernel_pkg_state: present
```

Ansible does provide official support for managing docker hosts: the [Docker module(**http://docs.ansible.com/docker_module.html**)]. With this module you can manage images and create/start/stop/delete containers, which is pretty much everything you can do directly with Docker. When it comes to containers, you can also specify restart policies that will determine what Ansible needs to do when a container fails. Ansible's Docker module also provides nice management facilities when you have multiple containers. You can, for example, address all of the containers that run the same image and restart them.

You can add Docker operations directly to your playbook. For example, with the following in your playbook, Ansible will ensure the image `myimage:1.2.3` is downloaded and a container named `mycontainer` is created, with a volume in `/usr/data` and running the downloaded image:

```
- name: data container
  docker:
    name: mycontainer
    image: myimage:1.2.3
    state: present
    volumes:
    - /usr/data
    command: myservice --myparam myvalue
    state: started
    expose:
    - 1234
```

The rest of the options should be self-explanatory at this point. The container will run the command `my-service` at startup, with the parameters `--myparam myvalue`. The container will also expose port `1234`.

Another example would be to restart all existing containers in a host that is running the exact same image. For example, to restart all containers running `myimage:1.2.3` you'd add the following to your playbook:

```
- name: restart myimage:1.2.3
  docker:
    image: myimage:1.2.3
    state: restarted
```

Another way in which Ansible can help with your Docker infrastructure is by building images from playbooks. For this to work, you need to create a Dockerfile that sets up Ansible locally and copies the playbook to run, and then run it:

```
FROM ubuntu
# install ansible
RUN apt-get -y update
RUN apt-get install -y python-yaml python-jinja2 git
RUN git clone http://github.com/ansible/ansible.git /usr/lib/
ansible
```

```
WORKDIR /usr/lib/ansible
ENV PATH /usr/lib/ansible/bin:/sbin:/usr/sbin:/usr/bin
ENV ANSIBLE_LIBRARY /usr/lib/ansible/library
ENV PYTHONPATH /usr/lib/ansible/lib:$PYTHON_PATH

# Download copy the playbooks and hosts
ADD playbooks /usr/lib/ansible-playbooks
ADD inventory /etc/ansible/hosts
WORKDIR /usr/lib/ansible-playbooks

# Run the playbook to let Ansible configure the image
RUN ansible-playbook my-playbook.yml -c local

# Other docker config
EXPOSE 22 4000
ENTRYPOINT [“myservice”]
```

With the Dockerfile above you can push all of the configuration affairs into the playbooks that will be copied into the image, and Ansible will take care of everything else. This is very convenient when you already have those playbooks and have no desire to redo the configuration automation work directly in the Dockerfile.

The official Docker module also offers functionality for image management, like pushing, pulling, the removal and the download of images. Notice though that most of these functions are equally easy to achieve by having Ansible run the Docker CLI directly.

Salt Stack

As of version `2014.7.0` Salt Stack has complete support for Docker, ranging for support for the main Docker operations to getting information from Docker into Salt Mine.

Salt provides the `DOCKERIO` module to manage Docker containers. With it, you define the desired Docker states and the Salt Minions will take care of contacting Docker to achieve the desired state. For example, if you want a state that represents a container named `mycontainer` that runs the image `myorg/myimage:1.2.3`, you can define the desired state as the following:

```
my_service:
  docker.running:
    - container: mycontainer
    - image: myorg/myimage:1.2.3
    - port_bindings:
        "5000/tcp":
            HostIp: ""
            HostPort: "5000"
```

The rest of the Docker operations are similar. As in the case of Ansible, operating Docker via its CLI with Salt Stack is fairly easy too. For example, following is roughly equivalent to the previous state:

```
my_service:
  cmd.run:
    - name: docker run -p5000 --name mycontainer
myorg/myimage:1.2.3
```

Puppet

You can install and manage Docker hosts, images and containers with Puppet through the very complete **Docker Module** provided by Gareth Rushgrove. This module makes operating Docker quite simple if you are already running Puppet.

Installing Docker itself is fairly easy and the installation works with Ubuntu 12.04 and 14.04 and Centos 6.6 and 7.0, although it might work unmodified on other Debian- and RHEL-based distros. This is how you go about installing the latest Docker on a host:

```
include 'docker'
class { 'docker':
  version => 'latest',
}
```

The installation class has many options, for example you can change the port that Docker will bind, or even where the socket gets created:

```
class { 'docker':
  version => 'latest',
  tcp_bind    => 'tcp://127.0.0.1:4243',
  socket_bind => 'unix:///var/run/docker.sock',
}
```

Once Docker is installed, managing its images and containers is also quite straightforward. For example, pulling our image is done with:

```
docker::image { 'myorg/myimage':
  image_tag => '1.2.3'
}
```

and removing it is done with:

```
docker::image { 'myorg/myimage':
  ensure    => 'absent',
  image_tag => '1.2.3'
}
```

Running a container is not much more difficult:

```
docker::run { 'myservice':
  image   => 'myorg/myimage:1.2.3',
  command => 'myservice --myparam myvalue',
}
```

`docker::run` has many more configuration options that map nicely to standard Docker run options, like exposed ports, environment variables, restart policies, etc. It also provides some options that transcend Docker itself, like container dependencies. These dependencies are coded into initd or systemd.

Finally, a handy feature in this module provides the ability to run `exec` commands inside running containers:

```
docker::exec { 'myservice-ls':
  detach    => true,
  container => 'myservice',
  command   => 'ls',
  tty       => true,
}
```

Summary

Current CM tooling offers some level of Docker support. Whether this support is sufficient and whether the cost of using these tools compensates the benefits is up to your own circumstances. Certainly these tools offer some transition for companies that have already made the investment in them for managing VMs.

Docker storage drivers provide Docker with efficiency, and they are covered in the next chapter.

11 Docker Storage Drivers

One of the biggest benefits of using Docker is the speed of starting new containers from an existing image. Historically, **LXC** containers, out of which Docker has evolved, would make a full physical copy of the image root filesystem into a separate path on the host for each newly created container. This was obviously quite inefficient. The disk space usage grew with each newly created container and container start time would take several seconds depending on the amount of data that needed to be copied from one path to another. Docker addressed both of these issues by using **layered images**.

On a high level, image layers are simply filesystem trees, which can be mounted and modified when needed. New image layers can be created either from scratch or from an existing layer also called a **parent layer**. Image layer created from an existing layer is an exact copy of it - both the parent and the new layer can be addressed by Docker via the same unique name. Once you've modified the new layer, a new unique name is generated and assigned to it. From this moment on, the parent layer stays untouched and the further modifications are made only to the new layer. If this reminds you of the well known **Copy on write** (CoW) mechanism then you would be spot on!

To implement the image layering, Docker relies on various CoW filesystems, some of which are included in the mainline Linux kernel. Instead of making a full copy of a parent image, Docker only keeps track of the changes (called **diffs**) between the parent and the layers created from it. This provides for quite a huge disk space savings. The image filesystems grow only by the size of the diffs between the image layers.

Docker applies the similar concepts to containers. Every container has two layers:

- **init layer** - layer based on the parent image layer; it contains few files that must exist in every Docker container: `/etc/hosts`, `/etc/resolv.conf` etc.
- **container filesystem layer** - layer based on the init layer; it contains the data stored in the container.

Docker exposes the image layers via its remote API, which provides for some handy features like container versioning and image tagging. If you want to save a layer from a container you simply invoke `docker commit container_id` and Docker finds all of the changes applied from the **init layer** all the way to the container layer, and creates a new layer on top of the parent layer. You can tag the committed layers and either build new

images or start new containers from them. By applying the same CoW concept to the container filesystems, Docker offers super fast container start times.

A picture is worth a thousand words - if you are interested in evaluating image layers of any Docker images stored in Docker Hub you can use the fantastic **Image Layers** tool created by **Centurylink Labs**, which allows you to inspect available image layers manually and provides even more functionality.

Now that we have covered the basics of how Docker deals with image and container filesystems, we will evaluate all of the storage driver options provided by Docker on the following lines. We will dive deep into some of the core concepts to give you a better understanding and show a few practical examples so you can follow them by running the commands listed in each of them directly on your Docker host.

Docker provides quite a few storage drivers out of the box. All you need to do is to pick one of them. Once you have decided which storage driver to use, you need to tell it to the Docker daemon by passing a `--storage-driver` command line flag via the `DOCKER_OPTS` environment variable. As always, you must restart the daemon in order to pick up the new configuration. We will start our exploration of the storage drivers by looking at the driver that is enabled in Docker by default: `aufs`.

AUFS

As mentioned earlier `aufs` is the default storage driver provided by Docker. The reason for this is **partly** because the Docker team were using it to run containers at **dotCloud**, so they have acquired a solid knowledge and operational experience running it in production setup.

As the name [suspiciously] suggests, `aufs` uses the **AUFS** filesystem for the image and container storage. AUFS works by "stacking" multiple filesystem layers called **branches** on top of each other and making them available to the user via a single mount point as one filesystem. Each branch is a simple directory containing regular files and metadata. The topmost branch is **the only** read-write layer. AUFS relies on metadata to look up files across all of the stacked layers. The lookup always starts at the top and if read-write operation is required, the file is copied to the topmost layer. This can take a significant amount of time when the file is too big.

So much for the theory. Let's have a look at a practical example to show how Docker uses the AUFS storage driver. First we will make sure that `aufs` is indeed in use by inspecting the Docker setup:

```
# sudo docker info
Containers: 10
Images: 60
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
```

```
 Dirs: 80
Execution Driver: native-0.2
Kernel Version: 3.13.0-40-generic
Operating System: Ubuntu 14.04.1 LTS
CPUs: 1
Total Memory: 490 MiB
Name: docker-hacks
ID: DK4P:GBM6:NWWP:VOWT:PNDF:A66E:B4FZ:XMXA:LSNB:JLGB:TUOL:J3IH
```

As you can see, the base image directory of AUFS driver is `/var/lib/docker/aufs`. Let's explore its contents:

```
# ls -l /var/lib/docker/aufs/
total 36
drwxr-xr-x 82 root root 12288 Apr  6 15:29 diff
drwxr-xr-x  2 root root 12288 Apr  6 15:29 layers
drwxr-xr-x 82 root root 12288 Apr  6 15:29 mnt
```

If you haven't created any containers yet, all of the above directories will be empty. Like the name suggests, the `mnt` subdirectory contains the mount points for the filesystems of the containers. These are only mounted when the container is running. So let's create a new container and see this in practice. We will run the `top` utility inside the new Docker container to keep it running until we stop it:

```
# docker run -d busybox top
Unable to find image 'busybox:latest' locally
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Im-
portant: image verification is a tech preview feature and
should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
f534838e081ea8c3fc6c76aa55a719629dccbf7d628535a88be0b3996574fa47
```

As you can see from the above output, the busybox image consists of five image layers that translate into 5 AUFS branches. AUFS branches are stored in the `diff` directory and you can easily verify that each subdirectory of the `diff` directory corresponds to each of the above image layers:

```
# ls -1 /var/lib/docker/aufs/diff/
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b
ea13149945cb6b1e746bf28032f02e9b5a793523481a0a18645fc77ad53c4ea2
4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125
f534838e081ea8c3fc6c76aa55a719629dccbf7d628535a88be0b3996574fa47
```

```
f534838e081ea8c3fc6c76aa55a719629dccbf7d628535a88be0b3996574fa47-
init
```

You can also see that the **init layer** has been created from the topmost layer when we started the container as discussed at the beginning of this chapter. Now that the container is running, its filesystem should be mounted, so let's verify that:

```
# grep f534838e081e /proc/mounts
/var/lib/docker/aufs/mnt/
f534838e081ea8c3fc6c76aa55a719629dccbf7d628535a88be0b3996574fa47
aufs rw,relatime,si=fa8a65c73692f82b 0 0
```

The mount point of the running container filesystem maps to `/var/lib/docker/aufs/mnt/container_id` and that is mounted in read-write mode. Let's modify the filesystem of the running container by creating a simple file in it (`/etc/test`) and commit it afterwards:

```
# docker exec -it f534838e081e touch /etc/test
# docker commit f534838e081e
4ff22ae4060997f14703b49edd8dc1938438f1ce73070349a4d4413d16a284e2
```

The above should create a new image layer containing the new file and storing the diff in a particular diff directory. This is easy to verify by listing the `diff` subdirectories:

```
# find /var/lib/docker/aufs/diff/
4ff22ae4060997f14703b49edd8dc1938438f1ce73070349a4d4413d16a284e2/
-type f

/var/lib/docker/aufs/diff/
4ff22ae4060997f14703b49edd8dc1938438f1ce73070349a4d4413d16a284e2/
etc/test
```

You can now start a new container from the above created image layer that contains the `/etc/test` file:

```
# docker run -d
4ff22ae4060997f14703b49edd8dc1938438f1ce73070349a4d4413d16a284e2
top
9ce0bef93b3ac8c3d37118c0cff08ea698c66c153d78e0d8ab040edd34bc0ed9
# docker ps -q
9ce0bef93b3a
f534838e081e
```

Verify that the file is present in the newly created container:

```
# docker exec -it 9ce0bef93b3a ls -l /etc/test
-rw-r--r--    1 root     root             0 Apr  7 00:27 /etc/
test
```

Let's have a look what happens when we delete a file and commit the container:

```
# docker exec -it 9ce0bef93b3a rm /etc/test
# docker commit 9ce0bef93b3a
e3b7c789792da957c4785190a5044a773c972717f6c2ba555a579ee68f4a4472
```

When you delete a file, AUFS creates a so called **"whiteout"** file, which is basically a renamed file with a *".wh."* prefix. This is how AUFS **marks** the files as deleted. This is very easy to verify by inspecting the contents of this particular image layer directory:

```
ls -a /var/lib/docker/aufs/diff/
e3b7c789792da957c4785190a5044a773c972717f6c2ba555a579ee68f4a4472/
etc/
.  ..  .wh.test
```

The hidden file is physically present on the host filesystem, but when you start a new container from the above created layer, AUFS magic kicks in and this file will not be present in the running container filesystem:

```
# docker run --rm -it
e3b7c789792da957c4785190a5044a773c972717f6c2ba555a579ee68f4a4472
test -f /etc/.wh.test || echo "File does not exist"
File does not exist
```

This concludes our tour of the `aufs` storage driver. We looked at how Docker uses some of the features provided by the AUFS filesystem for creating containers and showed some practical examples. We will finish this chapter by a short summary and move on to discuss another storage driver.

AUFS mounts are very fast and they provide for the very fast creation of new containers. They also offer native read/write speeds. This makes it a decent and battle tested option to run your containers. AUFS performance can suffer when used in scenarios that require writing big files, so using `aufs` storage driver to store database files is probably not a great idea. Similarly, too many image layers can lead to long file lookup times, so try not to go overboard with the image layers in your containers.

While you can work around the previous deficiencies (for example by volume mounting the data directories and squashing image layers), the biggest problem with `aufs` storage driver is that the AUFS filesystem has never been included into the mainline Linux kernel and most likely never will be. Therefore it's not available on the majority of Linux distributions and its use often requires some hackery, which is not very convenient for users and makes applying updates to the Linux kernel even more painful than it already is. Even Ubuntu, who were shipping AUFS support in their kernel decided to **disable** it in version

12.04 and encourage users to migrate to **OverlayFS**, which has been included in Ubuntu kernel from version **11.10** and which is being actively developed. We will talk more about about OverlayFS later in this chapter. Let's discuss another storage driver that builds on quite mature Linux storage technology: `devicemapper`.

DeviceMapper

Devicemapper is an advanced storage framework provided by Linux kernel, which maps physical block devices into virtual block devices. It is the underlying technology used by **LVM2**, block level storage encryption, **multipathing** and many other linux storage tools. You can read more about devicemapper in the official Linux kernel **documentation**. In this chapter we are going to focus on how Docker uses devicemapper to manage the container and image storage.

`devicemapper` storage driver uses the devicemapper's **thin provisioning** module to implement the image layers. On a high level, thin provisioning (also known as **thinp**) provides a pool of raw physical storage [blocks] out of which you can create virtual block devices or virtual disks of arbitrary size. The "magic" trick with `thinp` is that these devices are not taking any disk space, or the raw storage blocks won't be marked as used until you actually start writing data into them.

Additionally, `thinp` is capable of creating volume snapshots. You can also create a copy of an existing volume and the new snapshot volume won't take any extra storage space. Again, the extra storage will only be allocated from the storage pool once the you start writing into it.

The thin provisioner uses two block devices:

- **data device** - a storage pool device, generally quite large
- **metadata device** - stores information about mappings between the storage blocks in created volumes (including snapshots) and storage pool

Copy on write in the `devicemapper` storage driver will happen on a block device level as opposed to a filesystem level, which is the case with `aufs` driver. When the Docker daemon starts, it automatically creates two block devices required by the thin provisioning to work:

- device for the storage pool
- device to hold the metadata

By default these devices are just **sparse files** backed by a loopback device. These files are 100GB and 2GB in size, but because they are sparse they don't actually use much disk space on the host.

A practical example will provide a better understanding of these concepts. First we need to tell Docker daemon to use the `devicemapper` storage driver by setting the `DOCK-`

ER_OPTS environment variable accordingly and then restarting the daemon. Once the daemon has restarted we can verify it is now using the `devicemapper` storage driver:

```
# docker info
Containers: 0
Images: 0
Storage Driver: devicemapper
 Pool Name: docker-253:1-143980-pool
 Pool Blocksize: 65.54 kB
 Backing Filesystem: extfs
 Data file: /dev/loop0
 Metadata file: /dev/loop1
 Data Space Used: 305.7 MB
 Data Space Total: 107.4 GB
 Metadata Space Used: 729.1 kB
 Metadata Space Total: 2.147 GB
 Udev Sync Supported: false
 Data loop file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/devicemapper/
metadata
 Library Version: 1.02.82-git (2013-10-04)
Execution Driver: native-0.2
Kernel Version: 3.13.0-40-generic
Operating System: Ubuntu 14.04.1 LTS
CPUs: 1
Total Memory: 490 MiB
Name: docker-book
ID: IZT7:TU36:TNKP:RELL:2Q2J:CA24:OK6Z:A5KZ:HP5Q:WBPG:X4UJ:WB6A
```

Let's explore the `/var/lib/docker/devicemapper/` directory where all of the devicemapper action happens:

```
# ls -alhs /var/lib/docker/devicemapper/devicemapper/
total 292M
4.0K drwx------ 2 root root 4.0K Apr  7 20:58 .
4.0K drwx------ 4 root root 4.0K Apr  7 20:58 ..
291M -rw------- 1 root root 100G Apr  7 20:58 data
752K -rw------- 1 root root 2.0G Apr  7 21:07 metadata
```

As you can see in the output above, `data` and `metadata` files created by the Docker daemon use very little disk space. We can confirm that these files are actually backed by loopback devices by running this command:

```
# lsblk
NAME                                  MAJ:MIN RM   SIZE RO TYPE
MOUNTPOINT
loop0                                   7:0    0   100G  0 loop
docker-253:1-143980-pool (dm-0)   252:0    0   100G  0 dm
```

```
  docker-253:1-143980-base (dm-1) 252:1    0    10G  0 dm
loop1                                7:1    0     2G  0 loop
docker-253:1-143980-pool (dm-0)   252:0    0   100G  0 dm
  docker-253:1-143980-base (dm-1) 252:1    0    10G  0 dm
```

In addition to creating the sparse files for thin provisioning, the Docker daemon also creates a **"base device"** on the thin pool containing an empty `ext4` filesystem. All of the new image layers are a snapshots of the base device, which means that every container and image gets its own block device. At any point in time you can create a snapshot of any existing image or container. By default the size of the base device is set to 10GB, which is the maximum size the containers and images can take, but due to thin provisioning the usage footprint is actually much smaller. You can easily verify the existence of the base device by running this command:

```
# dmsetup ls
docker-253:1-143980-base    (252:1)
docker-253:1-143980-pool    (252:0)
```

Let's create a simple container and do a bit of exploring like we did with the `aufs` driver:

```
# docker run -d busybox top
Unable to find image 'busybox:latest' locally
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Im-
portant: image verification is a tech preview feature and
should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01
```

If we list the `/var/lib/docker/devicemapper/mnt/` directory we will find a list of directories for each image layer:

```
# ls -1 /var/lib/docker/devicemapper/mnt/
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b
ea13149945cb6b1e746bf28032f02e9b5a793523481a0a18645fc77ad53c4ea2
4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01-
init
```

These directories serve as mount points for a particular devicemapper image layer. Unless a particular layer is mounted, for example when a container is running, you will find

the contents of its directory completely empty. You can easily verify this by checking the mount point of the running container:

```
# docker ps -q
f5a805967279
# grep f5a805967279 /proc/mounts
/dev/mapper/docker-253:1-143980-
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01 /
var/lib/docker/devicemapper/mnt/
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01
ext4 rw,relatime,discard,stripe=16,data=ordered 0 0
```

If you now list the content of the above directory you will find the contents of the container filesystem inside it. Once you stop the container, its backing block device will be unmounted and the contents of the mount point directory will be empty:

```
# docker stop f5a805967279
f5a805967279
# ls -l /var/lib/docker/devicemapper/mnt/
f5a805967279e0e07c597c0607afe9adb82514d6184f4fe4c24f064e1fda8c01
total 0
```

This means that when using the `devicemapper` storage driver, you don't have much visibility of the diffs between the image layers.

Furthermore, when using the `devicemapper` driver by default Docker will use sparse files for container and image backend storage. This has a significant impact on the performance. Each time a container modifies its filesystem, new storage blocks have to be allocated from the storage pool, which in the case of sparse files can take some time. Now imagine running hundreds of containers each of which is modifying its filesystem.

Luckily, Docker allows you to use real block devices for both the data and metadata devicemapper devices by using particular command line options passed to the Docker daemon via the `--storage-opt` command line switch. These are:

- `dm.datadev` for data block device
- `dm.metadatadev` for metadata block device

You should **ALWAYS** use real block devices for both the data and metadata when using `devicemapper` driver in production setups. This is especially important when you are running a lot of containers!

You can read about all available options provided by `devicemapper` driver on the following **link**.

TIP: *If you are thinking of switching from the `aufs` driver to the `devicemapper` driver, you must first save all images to separate tar files by running `docker save`, and once the `devicemapper` driver has been been picked up by Docker daemon you can easily load the saved images in by running `docker load`.*

As you can see, the `devicemapper` storage driver provides an interesting container storage alternative in Docker. If you are familiar with **LVM** and its rich toolset, you can easily manage the Docker storage in the familiar way. However, as always there are some caveats you must keep an eye on when you decide to use this driver:

- `devicemapper` driver requires at least basic operational knowledge of device-mapper subsystem
- changing any of the `devicemapper` options requires stopping the Docker daemon and wiping out contents of the `/var/lib/docker` directory
- by default the size of the container filesystem is set to 10GB as discussed earlier (you can change this via `dm.basesize` when the daemon starts)
- expanding the size of the running container that has outgrown the configured base device size is quite laborious
- you can't easily expand the size of the image - committing a container which is bigger than its base size is not easy

Now that you have a pretty good idea about how to use the `devicemapper` driver, it's time to move on to discuss the next option that builds on a filesystem that has been making waves (positive and negative) in the Linux community for quite a while: `btrfs`.

btrfs

It does not take a rocket scientist to guess that the `btrfs` storage driver uses the **btrfs filesystem** to perform the Docker's copy on the write image layer magic. In order to understand the `btrfs` storage driver let's take a short tour of some btrfs filesystem features and then have a look at some practical examples of how they are used in Docker.

`btrfs` is an overlay filesystem that has been in the mainline Linux kernel for quite some time, but somehow it still has not reached the quality or the maturity required from a production filesystem. It has been designed to compete with some of the features provided by Sun Microsystems' **ZFS filesystem**. Some notable features provided by btrfs include:

- snapshotting
- subvolumes
- adding or removing block devices without interruption
- transparent compression

`btrfs` stores data in **chunks**. A chunk is simply a piece of raw storage typically ~1GB in size which btrfs can use to put the actual data on. Chunks are spread across all of the underlying block devices. You can run out of chunks even though there is still free storage available. When this happens you will need to rebalance your filesystem, which will relocate data from empty or near-empty chunks to free up some disk space. This operation can be done without downtime.

So that was a little bit of theory. Now, let's look at the practical usage of the `btrfs` driver. In order to use this driver Docker requires the `/var/lib/docker` directory to be on a btrfs filesystem. We will not list the steps on how to do that - we will simply assume that you have prepared a btrfs partition and created the particular directories required by the `btrfs` driver on it:

```
# grep btrfs /proc/mounts
/dev/sdb1 /var/lib/docker btrfs rw,relatime,space_cache 0 0
```

Now you need to tell the Docker daemon to use the `btrfs` driver by modifying the `DOCKER_OPTS` environment. Once the daemon has been restarted you can verify that the driver is now ready to use:

```
# docker info
Containers: 0
Images: 0
Storage Driver: btrfs
Execution Driver: native-0.2
Kernel Version: 3.13.0-24-generic
Operating System: Ubuntu 14.04 LTS
CPUs: 1
Total Memory: 490.1 MiB
Name: docker
ID: NQJM:HHFZ:5636:VGNJ:ICQA:FK4U:6A7F:EUDC:VFQL:PJFF:MI7N:TX7L
WARNING: No swap limit support
```

You can inspect the btrfs filesystem by running the following commands, which will give you a quick overview of the filesystem usage:

```
# btrfs filesystem show /var/lib/docker
Label: none  uuid: 1d65647c-b920-4dc5-b2f4-de96f14fe5af
    Total devices 1 FS bytes used 14.63MiB
    devid    1 size 5.00GiB used 1.03GiB path /dev/sdb1

Btrfs v3.12

# btrfs filesystem df /var/lib/docker
Data, single: total=520.00MiB, used=14.51MiB
System, DUP: total=8.00MiB, used=16.00KiB
System, single: total=4.00MiB, used=0.00
Metadata, DUP: total=255.94MiB, used=112.00KiB
Metadata, single: total=8.00MiB, used=0.00
```

Docker takes advantage of the btrfs **subvolume** feature. You can read more about subvolumes in the following **link**. In the gist, the subvolume is quite a complex concept that can be thought of as POSIX file namespace that can be accessed via the top-level subvolume of the filesystem, or it can be mounted in its own right.

Every newly created Docker container is allocated a new btrfs subvolume, and starts out as a snapshot of the parent subvolume if there is any parent. The same applies for Docker images. Let's create a new container and explore these concepts further:

```
# docker run -d busybox top
Unable to find image 'busybox:latest' locally
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Im-
portant: image verification is a tech preview feature and
should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
86ab6d8602036cadb842d3a030adf2b05598ac0e178ada876da84489c7ebc612
```

You can easily verify that each layer has been allocated a new btrfs subvolume:

```
# btrfs subvolume list /var/lib/docker/
ID 258 gen 9 top level 5 path btrfs/subvolumes/
511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158
ID 259 gen 10 top level 5 path btrfs/subvolumes/
df7546f9f060a2268024c8a230d8639878585defcc1bc6f79d2728a13957871b
ID 260 gen 11 top level 5 path btrfs/subvolumes/
ea13149945cb6b1e746bf28032f02e9b5a793523481a0a18645fc77ad53c4ea2
ID 261 gen 12 top level 5 path btrfs/subvolumes/
4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125
ID 262 gen 13 top level 5 path btrfs/subvolumes/
86ab6d8602036cadb842d3a030adf2b05598ac0e178ada876da84489c7ebc612-
init
ID 263 gen 14 top level 5 path btrfs/subvolumes/
86ab6d8602036cadb842d3a030adf2b05598ac0e178ada876da84489c7ebc612
```

You can take a snapshot of any of the layers at any time. A snapshot is again simply a subvolume that shares its data (and metadata) with another subvolume. Since a snapshot is a subvolume, snapshots of snapshots are also possible. A practical example will make this clearer. We will make a change in the running container and commit it:

```
# docker exec -it 86ab6d860203 touch /etc/testfile
# docker commit 86ab6d860203
32cb186de0d0890c807873a3126e797964c0117ce814204bcbf7dc143c812a33
```

Committing the container has created a new btrfs subvolume in the `/var/lib/docker/btrfs/subvolumes/32cb186de0d0890c807873a3126e797964c0117ce814204bcbf7dc143c812a33` directory as expected. If you explore the contents of this subvolume, however, you will notice that it actually contains a full image filesystem as opposed to just a differential image.

This is due to btrfs having no concept of the read-write layer and no easy way to list the differences between snapshots. This might change in the future.

Let's now discuss what to keep an eye on when using the `btrfs` storage driver in Docker. As already mentioned the `btrfs` driver requires `/var/lib/docker` to be present on a btrfs filesystem. This has the advantage of keeping the rest of your operating system protected from the potential filesystem corruption. We would also recommend to put the `/var/lib/docker/vfs/` directory on some battle tested filesystem like `ext4` or `vfs`.

Th btrfs filesystem is very sensitive to low disk space. You must make sure you monitor the chunk usage and continuously rebalance the filesystem when needed. This can be a bit of a burden for the operators, especially when you are running a lot of containers and you need to keep a lot of container images on the host. On the other hand, you get an ability to easily expand the storage without service interruption.

The btrfs copy on write capability makes backing up containers and images super easy by taking advantage of its snapshotting feature. The copy on write is not very suitable for containers that create and modify a lot of small files, such as databases. This often leads to filesystem fragmentation and therefore requires frequent filesystem rebalancing. You might want to disable the copy on write for the directory or volume that is bind mounted into the containers with high IO activity to avoid these issues.

If you decide to remove docker btrfs subvolumes make sure you use `btrfs subvolume delete subvolume_directory` before you actually physically remove the underlying directory by running `rm -rf`, since you can cause filesystem corruption. This sometimes happens when you try to remove an image or destroy a container, so make sure you keep an eye on this, too.

In summary, the biggest strength and weakness of the `btrfs` storage driver is the btrfs filesystem itself. It requires a solid operational knowledge of using it and lacks good performance required from various production workloads. The btrfs also does not allow page cache sharing, which can lead to higher memory usage. These are the main reasons why **CoreOS** decided to drop btrfs from their operating system distribution **recently**.

overlay

The `overlay` storage driver is the latest storage driver introduced to Docker. It uses the **OverlayFS** filesystem to provide the copy on write for Docker image layering. Again, turning this driver on requires modifying the `DOCKER_OPTS` environment variable and restarting the Docker daemon. Before we dive into some practical examples, let's first discuss the OverlayFS filesystem.

OverlayFS is a union filesystem that was merged into the mainline Linux kernel 3.18. It combines two filesystem:

- **upper** - child filesystem
- **lower** - parent filesystem

OverlayFS introduces a concept of **workdir**, which is a directory that resides on the upper filesystem and basically serves for atomic copies between the upper and lower layers.

The lower filesystem can be any filesystem supported by the kernel (including overlayFS itself) and is generally **read-only**. In fact, there can be multiple lower filesystem layers which are then stacked or "overlay-ed" on top of each other. The upper filesystem is normally writable. Being a union filesystem, overlayFS also implements the familiar "white-out" files to mark files for removal, just like the AUFS filesystem that we discussed in the first chapter.

Overlaying mainly involves **merging** directories. Two files with the same path in each directory tree would appear to occupy the same directory in the overlay-ed filesystem. OverlayFS improves the lookup times in comparison to AUFS where the lookup times can take longer the more image layers your container has. Whenever a lookup is requested in a merged directory, it is performed in both merged directories and the combined result is **cached** in the entry belonging to the overlay filesystem. If both lookups find directories, both are stored and a merged directory is created, otherwise **only one is stored**: the upper if it exists, otherwise the lower.

So how does Docker use OverlayFS? The lower filesystem serves as the base image layer, and when you create new Docker containers, a new upper filesystem is created containing only the diffs between the two layers. This is exactly the same behavior we have seen when discussing AUFS. As expected, committing a container creates a new layer containing only the diffs between the base and a new layer.

So much for the theory. Let's examine an example. In order to follow it you will need a Linux kernel of version 3.18 or higher. If your kernel version is smaller, Docker will drop to the next available storage driver and won't use the `overlay`.

As always, you need to tell the Docker daemon to use the `overlay` storage driver. Note, the name of the driver is **overlay not overlayfs**:

```
# docker info
Containers: 0
Images: 0
Storage Driver: overlay
 Backing Filesystem: extfs
Execution Driver: native-0.2
Kernel Version: 3.18.0-031800-generic
Operating System: Ubuntu 14.04 LTS
CPUs: 1
Total Memory: 489.2 MiB
Name: docker
ID: NQJM:HHFZ:5636:VGNJ:ICQA:FK4U:6A7F:EUDC:VFQL:PJFF:MI7N:TX7L
WARNING: No swap limit support
```

Now let's create a new container and explore the `/var/lib/docker` directory afterward:

```
# docker run -d busybox top
Unable to find image 'busybox:latest' locally
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Im-
portant: image verification is a tech preview feature and
should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
eb9e1a68c70532ecd31e20d8ca4b7f598d2259d1ac8accaa02090f32ee0b95c1
```

All of the image layers are available under the `/var/lib/docker/overlay` directory as expected. Finding out the base image and the container layer is as easy as looking up the mounted container filesystem and inspecting the **lower** and **upper** directories as you can see here:

```
# grep eb9e1a68c705 /proc/mounts
overlay /var/lib/docker/overlay/
eb9e1a68c70532ecd31e20d8ca4b7f598d2259d1ac8ac-
caa02090f32ee0b95c1/merged overlay
rw,relatime,lowerdir=/var/lib/docker/overlay/
4986bf8c15363d1c5d15512d5266f8777bfba4974ac56e3270e7760f6f0a8125/
root,upperdir=/var/lib/docker/overlay/
eb9e1a68c70532ecd31e20d8ca4b7f598d2259d1ac8ac-
caa02090f32ee0b95c1/upper,workdir=/var/lib/docker/overlay/
eb9e1a68c70532ecd31e20d8ca4b7f598d2259d1ac8ac-
caa02090f32ee0b95c1/work 0 0
```

Just like we have tested the previous drivers, we will now create an empty file in the running container and commit it, which will trigger creating a new layer:

```
# docker exec -it eb9e1a68c705 touch /etc/testfile
# docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
eb9e1a68c705        busybox:latest      "top"               9
minutes ago       Up 9 minutes
cocky_bohr
# docker commit eb9e1a68c705
eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b
```

You can easily confirm that the new image layer has been created and that it contains the new file:

```
# ls -l /var/lib/docker/overlay/
eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b/
```

```
root/etc/testfile
-rw-r--r-- 1 root root 0 Apr  8 21:44 /var/lib/docker/overlay/
eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b/
root/etc/testfile
```

All of the above is what you would expect and it is pretty much the same as when the `aufs` driver was used. There are a few differences, though. There is no `diff` subdirectory. However, that doesn't mean we can't inspect the filesystem deltas - they are just "hiding" somewhere else. `overlay` driver creates three subdirectories for each container:

- `upper` - this is the read-write upper filesystem layer
- `work` - temporary directory for atomic copy
- `merged` - mount point for the running container

Additionally, the driver creates a file called **lower-id**, which contains the id of the parent layer whose "root" directory shall be used as the lower layer in the overlay - this file basically serves as a lookup id for the parent layer.

Let's start a new container from the earlier committed layer:

```
# docker run -d
eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b
007c9ca6bb483474f1677349a25c769ee7435f7b22473305f18cccb2fca21333
```

We can easily verify the parent container id by investigating the "lower-id" file:

```
# cat /var/lib/docker/overlay/
007c9ca6bb483474f1677349a25c769ee7435f7b22473305f18cccb2fca21333/
lower-id

eae6654d10c2c6467e975a80559dcc2dd57178baaea57dd8d347c950a46c404b
```

We are not going to spend more time exploring the `overlay` storage driver given how it is very similar to what we have seen when we tested the `aufs` driver. So why is there so much excitement surrounding the `overlay` driver? There are a couple of reasons for it:

- OverlayFS filesystem is available in the mainline kernel - there is no need for any custom kernel patches
- it has low memory footprint since it allows for page cache sharing
- it's faster than the `aufs` driver, although it still suffers from slow copy between lower and upper layers
- identical files are hardlinked between the images, which avoids composing overlays and allows for quicker create/destroy times

Despite all of these advantages, OverlayFS is still a very young filesystem. Although the initial tests look promising, we have yet to see it properly being used in production setups.

Given that, more companies like **CoreOS** are jumping on the OverlayFS **board** train, so we can expect more active development and improvements in the future.

It's time to abandon the awesome realm of Copy on Write magic and look at the last storage driver available in Docker that does not provide it: `vfs`.

vfs

Like we mentioned earlier, `vfs` storage driver is the only available storage driver that does not take advantage of any copy on write mechanism. Each image layer is just a simple directory. When Docker creates a new layer it does a **full physical copy** of the base layer directory into the newly created directory. This can make using this driver very slow and disk space inefficient.

Let's have a look at a practical example when `vfs` is used. As always, first we need to modify DOCKER_OPTS to tell the Docker daemon to use the `vfs` driver and restart it:

```
# ps -ef|grep docker
root      2680     1  0 21:51 ?        00:00:02 /usr/bin/docker
-d --storage-driver=vfs
```

We will illustrate some of the features of `vfs` driver on a small busybox container that will run the familiar `top` utility:

```
# docker run -d busybox top
...
[FILTERED OUTPUT]
...
de8c6e2684acefa1e84791b163212d92003886ba8cb73eccef4b2c4d167a59a4
```

VFS image layers are stored in the `/var/lib/docker/vfs/dir` directory. We will now try to demonstrate the speed and disk space usage inefficiency when using this driver. We will start a new container and generate a reasonably large file inside it:

```
# docker run -ti busybox /bin/sh
/ # dd if=/dev/zero of=sample.txt bs=200M count=1
1+0 records in
1+0 records out
/ # du -sh sample.txt
200.0M    sample.txt
/ #
```

Commit this layer to trigger a creation of a new image layer and observe the speed:

```
# time docker commit 24247ae7c1c0
7f3a2989b52e0430e6372e34399757a47180080b409b94cb46d6cd0a7c46396e
```

```
real    0m1.029s
user    0m0.008s
sys    0m0.008s
```

It took over 1 second to create a new image layer - this is due to an existence of the 200M file we have created in the base image layer. Finally, let's create a new container from the newly committed layer and observe the speed again:

```
# time docker run --rm -it
7f3a2989b52e0430e6372e34399757a47180080b409b94cb46d6cd0a7c46396e
echo VFS is slow

VFS is slow

real    0m3.124s
user    0m0.021s
sys    0m0.007s
```

Creating a new container from the previously committed image layer took over 3 seconds! Furthermore, we now have two containers each taking up 200M in disk space on the host filesystem!

The above example should hopefully demonstrate that VFS was not designed for production use. It is a rather fallback storage driver solution when no copy on write filesystem is available on the host. Despite this, VFS continues to be a suitable solution for Docker VOLUMEs due to its platform portability and that it can be a good option when trying to run Docker on non-Linux platforms such as FreeBSD.

Summary

Docker provides quite a comprehensive choice of storage drivers. This is both a blessing and a curse since often an inexperienced user, once introduced to the great world of possibilities, ends up confused about what to do. The well known quote about **premature optimization** by [Donald Knuth] often comes to mind when discussing technical options.

While you should always pick a tool you have the most experience operating in production environment with, we will finish this chapter by providing a short summary to highlight some important points you might want to take into consideration when making a decision about which storage driver to use.

	AUFS	OverlayFS	BTRFS	DeviceMapper
Provisioning	Very Fast	Very Fast	Fast	Fast
Small Files I/O	Very Fast	Very Fast	Fast	Fast

	AUFS	OverlayFS	BTRFS	DeviceMapper
Large Files I/O	Slow	Slow	Fast	Fast
Memory Usage	Efficient	Efficient	Efficient	Not so efficient
Drawbacks	- Not in mainline Kernel			

- Layer limit
- Random concurrency quirks | - immature | - almost mature, but not really - laborious storage increase - requires solid operational knowledge | - high disk usage with high container density - rough around the edges - most operational knowledge required |

One area that Docker packs a punch is in the area of networking, which will be covered in the next chapter.

12 Docker Networking

In the previous chapter we learned about various Docker storage drivers and how they help to package and run Docker containers from the built images efficiently. The real power of Docker comes to light when using it to build distributed applications. Distributed applications consist of an arbitrary number of services running on a computer network over which they communicate in order to perform some computing task. This chapter will try to answer a simple question: How can we make applications running inside Docker containers accessible on the network? In order to answer this question we need to understand the Docker networking model.

The topic of Docker networking is threefold. It comprises IP allocation, [domain] name resolution and container or service discovery. All of these concepts are the basis of the concept known as *Zero Configuration Networking* or **zeroconf**. In short, **zeroconf** is a set of technologies that automatically create and configure a TCP/IP network without any manual intervention.

The concept of an automatic network configuration becomes especially important in complex environments in which the density of Docker containers on every host can [and often does] rapidly fluctuate. Managing and operating such environments, ideally within secure network infrastructure that sometimes spreads across multiple cloud providers, can be a real challenge. Achieving the zeroconf **"mantra"** in these environments should be the ultimate networking goal. It takes off the burden of manual network configuration from both developers and operators. On the following lines we will discuss what options are provided to us by Docker to get as close to this mantra as possible.

When making the applications running inside Docker containers available on a computer network users often face some of the following questions:

- How do I access an application or service running inside the Docker container?
- How do I connect or discover dependent application services running in Docker containers in a secure way?
- How do I secure the application running in Docker container on the network?

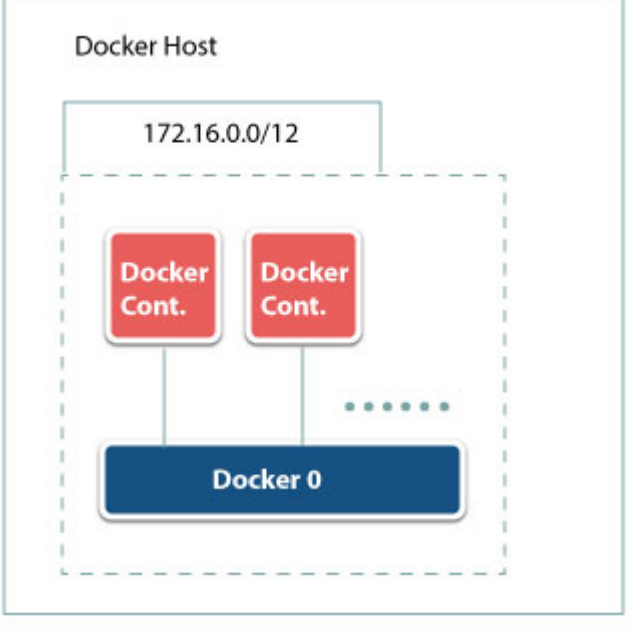
In this chapter we will try to find the answers on these questions. We will start by discussing the Docker networking internals, which should hopefully give you a good understanding of the current networking model. We will then examine some practical examples, which

you can easily follow by running the commands directly on a Docker host. Finally, we will discuss some alternative solutions available on the market should the current networking model not fit your requirements. Let's roll our sleeves up and get to it!

Networking basics

The Docker networking model is very simple, yet quite powerful. By default (i.e. with default Docker daemon configuration), all of the newly created Docker containers are automatically connected to the internal Docker network without any manual intervention required from the user: you simply run the familiar `docker run <image> <cmd>` command and once your container is started it is available on the network. This feels like magic, so let's have a closer look at what is hiding behind it.

When the Docker daemon starts on a host machine with a default configuration, it creates a Linux **network bridge** device and names it `docker0`. The bridge is then automatically assigned a random IP address and subnet from the private IP range defined by **RFC 1918**. The chosen subnet defines the IP range from which all newly created containers are going to have their IP addresses allocated. You can see the current network model in this figure:



Apart from creating the bridge device, the Docker daemon also modifies `iptables` rules on the host machine. It creates a special filter chain called DOCKER and **inserts** it on top of the FORWARD chain. It also modifies the `iptables nat` table to enable the outbound connections from the container to the outside world. The rules are set up so that the connections between the Docker containers will appear to the receiver as coming from the container's IP address, however, the connections to the outside world will appear to originate from one of the host's IP addresses, not the IP address of the container that originated the connection. This sometimes surprises first time users.

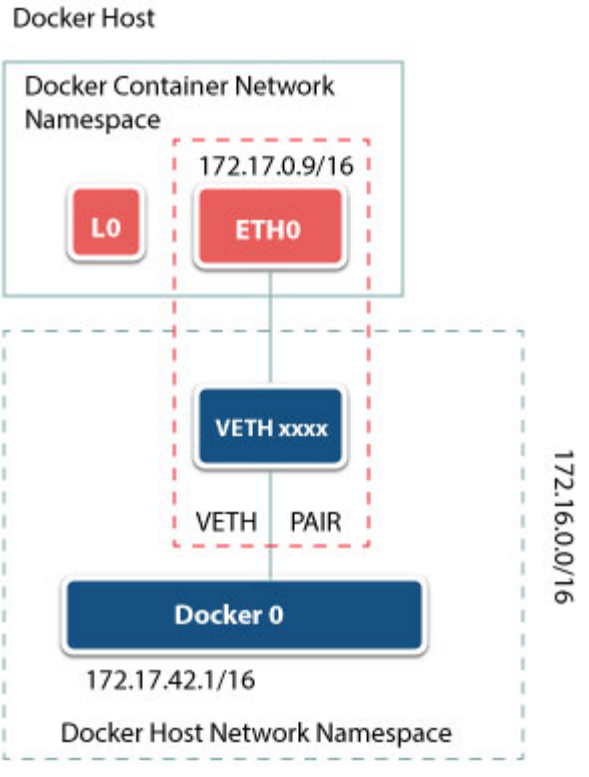
If you don't want Docker to modify any `iptables` rules on the host machine you **must** start the Docker daemon with the `--iptables` option set to **false**. You can do this easily by setting DOCKER_OPTS environment variable and restarting the daemon. In the default configuration this option is set to **true**.

NOTE: *DOCKER_OPTS must be set when the Docker daemon starts. If you want your changes to persist between Docker host restarts, you have to modify some init service configuration or script file. This differs between various Linux distributions; on Ubuntu you can accomplish this by modifying the `/etc/default/docker` file.*

By creating a separate `iptables` chain to manage the access to containers, Docker allows the system administrators to conveniently manage the external access to containers by modifying the DOCKER chain without touching any other `iptables` rules on the host machine that prevents accidental modifications. Appending and modifying the rules in the DOCKER chain is in essence how Docker manages **links** between the containers as we will find out later on in this chapter.

The Docker daemon creates a new network namespace for each newly created container. It then generates a pair of **veth** devices. Veth (shortcut for **V**irtual **Eth**ernet) is a special kind of Linux network device that always comes in pairs (a.k.a. peers) and on a high level acts like a "network pipe": whatever you send on one end comes out on the other end. It turns out, this is a convenient way to communicate between different network namespaces in the Linux kernel. Docker assigns one veth peer to the container network namespace and keeps the other peer, with randomly generated name starting with **veth** prefix, in the host's network namespace. Each peer of the veth pair is only visible in a network namespace it is present in.

Docker then adds the host's veth peer to the `docker0` bridge and assigns the container's veth peer an IP address from the private IP range chosen when the Docker daemon starts. Once the host veth peer is bridged to the `docker0` device, Docker inserts a new network route for the chosen IP range to the host's routing table and enables IP forwarding on the host. This allows the users to communicate with the containers directly from the host. From a detailed point of view we can illustrate this setup as shown here:



By default Docker containers are accessible on a private network, which is not publicly routable. Docker does not facilitate access to the containers from the public network, therefore they are not directly accessible from outside the host machine.

NOTE: *If you start the Docker daemon with the* `--iptables` *option set to* ***"false"****, the daemon will not touch* `iptables` *on the host. Neither will it set up IP forwarding, which means that your containers will not be able to communicate with the outside world nor with each other. If this is not the desired behavior, you will have to enable the IP forwarding manually. On Linux, you can do so by setting the* `/proc/sys/net/ipv4/ip_forward` *kernel parameter to "1".*

IP address allocation

Managing IP address space manually in an environment where the containers can come and go frequently and in large numbers is not a sustainable way of doing things. This is where the Docker IP address allocation steps in. Like mentioned earlier IP allocation is one

of the pillars of **zerconf** networking and Docker has a solution in hand. Docker assigns IP addresses to newly created containers automatically without manual intervention. The Docker daemon manages a list IP addresses that have already been allocated to running Docker containers to prevent assigning the same IP address again. When a container stops or is removed, its IP address is released back into the IP address pool managed by the Docker daemon and can be reused straight away when a new container starts.

Instant reuse of the released IP addresses can cause **ARP** collisions on a local network if the cached ARP entry mapping the released IP address to the container's **MAC address** has not been purged immediately after the container was destroyed. Docker addresses this problem by generating a random MAC address from the allocated IP address. The MAC address generator is guaranteed to be consistent: the same IP will always yield the same MAC address. Docker also allows users to specify the MAC address manually when creating a new container, however, we would not recommended doing this due to the above mentioned issue with the ARP collisions, unless you come up with some mechanism that will help you avoid it.

This is great! IP [and MAC address] allocation happens **"automagically"** without any user manual intervention. As soon as the containers are created they are available on the private Docker network via their automatically assigned IP addresses. This is exactly what you would expect from **zeroconf**. Docker takes this even one step further: in order to enable the communication with the services running inside the containers Docker must also handle **port allocation**.

Port allocation

When a container starts, Docker **can** automatically assign arbitrary UDP or TCP port(s) to it and make them accessible on the host machine. The ports can be **exposed** either via the EXPOSE directive in `Dockerfile`, from which the container image is built, or by using the `--expose` command line switch when the container is started. The command line flag allows you to define a port range as opposed to a single port, however, be aware of the implications of long port ranges since all of this information is made available by the Docker daemon via remote API, so querying a container that has a big port range can easily expose the daemon.

The Docker daemon then picks a random port number from a port range defined on Linux hosts from the following file: `/proc/sys/net/ipv4/ip_local_port_range`. If this fails, for example when the Docker daemon runs on a non-Linux host, Docker tries to allocate the port from the following port range: **49153-65535**. This is not done automatically: the Docker daemon merely maintains the port numbers exposed by containers running on the host. Users must explicitly trigger the host port mapping by what Docker calls **"port publishing"**. Port publishing allows the users to bind any **exposed port** to any externally routable IP address available on the host machine. If you did not expose any ports when you built a Docker image, publishing ports

will have no effect on external availability of the service running inside the Docker container.

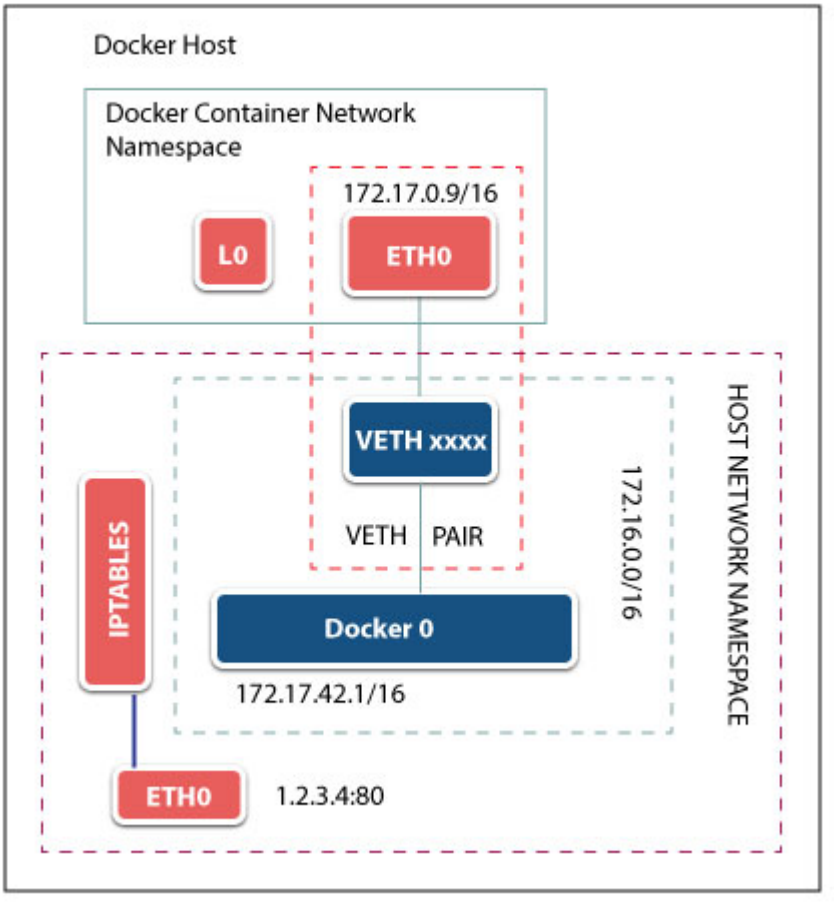
You can find out what ports have been exposed in the container by running the following command:

```
# docker inspect -f '{{.Config.ExposedPorts}}' <container_id>
```

WARNING: *You can only publish any of the exposed ports when you start a new container; once the container is running there is no way to publish any more of its exposed ports. You must recreate the containers from scratch!*

You can choose to publish all exposed ports or just pick the ones you are willing to make available externally. Docker provides convenient command line options to achieve various combinations. Please do check the Docker help for all of the available options.

The magic behind port allocation lies in the `iptables` "dance" Docker does with the previously mentioned DOCKER chain and `nat` table. To understand this better we will examine a practical example illustrated in the following diagram:



Let's say we want to run a `nginx` web server in a Docker container and make it publicly available on the host via externally routable IP address `1.2.3.4` on **TCP** port `80`. We will use the `library/nginx` Docker image, which is the default `nginx` image that gets pulled from **DockerHub** when you run `docker pull nginx`. We can accomplish this task by running the following command (note the use of the `-p` command line option):

```
# sudo docker run -d -p 1.2.3.4:80:80 nginx
a10e2dc0fdfb2dc8259e9671dccd6853d77c831b3a19e3c5863b133976ca4691
#
```

You can verify that the container created from this image exposes TCP port 80 by running the following command:

```
# sudo docker inspect -f '{{.Config.ExposedPorts}}' a10e2dc0fdfb
map[443/tcp:map[] 80/tcp:map[]]
```

You can see that the `nginx` image we used to start the container also exposes port 443. Now that the container is running you can easily inspect all of its **published** ports (a.k.a *host port bindings*) by running the following command:

```
# sudo # docker inspect -f '{{.HostConfig.PortBindings}}'
a10e2dc0fdfb
**map[80/tcp:[map[HostIp:1.2.3.4 HostPort:80]]]**
```

TIP: *There is a convenient command line shortcut for checking IP:port bindings of a particular Docker container:* `docker port container_id`.

Excellent! `nginx` is now running inside the Docker container and its service should be publicly accessible on the given IP address and port. We should be able to access the default `nginx` site by running a simple `curl` command from outside the host machine (given that your firewall is not blocking the access to port 80 from the outside):

```
# curl -I 1.2.3.4:80
HTTP/1.1 200 OK
Server: nginx/1.7.11
Date: Wed, 01 Apr 2015 12:48:47 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 24 Mar 2015 16:49:43 GMT
Connection: keep-alive
ETag: "551195a7-264"
Accept-Ranges: bytes
```

When you publish a port on the host machine, the Docker daemon appends a new `iptables` rule to the `DOCKER` chain, which forwards all of the traffic destined to `1.2.3.4:80` on the host to the particular container on port `80` and also modifies the `nat` table accordingly. You can easily inspect this by running the following commands:

```
# iptables -nL DOCKER
Chain DOCKER (1 references)
target     prot opt source               destination
**ACCEPT     tcp  --  0.0.0.0/0            1.2.3.4
tcp dpt:80**

# iptables -nL -t nat
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT)
```

```
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8
ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0
**MASQUERADE  tcp  --  172.17.0.5
172.17.0.5           tcp dpt:80**

Chain DOCKER (2 references)
target     prot opt source               destination
**DNAT       tcp  --  0.0.0.0/0            1.2.3.4         tcp
dpt:80 to:172.17.0.5:80**
```

NOTE: *When you stop or remove a container that published an arbitrary number of ports, Docker removes all of the necessary* `iptables` *rules that it originally created, so you don't have to worry about them any more.*

Finally, let's have a look at what happens if we start a new container with the `-P` switch enabled, which does not require us to specify any host port or IP addresses to bind the exposed ports to. We will use the same `nginx` image as in the previous example:

```
# docker run -d -P nginx
995faf55ede505c001202b7ee197a552cb4f507bc40203a3d86705e9d08ee71d
```

As mentioned earlier, since we didn't explicitely specify any host interfaces to bind the container's ports to, Docker will map the ports to a random port range. To discover the container port bindings you can run the following command:

```
# docker port 995faf55ede5
443/tcp -> 0.0.0.0:49153
80/tcp -> 0.0.0.0:49154
```

You can also inspect the networking ports directly by running the following command:

```
# docker inspect -f '{{ .NetworkSettings.Ports }}' 995faf55ede5
map[443/tcp:[map[HostIp:0.0.0.0 HostPort:49153]] 80/tcp:
[map[HostIp:0.0.0.0 HostPort:49154]]]
```

You can see that both exposed TCP ports have been bound to **all network interfaces on the host** to ports `49153` and `49154` respectively. As you may have guessed, Docker exposes this information via its remote API, which allows for a very simple service discovery of the containers that need to communicate with each other. When you are starting a new

container you can pass it the host port mappings via an environment variable. There is a better, and more secure way, which we'll discuss later on in this chapter.

All of this looks fantastic. We don't have to manually manage IP address space for Docker containers running on the host: the Docker daemon does all of this right out of the box! However, some problems can arise when we start scaling our Docker infrastructure. Since the Docker daemon is completely in charge of the IP assignment on a particular docker host, how do we make sure that the same IP address is not assigned to different containers on different hosts? Do we even need to care about this problem? The answer to these questions is the very familiar: **it depends**. We will explore this topic further in the chapter when we discuss advanced networking concepts. Let's now move on to the next zeroconf pillar: domain name resolution.

Domain name resolution

Docker provides a few options that allow you to manage domain name resolution in your container infrastructure. As always, in order to take advantage of these options and avoid unnecessary surprises we need to understand how Docker handles the name resolution internally.

When Docker creates a new container, by default it assigns it a randomly generated **hostname** and a unique **container name**. These are two completely different concepts that can be a bit confusing for the newcomers as their use sometimes overlaps.

The hostname works just like a regular Linux hostname: it allows the processes running inside the container to resolve the container hostname to its IP address. Container hostname is **NOT resolvable from the outside of the container environment** and by default it is set to **container ID** which is a unique string that allows you to address any container on the host machine from the command line or via remote API. **Container name**, on the other hand, is a concept internal to Docker and it has nothing to do with any Linux internals.

You can easily inspect the container name using the Docker command line client by running the following command:

```
docker inspect -f '{{.Name}}' container_id
```

The container name has two main uses in Docker:

- it allows you to address containers via remote API using human readable names as opposed to container IDs
- it helps to facilitate basic container discovery via Docker

You can override the default values set by the Docker daemon by using the particular command like options provided by the Docker client.

Both the container hostname and the container name can be set to the same value, but unless you have some way of managing them automatically, we recommend you avoid do-

ing so, as with a higher density of the containers, this can turn into an unmanageable situation and a maintenance nightmare.

Neither the hostname nor the name are hard-coded into the container images when they are built. What Docker actually does is that it generates `/etc/hostname` and `/etc/hosts` files on the Docker host and then bind-mounts both files into the newly created container on its start. You can easily verify this by checking the contents of the following files on the host machine:

```
# cat /var/lib/docker/containers/container_id/hostname
# cat /var/lib/docker/containers/container_id/hosts
```

TIP: *You can find out the paths to the above files by running* `docker inspect -f '{{printf "%s\n%s" .HostnamePath .HostsPath}}' container_id` *on the command line.*

We will discuss the **container name** concept in more detail later on in this chapter; for now let's just remember that if you want to address Docker containers by a human readable name instead of by a randomly generated container ID, you can assign them a custom name. You can **NOT** change the assigned name after the container has been created--you will have to recreate it from scratch.

NOTE: *As of version 0.7, Docker generates container names from the names of notable scientists and hackers. If you would like to contribute to the Docker project, you can open a* ***Pull Request*** *on* ***Github*** *suggesting a name of a person whom you think deserves a recognition. The* ***Go*** *package which handles this in Docker is called* `namesgenerator` *and you can find it in the pkg subdirectory of the Docker codebase.*

Now that you know how a container resolves its hostname to its IP address it's time to find out how it resolves external DNS names. If you guessed that the containers use the `/etc/resolv.conf` file just like a regular Linux host you would be spot on! Docker generates the `/etc/resolv.conf` file for each newly created container on the host machine and mounts it inside the container when it starts it. You can easily verify this by inspecting the following file on the Docker host:

```
# sudo cat /var/lib/docker/containers/container_id/resolv.conf
```

TIP: *You can find out the path to the above file by running* `docker inspect -f '{{.ResolvConfPath}}' container_id` *on the command line.*

By default Docker reuses the host machine's `/etc/resolv.conf` file in the newly created containers. As of version **1.5** when you modify this file to change the host machine DNS settings and want these changes to be propagated into already running containers (that were created with the original settings) you **must restart** all of them in order to pick up the new changes. If you are using an older version of Docker this is not the case and you **must recreate** the containers from scratch.

If you don't want your Docker containers to use the host machine's DNS settings you can easily override them by modifying the DOCKER_OPTS environment variable, either on the command line when starting the daemon or you can make the changes persistent by modifying a particular configuration file on the host machine (on Ubuntu Linux distribution it is `/etc/default/docker`).

Let's say you have a dedicated DNS server that you want the Docker containers to use for DNS resolution. We will assume that the DNS server is accessible on the `1.2.3.4` IP address and that it manages **example.com** domain. You would modify the DOCKER_OPTS environment variable as per the following:

```
DOCKER_OPTS="--dns 1.2.3.4 --dns-search example.com”
```

In order for the Docker daemon to pick up the new settings you need to restart it. You can easily verify that newly created containers now reflect the new DNS settings by inspecting the `/etc/resolv.conf` file by running the following command:

```
# sudo cat /var/lib/docker/containers/container_id/resolv.conf
nameserver 1.2.3.4
search example.com
```

All of the containers that were started before you modified the Docker daemon DNS settings will remain unaffected. If you want these containers to use the new settings, you must recreate them from scratch: simply restarting the containers will not have the desired effect!

NOTE: *The earlier mentioned Docker daemon DNS options will NOT override the host DNS settings. They will merely become the default DNS settings for all of the containers created on the host machine from the point the daemon picks up the change. You can, however, explicitly override them per container.*

Docker allows for even more fine grained control over the container DNS settings. You can override the global DNS settings on the command line when starting a new container like this:

```
# sudo docker run -d --dns 8.8.8.8 nginx
995faf55ede505c001202b7ee197a552cb4f507bc40203a3d86705e9d08ee71d
# sudo cat $(docker inspect -f '{{.ResolvConfPath}}'
995faf55ede5)
nameserver 8.8.8.8
search example.com
```

There is a small catch that you may have noticed. As you can see in the output of the command above, we only set the `--dns` setting for the new container, yet `search` directive inside the `/etc/resolv.conf` file has been modified, too. Docker **merges** the options specified on the command line with the settings imposed by the Docker daemon. When the Docker daemon sets `--dns-search` to some domain and you only override

the `--dns` option then when you start a new container. The container will inherit the daemon's search domain setting, not the settings set by the host machine. At the moment there is no way to change this behavior so keep an eye on this!

Additionally, containers created with custom DNS settings will not be affected by any changes made to the Docker daemon DNS settings even after you restart these containers. If you want them to pick up the settings set by the Docker daemon, you will have to recreate them without specifying the custom options.

WARNING: *If you modified either of the `/etc/resolv.conf`, `/etc/hostname`, or `/etc/hosts` files directly inside the running container, be aware that the changes you have made will not be saved by `docker commit` nor will they persist when the containers are restarted!*

As you can see in this chapter, Docker configures DNS settings in each container automatically without any manual intervention required from the user. This is once again perfectly in line with the **zeroconf** mantra. Even if you export the container and migrate it to another host, Docker will pick up its original DNS settings, so you don't have to worry about configuring them again from scratch.

Being able to resolve external DNS names from inside the containers straight after they are started is very convenient, but what if we want to access some container from inside another container? As you learned earlier, container hostnames are not resolvable from outside the containers themselves, so you can't use the container hostnames to enable inter container communication. In order for one container to communicate with another container it will have to know the other container's IP address, which you can find out by querying the Docker remote API. This is not possible from inside the container unless you bind mount the Docker daemon socket on container start or expose the Docker API on the `docker0` bridge interface. Furthermore, querying the API requires an extra effort that often introduces unnecessary complexity and is not really in line with the zeroconf ideal we set out to pursue. The solution to this problem comes down to the final zeroconf pillar: *service discovery*.

Service discovery

Out of the box Docker provides a basic, yet powerful service discovery mechanism: **docker links**. As we have already learned, all of the containers are accessible on the Docker private network, so by default all containers can communicate directly with each other if they know each other's IP addresses. Discovering IP addresses of another container is not enough. You also need to discover the port on which the containerized service accepts incoming connections.

Docker linking allows a user to let any container discover both the IP address and exposed ports of other Docker containers. In order to accomplish this, Docker provides a convenient command line option that does this automatically for us without too much effort. This option is called `--link`. When you create a new container and *link* it with another

container Docker exports all of the connection details to the source container via various environment variables. This can be hard to understand. Let's look at an example to make this much clearer.

We will illustrate the Docker linking on the `nginx` container, which we started when we were exploring IP address and port allocation. You can find out the container's IP address by running the following command:

```
# docker ps -q
a10e2dc0fdfb
# docker inspect -f '{{.NetworkSettings.IPAddress}}'
a10e2dc0fdfb
172.17.0.2
```

The `--link` syntax follows the following pattern: `container_id:alias` where the **container_id** is the id of the running container and **alias** is an arbitrary name that we will explain later. We will try to ping the IP address of the `nginx` container from inside a new, *"throwaway"* container (the `--rm` flag removes the container once the command executed in it exits):

```
# docker run --rm -it --link=a10e2dc0fdfb:test busybox ping -c
2 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.615 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.248 ms

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.248/0.431/0.615 ms
```

This is an expected outcome, since by default the containers are able to communicate with each other on the private network (unless the inter container communication is disabled as you will find out later in the advanced container networking chapter).

When the containers are linked Docker updates the `/etc/hosts` file of the source container with the IP address of the destination container using the link `alias` name provided on the command line, which in our case is simply **"test"**. You can easily verify this by running the command below (see the entry at the bottom of the output):

```
# docker run --rm -it --link=a10e2dc0fdfb:**test** busybox
cat /etc/hosts
172.17.0.13    a51e855bac00
127.0.0.1    localhost
::1    localhost ip6-localhost ip6-loopback
fe00::0    ip6-localnet
ff00::0    ip6-mcastprefix
ff02::1    ip6-allnodes
```

```
ff02::2    ip6-allrouters
**172.17.0.2    test**
```

This means that instead of using the destination container IP address for the previously done ping test, you could have easily referred the linked container via it's alias name as shown here:

```
# docker run --rm -it --link=edb055f7f592:test busybox ping -c
2 test
PING test (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.492 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.230 ms

--- test ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.230/0.361/0.492 ms
```

We mentioned earlier when linking two containers Docker also creates a few environment variables that can help the source container easily discover the ports exposed by the destination container. The source container does not need to know neither the IP address nor the exposed ports of the destination container when it tries to establish a connection with it. It can simply read the contents of the environment variables, which are automatically created by Docker **for every exposed port** and are made available to the source container execution environment. The names of these environment variables have the following pattern:

```
ALIAS_NAME
ALIAS_PORT
ALIAS_PORT_<EXPOSEDPORT>_TCP
ALIAS_PORT_<EXPOSEDPORT>_TCP_PROTO
ALIAS_PORT_<EXPOSEDPORT>_TCP_PORT
ALIAS_PORT_<EXPOSEDPORT>_TCP_ADDR
...
...
```

You can verify this by running the following command and inspecting its output:

```
# docker run --rm -it --link=a10e2dc0fdfb:test busybox env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin
HOSTNAME=ff03fba501ea
TERM=xterm
TEST_PORT=tcp://172.17.0.2:80
TEST_PORT_80_TCP=tcp://172.17.0.2:80
TEST_PORT_80_TCP_ADDR=172.17.0.2
TEST_PORT_80_TCP_PORT=80
TEST_PORT_80_TCP_PROTO=tcp
```

```
TEST_PORT_443_TCP=tcp://172.17.0.2:443
TEST_PORT_443_TCP_ADDR=172.17.0.2
TEST_PORT_443_TCP_PORT=443
TEST_PORT_443_TCP_PROTO=tcp
TEST_NAME=/mad_euclid/test
TEST_ENV_NGINX_VERSION=1.7.11-1~wheezy
HOME=/root
```

From the above you can see that the `nginx` container we have been using to explore the linking, exposes two TCP ports: 80 and 443. Docker therefore creates two environment variables: `TEST_PORT_80_TCP` and `TEST_PORT_443_TCP` for each of the exposed ports. By exporting the above environment variables through linking, Docker provides simple and portable service discovery while keeping the containers safe and secure. However as it happens, nothing comes for free. Links are an awesome concept, yet they are fairly static as you will find out.

Whenever a destination container you are linking to dies, the source container loses the connection to the service provided by the linked container. This is a problem in dynamic environments where containers come and go very frequently. Unless you implement some basic health checking into your application, or at least some failover, you should keep an eye on this behavior.

Furthermore, when you bring the broken destination container back up, the source container's `/etc/hosts` file is automatically updated with the newly allocated IP address of the destination container, but **the environment variables exported via links on the source container are not updated**, so this is not an ideal situation if you don't know the port of the service in advance. Additionally, you should not rely on these environment variables for discovering the IP address of the destination containers, but preferably use the link alias, which will resolve to the updated IP address via `/etc/hosts`.

One solution to address the static nature of the links is a tool called **docker-grand-ambassador**. It addresses some of the issues you can encounter when using Docker links. A more advanced and widely adopted solution to use for container discovery without introducing extra complexity is to use the good old DNS. There are some open source DNS server implementations available that provide service discovery for Docker containers out of the box and that don't require too much hassle to integrate into your Docker infrastructure.

By discussing the Docker service discovery we have concluded the tour of the **zeroconf** pillars. You can see that as long as you run all of your containers on one host machine, Docker can satisfy all of the zeroconf requirements to a high degree. However, Docker unfortunately fails to deliver once you start scaling containers across multiple machines or even multiple cloud providers. We are very excited about the upcoming development of **docker swarm** and the new networking **model**, which will address the woes discussed in this chapter.

Let's now move to more advanced networking topics that will dive deeper into the core networking model and discuss some solutions available to you to address the issues discussed in the previous chapters.

Advanced Docker networking

We will start this section by discussing network security, then we will move on to the topic of inter container communication across multiple Docker hosts and finish by discussing network namespace sharing.

Network security

The topic of network security is quite complex and it could easily cover a whole separate book. We will only touch on a few points in this chapter that you should keep an eye on when designing your Docker infrastructure or when using one provided for you by an external vendor. Our focus will be on solutions which are native to Docker and we will mention other non-native alternatives at the end of this chapter.

By default, Docker allows inter-container communication without any restrictions. This is, as you may have guessed, a security risk. What if one of the containers on the Docker network gets compromised and launches a **Denial of Service attack** on any other containers on the same network? The attack can be triggered either deliberately or it could simply be a result of a software bug. This is especially concerning in multi-tenant environments.

Luckily, Docker allows you to completely disable **I**nter **C**ontainer **C**ommunication by passing a special flag to the Docker daemon when it starts. The flag is called `--icc` and by default it's set to **true**. If you want to completely disable the communication between Docker containers you must set this flag to **false** by modifying the DOCKER_OPTS environment variable and restart the Docker daemon afterwards. What happens in the background is, the Docker daemon inserts a new DROP policy `iptables` rule to the FORWARD chain, which drops all of the packets destined for the Docker containers. You can easily verify this once the Docker daemon has been restarted by running the command below:

```
# sudo iptables -nL FORWARD
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
**DROP       all  --  0.0.0.0/0            0.0.0.0/0**
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ctstate RELATED,ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
```

From this moment on, the containers can no longer communicate with each other. We can verify this by doing a simple `ping` test against the already running `nginx` container. First, we need to find out the IP address of the `nginx` container:

```
# docker inspect -f '{{.NetworkSettings.IPAddress}}'
a10e2dc0fdfb
172.17.0.2
```

Now, let's ping it from a throwaway container:

```
# docker run --rm -it busybox ping -c 2 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

As you can see, the ping test has **correctly** failed since the inter-container communication has been disabled. We can, however, still reach the `nginx` container from the host machine:

```
# ping -c 2 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.098 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.066 ms

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.066/0.082/0.098/0.016 ms
```

Additionally, all of the previously published ports remain untouched, so the `nginx` server running inside the `nginx` container is still perfectly accessible:

```
# curl -I 1.2.3.4:80
HTTP/1.1 200 OK
Server: nginx/1.7.11
Date: Wed, 01 Apr 2015 12:48:47 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 24 Mar 2015 16:49:43 GMT
Connection: keep-alive
ETag: "551195a7-264"
Accept-Ranges: bytes
```

How do we now enable access **only** between the containers that need to communicate with each other? The answer is `iptables`. If you don't like manipulating the `iptables` manually, don't worry. Docker provides a convenient command line option, which does this automatically for you when you create a new container. In fact, we are familiar with this option since we explored the service discovery: `--link`. Linking the Docker containers not only allows for a simple service discovery mechanism, but it also secures the communication between the linked containers: **ONLY** the containers that are linked together

can talk to one another and even then **ONLY** via the ports which are exposed by them. Docker does this by inserting a **"bidirectional"** communication `iptables` rules into DOCKER chain.

Practical example will help us understand this better. Again, we are going to reuse the already running `nginx` container and link to it from a throwaway container. Let's first verify that links really allow only the communication on the exposed ports. Since the `nginx` container is still running it should have the same IP address allocated:

```
# docker inspect -f '{{.NetworkSettings.IPAddress}}'
a10e2dc0fdfb
172.17.0.2
```

Let's try to ping it from within a throw away container:

```
# docker run --rm -it -link=a10e2dc0fdfb:test busybox ping -c 2
172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes

--- 172.17.0.2 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

Ping is **correctly** failing as expected because ping uses **ICMP protocol**, which is a network layer protocol and has no concept of port. We can easily verify that the default site served by the `nginx` container is still perfectly accessible as expected:

```
# docker run --rm -link=a10e2dc0fdfb:test -ti busybox wget
172.17.0.2
--2015-04-01 14:11:58--  http://172.17.0.2/
Connecting to 172.17.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 612 [text/html]
Saving to: `index.html'

100%
[======================================================================>]
612         --.-K/s   in 0s

2015-04-01 14:11:58 (16.4 MB/s) - `index.html' saved [612/612]
```

You can inspect the `iptables` rules which have been created by Docker when the containers were linked by checking the DOCKER chain using the following command:

```
# iptables -nL DOCKER
Chain DOCKER (1 references)
target     prot opt source               destination
ACCEPT     tcp  --  0.0.0.0/0            172.17.0.2
tcp dpt:80
```

```
ACCEPT     tcp  --  172.17.0.9           172.17.0.2
tcp dpt:443
ACCEPT     tcp  --  172.17.0.2           172.17.0.9
tcp spt:443
ACCEPT     tcp  --  172.17.0.9           172.17.0.2
tcp dpt:80
ACCEPT     tcp  --  172.17.0.2           172.17.0.9
tcp spt:80
```

Like we have already discussed, another alternative to secure the inter container communication is by modifying `iptables` rules in the `DOCKER` chain directly. We would advise you not to do this, because with the higher density of containers running on the host this can become a real maintenance problem since you must keep the track of the every container lifetime and continuously add and remove the rules when the containers come and go.

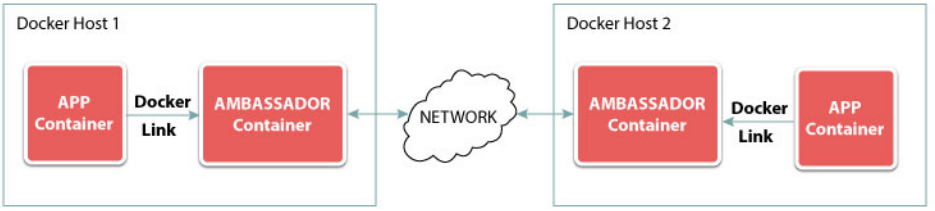
Another important point with regards to security is a network segmentation. At the moment Docker does not provide any native way to segment your container network. All of the container traffic is passing through the `docker0` bridge device. The Linux bridge is a special network device that runs in **promiscuous** mode. What this means is anyone with root privileges on the host machine can snoop all of the container traffic. This can be a bit concerning in multi-tenant environments, so you should always make sure any network traffic entering and leaving your containers is encrypted on both ends.

Like we've said, there is so much more to talk about with the network security. We have merely scratched this topic on the surface to highlight some of the basic concepts. We will now move on and explore how we can interconnect the containers across multiple Docker hosts.

Multihost inter-container communication

As always, we have a few options at our disposal to achieve this. Armed with our knowledge of port publishing and container linking we can apply what came to be known in the Docker community as **"ambassador pattern"**, which combines both of these concepts.

Ambassador pattern works by interconnecting containers running on different Docker hosts using a special container that runs on all of the Docker hosts you are trying to interconnect. This special container runs a `socat` network proxy that proxies the connections between the Docker host machines. The following diagram should hopefully make this clearer:



You can read more about this pattern in the Docker official **documentation**. In the gist, this patterns boils down to publishing ports on one host and discovering the published bindings on the other host via environment variables exposed to us through linking to the ambassador container running on the other host. The downside of this pattern is, you must run extra container on each host which handles the proxying. The upside is, your containers can become more portable if you refer to destination container via their name as oppose to container IDs. If you export any container and transfer it to another docker host where you have another container running which uses the same name as your exported container is supposed to be linked in to, all you need to do is just to start the migrated container and you're done.

Now that we have covered how to connect containers across multiple hosts via native Docker options, it's time to look at more complicated concepts. In particular, we will look at how to run and integrate containers on your own dedicated private network.

Docker allows you to explicitly specify an IP address range for the network you want to run your containers in. Like we learned earlier, the IP addresses of the containers are allocated in the private IP subnet chosen randomly by the Docker daemon when it starts. If you want to run the containers on your own dedicated network you can specify your own IP subnet by passing special options to the Docker daemon. This seems like a natural way to interconnect the containers across multiple hosts on your own private network.

You must first assign an IP address to the `docker0` bridge and then specify an IP address range for the Docker containers. The IP address range **must be a subset** of the subnet the bridge has been placed into. So if we wanted our containers to be on the `192.168.20.0/24` network we could set the `DOCKER_OPTS` environment variable to the following value:

```
DOCKER_OPTS="--bip=192.168.20.5/24 --fixed-cidr=192.168.20.0/25"
```

What seemed like an easy and convenient way to configure custom private networking for Docker containers, actually requires adding special routes and `iptables` rules on the host machine. Another downside to this solution is as you may have expected the IP address assignment, which we discussed earlier. Since the Docker daemons don't communicate with each other, IP address allocations can lead to IP address clashes. This will hopefully be addressed by Docker once the **libnetwork** is introduced in the future version,

which should be pretty soon. In the meantime we can take advantage of integrating third party tools with Docker.

One popular solution to private Docker multihost networking is to use **Open Virtual Switch** (OVS) and **GRE tunnelling**. The idea behind this is to replace the default `docker0` bridge with the one created by OVS and create a secure GRE tunnel between the hosts on the private network. This requires at least a basic knowledge of how OVS works and can get a bit too complex the more Docker hosts you are trying to interconnect (this will be addressed in the future releases of Docker via **libnetwork** or can be avoided by using **swarm**). By default, using OVS still does not solve the issue of IP address allocation across multiple Docker hosts like we have mentioned few times earlier, so you will need to come up with some solution to it. You can read more about how you can go about using OVS with Docker on the following **blog post**.

Docker offers another way to make the inter-container communication across multiple hosts easier: **network namespace sharing**.

Network namespace sharing

The concept of network namespace sharing has been popularized by the **Kubernetes** project where it became known as a **pod**. We will show how you can exploit network namespace sharing for interconnecting the Docker hosts.

When starting a new container you can specify a container **"network mode"**. Network mode allows you to specify how the Docker daemon creates network namespaces in the newly created container. The Docker client provides the `--net` command line option to do this. By default it is set to `bridge`, which we have already described at the very beginning of this chapter.

In order to share the network namespace between arbitrary number of containers, the argument which we need to pass to `--net` command line switch has the following format: `container:NAME_or_ID`.

In order to use this option, you must first create a "source" container that will create a "base" network namespace which other containers can join. Again a simple example will illustrate this better.

We will create a new container that will join a network namespace created by our familiar and already running `nginx` container. But first, let's list the network interfaces inside the nginx container:

```
# docker exec -it a10e2dc0fdfb ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UN-
KNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
71: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP mode DEFAULT
    link/ether 02:42:ac:11:00:05 brd ff:ff:ff:ff:ff:ff
```

There should be nothing surprising in the list of interfaces printed above. We will now create a new throw away container that will join the above container's namespace and list the network interfaces available to it:

```
# docker run --rm -it --net=container:a10e2dc0fdfb busybox ip
link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UN-
KNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
71: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP mode DEFAULT group default
    link/ether 02:42:ac:11:00:05 brd ff:ff:ff:ff:ff:ff
```

This is again exactly what we would expect based on the documentation: the newly created container can see and bind to the `nginx` container network interfaces. No new network namespace is created for the new container - it merely joins an existing namespace.

The advantage of namespace sharing is a smaller resource usage in the Linux kernel, as well as simpler network management in some situations: there is no need to keep an eye on any `iptables` rules, and incoming connections can appear to originate from the container IP if the connection is established on the internal network. Another advantage is that you can communicate between the services running inside the containers via loopback interface. However, you can no longer bind different services to the same IP address and port.

Furthermore, if the source container stops running you will have to recreate it and get all the other containers to rejoin it's new network namespace: network namespaces can't be recycled! You can get around this problem by creating a "named" network namespace. You can find out more about network namespaces in this **presentation**. Docker currently does not provide an easy way to list what containers share network namespace together.

Now that we have a better understanding of the network namespace sharing between Docker containers, we will have a look at how we can exploit this for connecting multiple Docker hosts. The magic again lies in the `--net` command line option.

Docker allows you to create containers that share the network namespace with the host machine. The host machine itself is running it's network stack in the namespace of the process with PID 1, therefore it is perfectly possible for containers to simply join the host network namespace. What this translates into is: container network stack is shared with the host.

This means that the container, which joins the host's network namespace has the **read-only** access (unless `--privileged` flag is passed) to the host's network. Whenever you start a new service in the container that shares the network namespace with the host, the service can bind to any of the IP addresses available on the host and therefore it is available on the host without any further effort.

WARNING: *Try to avoid sharing network namespace with the host as you might be introducing a potential security hole on your host machine. Until the security woes of Docker are addressed you must do this with caution!*

To understand what this looks like in practice we will look at a very simple example. We will create a new container, list the network interfaces inside it, and compare it to the network interfaces available on the host. First, let's list the network interfaces on the host by running the following command:

```
# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UN-
KNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfi-
fo_fast state UP mode DEFAULT group default qlen 1000
    link/ether 04:01:47:4f:c6:01 brd ff:ff:ff:ff:ff:ff
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc no-
queue state UP mode DEFAULT group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
```

There's nothing extraordinary to notice above: we have a loopback device, one ethernet device and a Docker bridge. Now, let's create a new throw away container that shares the network namespace with the host and lists the network interfaces available inside it:

```
# docker run --rm --net=host -it busybox ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UN-
KNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfi-
fo_fast state UP mode DEFAULT group default qlen 1000
    link/ether 04:01:47:4f:c6:01 brd ff:ff:ff:ff:ff:ff
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc no-
queue state UP mode DEFAULT group default
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
```

As you can see, no new network interface has been created for this container and all of the host interfaces are available to it. This allows for easy container interconnection between the hosts as it simply removes the need for docker private network. This can however make your containers easily exposed on the network and removes a lot of advantages you gain if you run containers in their own private network namespaces.

Furthermore, when sharing the network namespace with the host we need to keep an eye on one not very obvious thing. Network namespace includes Unix sockets, which means that all the the Unix sockets exposed on the host by various operating system services are available to the container as well. This can sometimes lead to unexpected **surprises**. Keep this in mind when you start stopping services inside the container that shares

the network namespace with the host and try to avoid running these containers in privileged mode.

A simple example case where this could be useful is to proxy a traffic via SSH-agent **at docker build time** when you don't have the private SSH key available to you, since having it packaged in the container image is a security risk (unless you recycle the keys after every build). One workaround to this problem is to run a `socat` proxy container, which shares the network namespace with the host and therefore has access to the `ssh-agent` on the host. The proxy can then listen on a TCP port through which you can proxy the private traffiɑt build time. You can see an example of this in the following **gist**.

A more secure version of this is to create a network namespace as normal and then move a host interface into the new network namespace, using the `ip link set netns` command, and then configuring it in the container. There is not yet any Docker command line support for this.

This type of setup precludes running in most cloud environments, as they rarely provide more than a single physical network port, and do not accept additional MAC addresses being added to existing ports. However, if you are running on physical hardware multiple physical ports, SR-IOV ports, vlans or macvtap interfaces are all possible.

IPv6

Most Docker deployments use the standard IPv4, but many large scale data center operators such as Google and Facebook have switched to using the newer IPv6 internally. There is no shortage of IPv6 addresses, as addresses are 128 bits long, and the standard allocation for a single host is a /64, or 18,446,744,073,709,551,616 addresses!

This means that it is perfectly possibly for each container to have its own globally routable IPv6 address, which means that communication between containers on different host is much simpler.

Recent versions of Docker have IPv6 support, and setup **is now well documented**. Docker needs at least a /80 address to allocate to containers, which means that it can work on a host with a /64 or even in a larger setup where a single /64 is divided between multiple hosts, in a setup with up to 4096 hosts. Direct routing between containers on different hosts is possible.

Again, in a cloud environment there are few providers that provide IPv6 yet, but this is gradually changing. Digital Ocean only provides a very minimal 16 IPv6 addresses per host, which is supported but with difficulty, as discussed in the NDP proxy section of the documentation. However, there are other providers such as Vultr and BigV from Bytemark that provide cloud services with a full /64 that makes Docker ipv6 support easy.

The ongoing IPv4 address shortage may well encourage more people to look at IPv6 as a way to return to simpler network topologies without NAT, but the lack of widespread support does make it problematic still.

Summary

As shown in this chapter, the Docker engine provides an extensive amount of options available to you to connect the containers in your infrastructure. Sadly, more advanced setups require a fair amount of user effort and solid understanding of networking internals like routing and `iptables` that can be rather off putting for a user who just wants to run her application and not worry about the nitty gritty low level networking details. Furthermore, the native options provided by Docker are rather static and don't scale very well across multiple hosts. We hope that all of these woes will eventually be solved by the promised network API which will be based on the `libnetwork` we mentioned earlier in a future Docker release. There is no need to worry until this materializes, as this is where the awesome rapidly growing Docker ecosystem steps in.

While standard Docker networking tries to abstract away the network setup, the use of `iptables` and potentially tunnelling protocols has a performance cost. **New Relic** found out that in some cases, for applications needing high network performance, it could be as much as twenty times slower to use the standard docker networking setup.

There are lots of ready to use tools available to you that can address the discussed networking issues. The simplest, but still very powerful is called **pipework** created by the one and only **Jérôme Petazzoni** of Docker Inc. Pipework is essentially a simple shell script which allows for more advanced network setups like configuring multiple IP addresses in containers, using Mac VLAN devices or even using DHCP for IP address allocation. There is even a Docker **image**, which delivers pipework in a container. This provides an opportunity to integrate it into your application via `docker compose`.

Another very useful tool is called **weave** by a company called **Weaveworks**. Weave probably deserves a separate chapter, however, we will provide a simple intro and various links you can explore it further. Weave can create an overlay network across multiple hosts and cloud providers and thus easily connect the Docker containers without any hassle. Weave also provides some really powerful features like **enhanced security** and already mentioned `weave-dns` for easy DNS based service discovery. Weave is arguably the easiest tool on the market to use based on our experience, since it does not require the user to think too much about the low level networking details. The company has recently announced some really interesting features like weave **scope**, which is a tool that provides better container visibility which can be priceless in high density environments. You can read a quick introduction to scope on the following **post**. With the **announcement** of version 1.0 comes big news of a **fast network path**, which is built on top of a previously discussed OVS. Using the fast path means that you no longer have the strong cryptography, but it's a trade off worth considering. Finally, with the recent announcement of **docker plugins**, the native Docker integration got much easier as you can use weave simply as a Docker **plugin**.

The team at **CoreOS** developed their own overlay networking solution called **flannel**. Flannel was originally developed to address Google Cloud Platform centric networking nature of Kubernetes, but the project has evolved since the first release and it now offers

some really interesting features like **VXLAN**. Flannel stores its configuration in **etcd**, which makes the integration with the CoreOS operating system an easy ride.

Finally, the latest player on the field of Docker networking is **project calico**, which offers layer 3 solution to Docker as well as Open Stack. At the moment the project takes advantage of the ClusterHQ **powerstrip**, which allows for simple plugin registration before the native plugin is implemented in Docker engine. The big advantage of project calico is native IPv6 support and easy scaling across multiple Docker hosts. Calico achieves this by implementing some sort of distributed BGP router across multiple Docker host machine. This is a project to keep an eye on.

By listing the third party tools we will now conclude the Docker networking tour. Hopefully the topics described here helped you to get a better understanding of how you can model network in your own Docker infrastructure.

We will now move on to discuss how you can schedule Docker containers across multiple hosts in the next chapter.

13 Scheduling

So far we have looked at Docker running on a single machine. At some point your infrastructure will start growing beyond one host or you simply have a requirement to run Docker containers in some high availability setup. You need to start scaling out to multiple machines. This brings a lot of challenges not only around multi-host docker networking but even more importantly around container orchestration and **cluster** management.

Without proper clustering or orchestration tools you will either find yourself fighting the container infrastructure, as opposed to managing and operating it, or you will simply end up writing a lot of container management tooling yourself. Indeed, Docker has recognized this need and invested a significant effort into reworking its **networking model** and creating the Docker native cluster manager called **swarm**.

The topic of cluster management would certainly deserve a separate book. Among many issues it covers adding and removing hosts, making sure they are healthy, and scaling up and down to manage load. In this chapter we will focus only on one particularly important aspect of cluster management and orchestration: **scheduling**.

The basic idea of job scheduling is simple, and goes back a long way to mainframes and high performance computing (HPC); Rather than dedicating computers to particular workload(s) you treat the entire datacenter as a large pool of compute and storage resources, and just set computing jobs running, as if it is one huge computer. Indeed Mesosphere, one of the infrastructure players in the Docker space, offer "Datacenter Operating System" as their main product.

Let's take a step back now from the concrete implementations and focus a little bit on the underlying problem. In order to understand the problem of container scheduling across a cluster of Docker machines, we need to lay some groundwork and define what scheduling actually means.

What is scheduling?

From a 30 thousand foot view the scheduling problem can be defined as assigning some computing tasks to a set of hardware resources (CPUs, memory, storage and network capacity for example) that can complete them while satisfying the task requirements.

There are a lot of factors that can influence the scheduling decisions. Cost obviously comes to play, but so does latency, throughput, error rates, time to completion and measures of quality of service. So tasks need to be prioritized for example as important (customer facing), or less time critical jobs that can be scheduled as load fluctuates.

Each task may also be made up of a number of processes, with different data storage and networking requirements that may require certain bandwidth or locality, so they need to be placed near each other, and also redundancy requirements that may require placement of replicas far away, such as an another data center or availability zone.

Applications have varying levels of resource requirements over time as the load changes, so they may need to be rescheduled as they scale up and down. In addition there will be new applications deployed over time, so the scheduling decision must be re-examined regularly. Some tasks are only short lived, while others are long running and may perhaps benefit from being moved over their lifetime.

Measurement and monitoring are the keys: throughput and latency versus requirements, resource usage and capacity planning are important, as well as cost and budget, and the value that is being generated. Often the solution to scheduling problem falls into a category of the **optimization problem** when you are trying to optimize for some end goal (such as maximizing throughput, minimizing latency, and so on).

We take scheduling on a single machine for granted most of the time, since operating systems generally do a good job, having been tuned for decades. Scheduling across multiple machines is much less mature.

Unlike scheduling virtual machines, containers are not a fixed size; for example they may change their memory usage over time. This makes the scheduling problem even much harder than the **bin packing** problem of fitting a few known size Virtual Machines (VMs) into bare metal machines.

Optimal scheduling is thus a very difficult problem, and it is only at very large scale that significant engineering efforts may be needed.

Strategies

Like we mentioned earlier, compared to the scheduling problem on a single computer, scheduling across a cluster is much more complex. The scheduling on a single computer concerns running a large number of threads and processes on a small number of CPUs, and the aim is to avoid starvation, where one process does not run for a very long time, and to make sure that deadlines for interactive processes are hit. Beyond that, some notions of fairness in resource use, such as IO bandwidth, are used. Locality is accounted for a little, in **NUMA** systems, but it is not a major driver.

In a datacenter, or multiple datacenters, or in the cloud, the scale of the problem is a few orders of magnitude larger. Latency becomes hugely more important, computers are failing while running jobs, and the timescales are longer. In addition, workloads vary a lot, from Hadoop **MapReduce** jobs, a workload which has usually expected to own a whole

cluster, through **MPI** that expects certain amount of resources to be available before the work can even start, to traditional web applications, which used to just be given more capacity than they need.

So, scheduling containers on a cluster is a hard problem that involves both with heterogenous environments and heterogenous workloads. As always there is no "One size fits all" solution so a whole suite of schedulers have been created by various organizations, most of them addressing a particular workload problem from different angles.

Mesos

One particularly interesting solution developed by researchers at the **University of California, Berkley** that has been creating a lot of waves recently is called **Mesos**. On a very high level, Mesos enables what is referred to as two-level scheduling in which the Mesos itself provides abstractions of raw computing resources in the compute cluster and offers [or schedules] them to arbitrary number of [heterogenous] frameworks running on top of it. Each framework then implements its own task scheduling based on its own particular workload.

This means that Mesos merely manages the cluster resources and decides which ones to offer to which framework based on the **Dominant Resource Fairness** algorithm. What is important is the actual task scheduling is delegated to particular framework, which has its advantages and disadvantages.

This model gives you a flexibility to write your own task scheduler tuned for your particular workload without worrying about low level compute resources. On the other hand, you have no overview of the overall cluster state, and neither can you force Mesos to allocate more resources if you need them at some point in time than what Mesos assumes is a fair share. This can be based on some criteria you specify to the Mesos allocator when the framework starts.

This can get tricky if you are running a lot of long running tasks that can hog up the compute resources and starve out the yet to be scheduled tasks. In other words: Mesos performs the best when used in homogenous batch processing workloads that is very well manifested by success of the projects like **Apache Spark**, which implements a super fast data processing framework running on top of Mesos.

The actual computing tasks scheduled by Mesos frameworks running on top of Mesos are performed by what Mesos refers to as process containerizer. Since version 0.20 Mesos supports Docker as one of the containerizer options. While the full Docker support has not been fully implemented yet, the feature coverage is big enough to schedule complex Docker tasks.

Twitter has proven that you can use Mesos to run large scale infrastructure with heterogenous workloads successfully. They have developed the **Aurora** framework for long running processes that support preemption and many other useful features. Another very

popular framework is **Marathon** developed by **Mesosphere**, which even provides a remote REST API for easier orchestration.

If you are looking to find more deatiled information about Mesos you should read the actual **white paper**.

Kubernetes

Kubernetes is a much younger product than Mesos, only having been started in June 2014, and the 1.0 release, which was released in July 2015. However, it has a longer history, being created by Google as an open source version of its internal scheduling systems, which started with Omega and then Borg.

The paper published by Google on **Borg** gives a great overview of that history, and how Google uses scheduling in its internal, container based, architecture. This heritage means that there has been a lot of interest in Kubernetes, and it is being developed and used by companies such as CoreOS and RedHat.

Kubernetes covers the whole scope of application deployment, scheduling, updating, maintenance, and scaling. It supports Docker containers, grouped into small clusters called "pods" that share a network namespace, so can communicate more simply among themselves; this is meant for a small group of containers that together make up a single application, such as a web application and a Redis store.

The Kubernetes API lets containers, pods, replication, volumes, secrets, metadata and all the other components be controlled, and in many ways is the key strength of Kubernetes; the API design comes from Google's earlier projects so it is already well tested.

Kubernetes is not yet the easiest thing to install. A very clear tutorial is **RedHat's getting started with Kubernetes guide**, but there are a lot of manual steps. Docker may reduce the amount of config management you have to do, but setting it up as a scheduling environment still requires a lot.

Right now the easiest way to try it out and get a feel is to use the Google Container Engine implementation. There is the obligatory **Wordpress hello world** tutorial, which is very quick and easy, at which point you will have the standard "kubectl" Kubernetes command line tool installed and configured, and can run your own Docker containers easily in your cluster.

OpenShift

Scheduling systems are a basic building block for a platform as a service, and Kubernetes is becoming a key building block in this space.

RedHat's **OpenShift** product is an open source (and hosted) platform as a service product. Prior to version 3 it was a fairly standard open source "Heroku clone" PaaS, but **version 3 is a complete rewrite**, based on Docker and Kubernetes in particular, and **Project Atomic**, RedHat's answer to CoreOS, as well as a whole host of other open source projects.

OpenShift 3 was released for general availability in June 2015, and provides another entry point to Kubernetes if you want a fully fledged PaaS.

Thoughts from Clayton Coleman at RedHat

To get a feel for how OpenShift fits in with containers we interviewed Clayton Coleman, the lead engineer for OpenSHift at RedHat.

Is OpenShift v3 an opinionated product or a bunch of open source projects that work together and can be swapped out? Obviously to some extent it is both, but I am interested in how this dynamic of product versus tools works with your clients and your development process.

Clayton Coleman: "OpenShift is a collection of deployment, build, and app lifecycle tools build on top of Kubernetes. We try not to be axiomatic about particular technologies for the individual components. For instance, for edge routing and proxying we are writing in a way that is compatible with Apache, Nginx, F5, and many other configurable load balancers, but we ship a default HAProxy setup. We expect to receive build inputs from many sources, but we primarily enable Git. Kubernetes is gaining support for the Rocket container engine, so OpenShift would be able to leverage that instead of Docker eventually. We try to be the glue between many different concepts and developer workflows rather than assuming a total top to bottom application lifecycle."

How is PaaS being introduced into the enterprise? Who are the early adopters, what kinds of decisions are being made? Have containers and the Docker hype changed speed of adoption? Obviously it is a big change, and the processes of change are interesting.

Clayton Coleman: "PaaS in the enterprise is driven by large shops with extensive internal application deployments looking to standardize their patterns and processes, as well as organizations looking to benefit from increased flexibility in infrastructure and tooling. I think Docker has been more of a bottom up change, being that it's a technology that has a low barrier to entry for simple tasks that it does well. The way we have seen Docker being used in many organizations is in high touch operational flows, where dev/ops teams benefit from the deployment of images as atomic chunks. However, the flip side of that is that they are still heavily custom / scripted flows, so while they can give big improvements, we think the introduction of larger scale patterns (via cluster management / PaaS tools) can deliver a similarly large improvement on top of that (multiplicative improvement, vs additive)."

You described the pod model in Kubernetes as a bit like a mini VM, a collection of applications working together with local communication over Unix sockets and disks for example, rather than remote network IO. Is this a tool for migrating existing applications with this structure, with improved security potentially as per **Redhat's container security page**. So, do you see a migration of existing applciations, or are people largely running new applications on PaaS?

Clayton Coleman: "The mix we have seen is very much split between new and old, so any model that picks one or the other tends to oversimplify in a way that makes the other mode hard. A good percentage of applications are more complex than "just one container",

and the abstractions they typically need are shared disk, local network, or "side-car" style agents that shouldn't be coupled with the main container. So building from day one with the idea that you need a local grouping concept helps both new and old application models."

Immutability, and Project Atomic are obviously very new, how much use of or demand for immutability are you seeing? How do you see this going forward?

Clayton Coleman: "To make immutability work, you need to be able to constantly change in a repeatable way. The flow of new immutable images is totally predicated on an automatable and understandable build pipeline. The benefit of immutability is reducing the number of variables you need to understand to reproduce a problem or triage a failure, but the cost of immutability is automating all the things."

Some other Linux vendors seem to be pushing containers for legacy stuff, i.e. full mutable systems running multiple applciations, while RedHat seems to be pushing for the single app per container structure, with pods around that if necessary. Is that a fair characterization? Is that driven by customer demand?

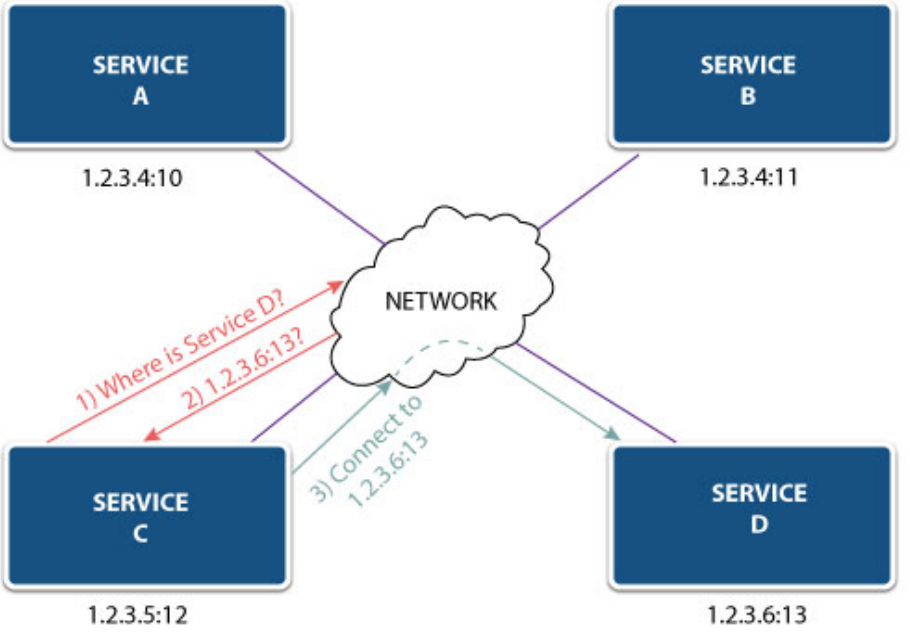
Clayton Coleman: "Occasionally it is useful and necessary to have more complex containers. It's good practice to reduce complexity in general, but there are often real needs that complex containers satisfy that can still benefit from flexible infrastructure around them. I think the benefits of containerized infrastructure depends to a large degree on the ability to split areas of responsibility into multiple components (as people advocated with SOA, or lately micro services). But that should not preclude large, mutable, or complex containers."

In the next chapter we look at service discovery in Docker, which allows any service on a computer network to discover other services it needs to communicate with.

14 Service Discovery

Service discovery is a mechanism that allows any service on a computer network to discover other services it needs to communicate with. It is a key component of most distributed systems. If you are running infrastructures that run or follow **Service Oriented Architecture** (SOA) you can almost never avoid deploying some kind of service discovery solution. The same applies for a new concept in software application design, which shares a lot of similarities with traditional SOA and which has become to be known as **microservices**.

The service discovery problem definition is quite simple: *How does a client discover an IP address and a port of a service it is trying to communicate with?*



Various solutions to this problem often hide a lot of subtleties before users. In order to understand service discovery better we first need to define at least some basic requirements for it. The absolute bare minimum we need from service discovery solution is:

- **service registration/service announcement**: a process in which a service registers its presence on a network. This is normally done by adding a record to some kind of service database, which is often referred to as *service registry* or *service directory*. Service registry entry must contain at least the service IP address and port, but can and often does contain service metadata like protocol, environment details, version, and more.
- **service lookup/service discovery**: a process of querying the network to find out the connection details for the service you are trying to communicate with. This boils down to finding the IP address and port number of given service by querying the service directory database. Ideally, you should be able to query the service directory by different criterions for example by the above mentioned metadata.

The process of service registration is a bit more complex than what we have outlined above. In general, there are two ways in which it can be implemented:

- embed the service registration directly into your application service source code
- use a sidekick process (a.k.a. co-process) which will handle the registration for you

Embedding the service registration into your application source code puts certain requirements on client libraries. Often the availability of the client libraries for a particular programming language is limited so you might end up writing a lot of extra code that might introduce unnecessary complexity in your code base. Embedding service discovery code into the application source code often leads to creating *"thick clients"*, which are not easy to write and often result in debugging challenges. Taking this approach can tie you to a certain service discovery solution that can lead to smaller portability of your applications and services. Finally, integrating with "third party" services that you might need to run your infrastructure, such as `redis`, will require an extra effort. However, if there is a stable and actively maintained solution, you can gain a lot of benefits from it such as client side service load balancing, connection pooling, automatic service heartbeats and failover.

Another widely adopted solution to service registration is to run a sidekick process that runs alongside the application service and performs the service registration on its behalf. The advantages of this solution are obvious. It is very pragmatic as it does not require the application author to write any extra code. By integrating the sidekick process with your init system you can make sure (to certain degree) that the service is only registered when it's fully started and when it operates properly. Taking this approach to service registration also allows for easy integration with third party services. However, the downside is that you often need to run a separate process for every service you need to register. Additionally, using the sidekick process puts extra requirements on the actual service registration process: how do you provide the sidekick process with the configuration of the service you are trying to register? Knowing the service configuration is necessary as the service registration

often requires to update some service metadata. Clearly, using sidekick method provides some advantages but it also introduces some operation and maintenance challenges.

The more complex our infrastructures get the more requirements we put on the service directory database. Comprehensive solutions should offer much more than what we have outlined earlier, in particular at least the following:

- high availability of service directory database
- ability to easily scale the service directory database across multiple hosts while guaranteeing a required level of data consistency
- ability to notify about the service availability so that the service is removed from the directory when it becomes unavailable
- ability to notify about changes to particular service directory entries for example when some service metadata changes

What does this all have to do with Docker you ask? Well, Docker in fact makes the problem of service discovery even more visible. This is especially important when you start scaling your container infrastructure across multiple host machines. Once you start using Docker to package and run the applications you find yourself continuously looking up the IP address and port which the particular service running inside the docker container is listening on. Luckily, Docker makes it quite easy to discover the service connection details: all you need to do is to query the remote API exposed to you by the Docker daemon. Often times users end up writing simple shell scripts and pass the connection details information parsed out from Docker API response to new containers via environment variables. While this can be a handy aproach for local environments, it is arguably not a very scalable and sustainable solution. Not only can the maintenance of custom scripts turn into a nightmare, the problem gets even worse when the application or service instances are spread across multiple hosts or data centers. This problem is even more exacerbated in cloud environments where the services come and go very frequently.

In this chapter we will try to shed a bit more light on the topic of service discovery and discuss various open source solutions we have at our disposal to tackle this problem when running Docker containers. Roll up your sleeves and read on.

DNS service discovery

DNS is one of the core technologies supporting the **World Wide Web**. On a high level, DNS is an eventually consistent distributed database primarily used to resolve human readable names to machine readable IP addresses. It is also well known to be used for looking up email servers responsible for a given domain. Using DNS for service discovery seems like a natural choice - it is a battle tested, widely deployed and very well understood technology. There is a wealth of open source server implementations available for use as well as whole suite of client libraries pretty much in any programming language you can think of. DNS

supports client side caching and a simple domain name delegation which makes it a very scalable solution.

The most well understood part of DNS is the already mentioned resolution of domain names to IP addresses via DNS **A records**. Using just `A records` for service discovery is however not sufficient enough as they do not provide any information about what port a given service listens on. Furthermore `A records` don't provide any metadata information about the services running in your infrastructure. In order to use the DNS to implement full feature service discovery we need to use at least two extra types of DNS records:

- **SRV** - usually used to provide the information about service location on the network, such as the port number
- **TXT** - usually used to provide arbitrary service metadata information i.e. environment, version, and more

These resource records are the bare minimum to use DNS as a service discovery solution. While you can rely on conventions and run the services on well known ports and then simply use `A records` you won't be able to perform more complex lookups should you need to.

Service registration and unregistration requires adding and removing particular DNS records into DNS server configuration, which then often needs to be reloaded in order for the changes to take a desired effect. You must implement a certain level of automation to keep the server configuration up to date in line with the services running in your infrastructure. DNS was created in a fairly "static" Internet world when the need to modify DNS records was not as frequent as it is in the new "cloud era" when servers and services can come and go very frequently. DNS record modification can take some time to propagate due to multiple layers of caching in the DNS infrastructure. Users often try to work around the propagation times by lowering **TTL** value to minimum, which has the effect of creating way too much unnecessary traffic, slows down the communication, and adds an extra load on the DNS servers as the service lookup happen way too frequently.

While there are some well known DNS server **implementations** that are more dynamic with regards to its configuration than the battle-tested **bind** server, often times users are not willing to run their own DNS servers since regardless of the amount of implemented automation, they still require a fair amount of operation and maintenance effort. And while there is a wealth of choice of cloud DNS providers that offer simple API control, like **AWS Route53**, using them will require again an extra effort to write and maintain code, which does not have to be desirable and still does not solve the problem of service chattiness. Indeed, nothing comes for free.

With the dawn of Docker and the *microservices* architecture, a new breed of DNS servers has come to its existence. These DNS servers alleviate some of the earlier discussed issues and are often easily used to provide simple service discovery solution not only for services running inside Docker containers. In the next chapter we will have a look at the most well known implementations and discuss how you can use them in your infrastructure.

DNS servers reinvented

One of the most well known "new generation" DNS server implementations that can be used to implement service discovery in your infrastructure is called **SkyDNS**. You can either compile it from source or deploy it as a Docker container. You can find its Docker image on **docker hub**. The latest version of SkyDNS uses `etcd` service for storing its DNS records. We will discuss `etcd` later on in this chapter - for now, let's just assume that `etcd` is a distributed key-value store.

SkyDNS offers a remote JSON API, which allows you to handle the service registration dynamically by sending a `HTTP POST` request to a given API endpoint. This will create a `SRV` DNS record, which as we learned earlier, can be used to discover connection ports of the services running on the network. You can also set the `TTL` on any value, which will automatically unregister the service once the `TTL` expires. SkyDNS also offers **DNSSEC**. In order to set it up you need to make some extra configuration. Head over to Github to read the project **documentation**.

If you want to use SkyDNS with Docker, you will need to write a SkyDNS client library that will help you handle service registration. Luckily, as mentioned earlier this should not be a big problem thanks to the remote API provided by SkyDNS. Once the service is registered, you can use any DNS library to look it up. This is quite a big win given the abundance of open source DNS libraries. Easier options to use SkyDNS with Docker, although not quite scalable at the moment, is to use **skydock** created by **Michael Crosby** of Docker Inc. Skydock will handle all the Docker container service registration dance for you automatically as it watches for Docker API events: service registration and unregistration are handled automatically for you. The only problem with Skydock is that it is usable only on one Docker host at the moment, however, that might change in the future. Skydock certainly is one of the tools worth to keep an eye on. If you want to find out more about Skydock and how it can used with Docker containers there is a great YouTube **screencast** created by Michael who walks you through all of the Skydock features.

Another player on the Docker DNS server field is **weave-dns**. The biggest advantage of `weave-dns` is that you can use it very easily out of the box with very little effort. Just like Skydock, it watches docker API events and registers containers by adding and removing particular DNS records automatically. While Skydock use is limited to one Docker machine, `weave-dns` works across multiple hosts. However, to take the full advantage of it, you must use it in `weave` overlay network, which can be a bit of downside for some users. Weave network is a **Software Defined Network** (SDN) which gives you a simple and secure overlay network across multiple docker hosts, however, if you are already using a different SDN solution, `weave-dns` might not be the best option for you. Furthermore, at the moment `weave-dns` relies on users using well known ports as opposed to discovering them via `SRV` or `TXT` records.

Given the size and a speed of growth of the Docker ecosystem, there are most likely many more DNS server implementations available to use, some as standalone servers other ones as part of the full blown SDN offering. We will leave those for you to discover as it

would be almost impossible to cover all of them in this book. We will now move on to discuss the service discovery solutions which have become de facto a standard in distributed systems world. We will start with the well known **Zookeeper** project.

Zookeeper

Zookeeper is an **Apache Foundation** project that offers distributed coordination services to distributed systems. It provides *simple primitives* to empower the clients to build more complex coordination functionality. Zookeeper really aims to be a kernel or merely a base building block for building powerful distributed applications. We could write a whole book about Zookeeper, and indeed there already is **one**. Instead we will discuss the basic concepts and focus on how you can use Zookeeper as a service discovery solution in your infrastructure.

On a very high level Zookeeper provides distributed *in memory* data storage registers called `znodes`. These are organized in hierarchical namespaces similar to standard filesystems. Znode hierarchy is often referred to as "data tree". There are two types of `znodes`:

- **regular** - created and deleted by clients explicitly
- **ephemeral** - same as regular with addition of client having the option of delegating their deletion to the cluster automatically once the client session is terminated

Clients can set up a **watch** on any `znode` in the cluster, which lets the Zookeeper notify them about any data modifications or removals automatically. Arguably one of the biggest advantages of Zookeeper is the simplicity of its API; it provides only seven `znode` **operations**. Zookeeper offers strong data consistency guarantees and partition tolerance by implementing **Zookeeper Atomic Broadcast** (ZAB) consensus algorithms. In a very short description, ZAB defines a *leader* and *followers* (which elect the leader). All write requests are forwarded to the leader that then applies them to the system. Read requests can be served by followers. ZooKeeper can only work properly if the quorum (majority) of the servers is correct, therefore you *must* always deploy Zookeeper clusters in sizes of 3, 5 etc. or more formally: you must always run the cluster of `2n+1` nodes (n is a postive number of servers). A cluster of this size tolerates `n` node failures. The earlier described properties have implications on zookeeper's scalability; adding new nodes improves read throughput, but degrades write throughput. Furthermore, write speed decreases when the quorum must wait for leader election votes over the remote site, so if you are thinking of running zookeeper cluster across multiple data centers you should take all of these points into consideration.

Service discovery with Zookeeper

You can implement service discovery with Zookeeper by taking advantage of its ephemeral znodes feature. Registering service would create an ephemeral znode in the cluster under a given namespace on its start and then populate its content with its location on the network (IP address and port). Zookeeper hierarchical namespaces provide a simple mechanism for grouping services of the same kind which is useful if you have multiple instances of the same service running in your infrastructure.

Service registration has to be embedded into the advertising service source code or you can write a simple sidekick service, which will speak Zookeeper protocol and handle the registration on the service behalf. Either way you won't be able to avoid writing code. Clients can discover registered services by looking them up in particular Zookeeper znode namespace. Like we have already mentioned earlier ephemeral znodes exist as long as the TCP session created by the registering service is active. As soon as the service disconnects from Zookeeper, the znode is deleted and the service is unregistered. Clients set up **watch** on a znode they want to be notified about. A watch is **triggered and removed** when the znode changes.

Zookeeper is fully written in Java. The project also provides a comprehensive Java client library to communicate with a Zookeeper cluster. Other client **programming language bindings** exist, however, not all of them provide full feature set and their implementation often differs from one to another which is sometimes confusing to end users. Clients have to handle load balancing between discovered services as well as automatic service failover in case some of the discovered services no longer respond or closed their zookeeper session after the client lookup. If you are using the Java programming language, there is an excellent library which wraps zookeeper client library and provides a lot of extra functionality out of the box. It's called **Curator** and just like Zookeeper it is also an Apache project. You can find a great guide for working with Curator on the following **link**.

Zookeeper can provide a rock solid battle tested solution for service discovery. However, running Zookeeper cluster in your infrastructure introduces a certain amount of complexity which requires some operational experience and carries an extra maintenance cost. Given that Zookeeper provides strong consistency guarantees, when a partition happens services located on a non-quorum side will not be able to register or find other registered services even if they continue to function properly. Zookeeper might not be the best option in the write heavy environments where services come and go too frequently. One of the trickiest problems with using Zookeeper for service discovery, though, is its reliance on the existence of the TCP session created by the registered service. The mere existence of the TCP session provides no guarantees of the service health. Application services can perform a diverse set of tasks. You can't often test the health these tasks by merely checking if the TCP session is alive. Therefore you should **never rely just on the TCP connection liveness** to assume that the service is healthy! This is often underestimated by a lot of users and leads to a lot of unexpected surprises.

Zookeeper shines when used as a building block in your architecture. It can't be easily used directly in your Docker infrastructure, but it is "sneakily" crawling in via other systems which build on top of it. A classic example is **Apache Mesos**, which you can use with to schedule your containers across a Mesos cluster of Docker hosts by using one of its plugins. If you want to use Zookeeper as a standalone service discovery solution you would have to write a simple sidekick client which would run alongside your "dockerised" application service and handle the registration on its behalf. However it's much easier to use some solution that builds on top of Zookeeper such as *Smartstack*, which we will discuss later on in this chapter.

We will now move on to discuss relatively new entrants on the distributed key-value store field which are arguably easier to use as service discovery solutions in your Docker infrastructure than Zookeeper. The first subject of our discussion will be a tool called `etcd`.

etcd

Etcd is a distributed key-value store written in **Go** programming language by **CoreOS**. It shares a lot of similarities with Zookeeper. We will have a look at some of its basic features and then describe how can it be used for service discovery.

Similar to Zookeeper, `etcd` stores data in a hierarchical namespace. It defines the concepts of `directory` and `key` (this is not unique to `etcd`). Any directory can contain multiple keys that are essentially unique identifiers used to look up data stored in `etcd`. Data stored in `etcd` can be persistent or ephemeral. The difference between `etcd` and Zookeeper is the way the ephemeral data is implemented. While in the Zookeeper the lifetimes of the ephemeral data is equal to the lifetime of the TCP session of the client which creates it, etcd takes a different approach which is similar to DNS. Any key in `etcd` can have a `TTL` (Time To Live) value set on it. TTL defines amount of time after which the data set under its key expires and is permanently deleted. TTL can be refreshed any time by the client to extend the lifetime of the stored data. Clients can also set up watch on any key or a directory which allows them to get notified about any changes made to the data stored under them. Etcd watch implementation relies on **HTTP(s) long polling**.

One of the biggest advantages of using `etcd` is that it abstracts away low level data operations by providing a remote JSON API. This is a big win for application developers since they are no longer tied to a particular programming language client implementation. All you need to do to interact with `etcd` is a simple HTTP client (every `etcd` release ships with a command line client utility called **etcdctl**); you can even simply use `curl` to interact with a `etcd` cluster as you can see in many examples in the official **documentation**. Etcd uses the **Raft** consensus algorithm to manage data via replicated log across the cluster. Similar to ZAB, used by Zookeeper, Raft defines leader and follower nodes. All write requests must be applied by leader which keeps log which is then replicated/replayed by its followers.

In order for `etcd` to operate properly it must be deployed in clusters of 2n+1 nodes i.e. 3, 5, 7 etc. You can see recommended production cluster setups on the following **link**. Just like Zookeeper `etcd` provides strong data consistency guarantees and partition tolerance. Etcd also provides a decent **security model**, which allows to use SSL/TLS as well as authentication through client certificates both for the communication between client and cluster and between the nodes in the cluster. Managing the `etcd` cluster can be a bit of a challenge, though. Thankfully, `etcd` provides great **administration** and **clustering** guides that are an **absolute must read** before you deploy an `etcd` cluster in your infrastructure. There is much more to `etcd` than what we have shortly described here, so I would encourage you to explore the official documentation. We will now move on to discuss how to use `etcd` for service discovery.

Service discovery with etcd

You can implement service discovery using `etcd` by taking advantage of its `TTL` feature. A registering service creates a new key entry in the `etcd` cluster and populates it with connection details. The new key can either be created standalone or inside a key directory; directory provides a nice way of grouping multiple instances of the same service running in your infrastructure (directory itself is also represented as a key). The service can then set up a `TTL` on a given key which allows the data stored under it to expire automatically without any further effort to be done by it; `TTL` thus provides a simple automatic mechanism to unregister the service. The `TTL` value can be updated to extend the data expiry time which is exactly what long lived services must do to avoid a continuous (un)registration. Clients discover registered services by querying a particular key or directory in the cluster. Like we have already mentioned earlier, clients can set up watch on any key and be notified about any data modifications stored under the given key.

Thanks to the remote JSON API provided by `etcd` service registration can either be embedded into the service source code or you can use a sidekick process that would implement a simple HTTP client - `etcdctl` or `curl` will do - to interact with the `etcd` cluster. Being able to register a service using a simple command line utility makes third party service integration quite easy: you create a new entry in `etcd` on the service startup and remove the key when the service shuts down. **SystemD** makes starting and stopping sidekick processes along the main service process quite easy.

You can easily take the same approach to service registration when using docker. The sidekick process introspects the service container a populates the particular `etcd` key with the parsed IP address and port. This key can then be read by other docker containers which simply need to query `etcd` cluster.

Using a sidekick process for the service registration can turn into a maintenance headache as you will need to continuously update the `TTL` value as well as monitor the health of the registered service. Users usually implement this via a simple shell script which runs in an infinite loop and checks the health of the running service in certain time interval and

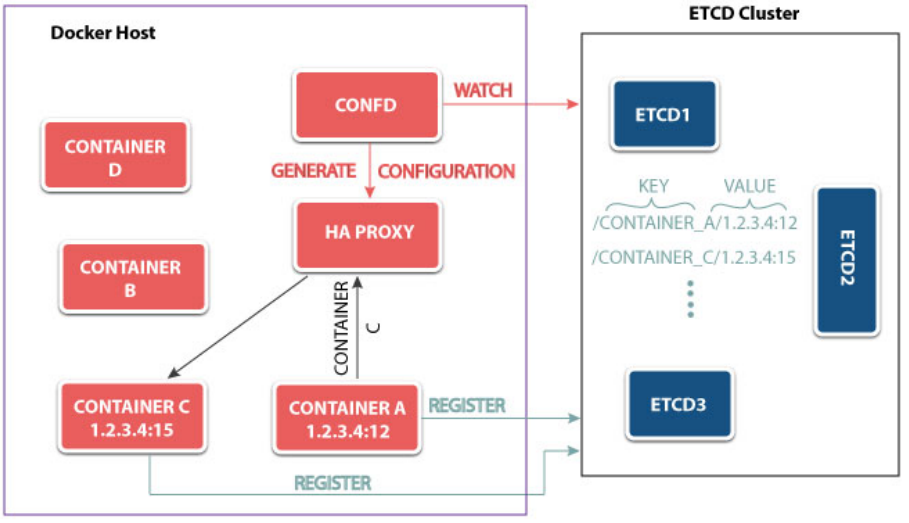
updates particular `etcd` key. You can read about a simple example which takes this approach on the following **link**.

As for the client, there is plenty of tools and libraries in different programming languages available at your **disposal**, many of which are under active development. Again, as it's always the case, the native Go **library** offers a full feature coverage. Unfortunately, it still does not provide service load balancing or failover, so you will need to handle those aspects yourself.

Etcd provides a really good solution for service discovery. It is still under active development, however many companies already use it their production environments. Programming language agnostic remote API interface means a big plus for application developers as it gives them much more freedom. However, just like Zookeeper, `etcd` introduces an extra complexity to your infrastructure. You need to get familiar with how to operate the cluster, which is not an easy feast. Often the lack of understanding of `etcd` internals can lead to unexpected behaviour or even the data loss in some situations.

Consequently, scaling etcd can be a rather challenging task for a newcomer depending on the amount of data you store in the cluster. Service directory records must have their TTLs to be continuously refreshed by the registered service, which requires a bit of extra work from the developer and in the environment where the service lifetime is short it can create a fair amount of traffic. Provided that etcd is the base stone of many other projects like already mentioned SkyDNS or **Kubernetes** and that it ships by default with CoreOS Linux distribution, it's more than likely that the project will continue to **evolve** and certainly improve.

Etcd has become a very popular building block when implementing service discovery solutions in Docker infrastructures. It has inspired a whole suite of new solutions. One many which is worth mentioning was created by **Jason Wilder**. It combines `etcd` with another popular open source software called HAproxy. It takes a sidekick process approach to service discovery and leverages Docker API events, which it then uses to generate HAproxy configuration that routes and load balances the service requests to other services running in Docker containers. You can read more about it on the introductory **blog post**. Again, there is probably tonnes of service discovery implementations that use `etcd`, however Jason's project illustrate the basic concepts very well and defines a service discovery pattern which has been adopted by large in the Docker community. We can see an illustration of it in the following diagram:



In this setup, HAproxy runs directly on the host (i.e. not in a Docker container) and provides a single point of entry for all the services running inside Docker containers. Services running inside Docker containers register with `etcd` when they start by creating a particular key entry. A special process (such as **conf.d**)monitors the key namespace in `etcd` cluster, generates new HAproxy configuration and reloads HAproxy process afterwards. The service requests get automatically routed and load balanced to other services running in Docker containers. This resembles a model championed by Smartstack - another service discovery solution which we will discuss later on in this chapter.

We will leave it to you to explore the Docker ecosystem to find other service discovery solutions built around `etcd`. We will now move to discuss the `etcd`'s younger yet arguably more powerful sibling: **consul**.

consul

Consul is a multifunctional distributed system tool written by **HashiCorp**. Just like the earlier described `etcd`, consul is written in Go programming language. Consul nicely integrates all of its features into a bespoke piece of software which is simple to use and operate. We are not going to spend too much time describing what consul is as you can find an excellent and very extensive documentation with a lot of practical examples on its dedicated **site**. Instead we will summarize its main features and discuss how you can use it to implement the service discovery in your Docker infrastructure. Finally, at the end of this chapter

we will show a practical example using a simple tool, which leverages features provided by consul as one of the pluggable backends to provide hassle free service discovery for applications running in Docker containers.

To describe consul we have chosen to use the word **multifunctional** on purpose as consul really can be used without any extra effort as a standalone tool for any of the following functions:

- distributed key-value store
- distributed monitoring tool
- DNS server

The above features and the ease of use makes consul a very powerful and popular tool in DevOps community. Let's have a quick look at what hides behind the consul's curtains which allows it to be used in so many different ways.

Consul, just like `etcd`, implements the Raft consensus algorithm, therefore it should be deployed in clusters of 2n+1 nodes (n is a positive number of nodes) to guarantee the proper functionality. Equally to `etcd`, consul provides remote JSON API which makes it more accessible for clients written in different programming languages. By providing the remote API consul allows the users to build new services on top of it or leverage the features provided by it out of the box.

In terms of deployment, consul defines a concept of `agent`. The agent can be run in either of the two available modes:

- **server** - provides distributed key value store and DNS server
- **client** - registers services, runs health checks, and forwards queries to servers

Both server and client agents form a cluster. Consul implements cluster membership and node discovery by leveraging another tool written by HashiCorp called **serf**. Serf provides the **SWIM** gossip protocol implementation with some performance improvements. You can read more about the gossip internals in consul in the following **link**. Leveraging the gossip protocol and combining it with local service health checking allows consul to implement a simple but powerful distributed fault detector. This is a huge win for both developers and operators. Developers can expose health check endpoints in their applications and easily add the application services to the consul's distributed service collection. Operators can write simple tools which use consul API to monitor the health of the services or they can simply use the **consul Web UI**.

There is so much more to consul than what we have discussed here. I would **strongly encourage** you to read the fantastic official documentation, as we have barely scratched on the surface. If you are interested in finding out how consul compares to other tools available on the market, don't hesitate to read the **documentation** dedicated to this topic in detail. Let's move on now and see how we can use consul to provide service discovery in our infrastructure.

Service discovery with consul

Of all the already described tools, consul is arguably the easiest one to use as a bespoke service discovery solution. There are several options available to you to register your service into consul's service catalog:

- embed the service registration into your application code by taking advantage of consul's remote API
- use a simple sidekick script/client tool which will register the service on start via remote API
- create a simple service definition **configuration file** which the consul agent reads on start or when it's reloaded

Registered services can be looked up either via remote API or simply via DNS which is provided by consul out of the box. This is handy because you're no longer limited by one particular service lookup option and you get both of the options without any further effort required form you! Furthermore, consul allows you to define custom health checks for your application services. You're not tied to TCP session lifetime like you would be with Zookeeper or to TTL values as is the case with `etcd`. The consul agent continuously monitors the health of the registered services locally and automatically removes it from the service catalogue when the health check fails.

Consul provides a very comprehensive service discovery solution that requires surprisingly very little effort. Services can be looked up either via remote API or DNS. In order to register the services you can avoid using remote API completely and simply use JSON based configuration files; this makes it easy to integrate consul with traditional configuration management tools. Using consul introduces an extra complexity in your infrastructure, but you gain a lot of benefits in return. In comparison to `etcd`, consul clusters are arguably easier to administer. Consul scales well across multiple datacenters - in fact consul provides some extra tools solely dedicated for multi datacenter scaling. Consul can be used as a standalone tool as well as a building block in building complex distributed systems; there is a whole new ecosystem of tools built around it. In the next chapter we will look at one such tool called `registrator`, which provides an easy automatic service registration solution to applications running in Docker containers using consul as one its pluggable backends.

registrator

On a very high level `registrator` listens on Docker Unix socket for Docker container start and die events, and automatically registers the container by creating new records in any of the plugged in backends. It's meant to be run as a Docker container; you can find the `registrator` Docker image on **Docker Hub**. Registrator provides a fair amount of configuration options, so feel free to check them out in the extensive documentation on the Github project **page**.

Let's have a look at a short practical example. We will use `registrator` to make `redis` in-memory database running in Docker container, which is easy to discover on your network via `consul`. You can apply a similar approach to any application service running in your Docker infrastructure.

First we need to start consul container. We will use the image created by the `registrator` creator **Jeff Lindsay**:

```
# docker run -d -p 8400:8400 -p 8500:8500 -p 8600:53/udp -h
node1 progrium/consul -server -bootstrap
37c136e493a60a2f5cef4220f0b38fa9ace76e2c332dbe49b1b9bb596e3ead39
#
```

Now that our backend discovery service is running we will start the `registrator` container and pass it a consul connection URL as an argument:

```
# docker run -d -v /var/run/docker.sock:/tmp/docker.sock -h
$HOSTNAME gliderlabs/registrator consul://$CONSUL_IP:8500
e2452c138dfa9414e907a9aef0eb8a473e8f6e28d303e8a374245ea6cd0e9cdd
```

We can verify that both containers are up and running and we are all set up to register `redis` service:

```
 docker ps
CONTAINER ID        IMAGE                              COM-
MAND                CREATED             STATUS
PORTS
NAMES
e2452c138dfa        gliderlabs/registrator:latest   "/bin/regis-
trator co   3 seconds ago       Up 2 sec-
onds
distracted_sammet
37c136e493a6        progrium/consul:latest          "/bin/start
-server    2 minutes ago       Up 2 minutes        53/tcp,
0.0.0.0:8400->8400/tcp, 8300-8302/tcp, 8301-8302/udp,
0.0.0.0:8500->8500/tcp, 0.0.0.0:8600->53/udp   furious_kirch
```

For the completeness, the following command will show that we are running one node consul cluster and that there are no registered services running at the moment:

```
# curl $CONSUL_IP:8500/v1/catalog/nodes
[{"Node":"consul1","Address":"172.17.0.2"}]
# curl $CONSUL_IP:8500:8500/v1/catalog/services
{"consul":[]}
```

Let's start a `redis` container now and publish all of the ports it exposes:

```
# docker run -d -P redis
55136c98150ac7c44179da035be1705a8c295cd82cd452fb30267d2f1e0830d6
```

If everything went as expected we should be able to find the `redis` service in the consul service catalog:

```
# curl -s localhost:8500/v1/catalog/service/redis |python -
mjson.tool
[
    {
        "Address": "172.17.0.6",
        "Node": "node1",
        "ServiceAddress": "",
        "ServiceID": "docker-hacks:hungry_archimedes:6379",
        "ServiceName": "redis",
        "ServicePort": 32769,
        "ServiceTags": null
    }
]
```

In the above output you can see the format of the service definition used by the `registrator`. You can read more about it in the project **documentation**. As we have learned in the previous chapter, consul provides DNS service out of the box, so all the registered services can be discovered via DNS. We can verify that very easily. First we need to find out what port is the DNS server provided by consul mapped to on the host machine:

```
# docker port 37c136e493a6
53/udp -> 0.0.0.0:8600
8400/tcp -> 0.0.0.0:8400
8500/tcp -> 0.0.0.0:8500
```

Excellent, we can see that the DNS service is mapped to all interfaces on the host and listens on port **8600**. Now we can fire some DNS queries using the well known Linux `dig` utility. From the consul documentation we know that by default the DNS records of the registered services have the form of **NAME.service.consul**. So in our case this would be `redis.service.consul` as `registrator` uses the Docker image name when registering a new service (you can indeed override this if you need to). Let's run the DNS query now:

```
# dig @172.17.42.1 -p 8600 redis.service.consul +short
172.17.0.6
```

We now know the IP address of the `redis` server, but that's not enough information to communicate with the service. We need to find out the TCP port which the server listens on. Luckily that's easy enough. All we need to do is to query consul DNS for SRV record of

the same DNS name. If all goes well we should get back port number **32769** as we could see earlier when we queried consul service catalog via its remote API:

```
# dig @172.17.42.1 -p 8600 -t SRV redis.service.consul +short
1 1 32769 node1.node.dc1.consul.
```

This is fantastic! We have got the full discovery solution running for our Docker containers and all we needed to set it up was to run two simple commands! We didn't have to write a simple line of code.

If we now stop the `redis` container, consul will mark it as stopped and it will no longer return any response back to our queries. This is again very easy to verify:

```
# docker stop 55136c98150a
55136c98150a
# dig @172.17.42.1 -p 8600 -t SRV redis.service.consul +short
# dig @172.17.42.1 -p 8600 -t SRV redis.service.consul

; <<>> DiG 9.9.5-3ubuntu0.1-Ubuntu <<>> @172.17.42.1 -p 8600 -t
SRV redis.service.consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 56543
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDI-
TIONAL: 0

;; QUESTION SECTION:
;redis.service.consul.          IN    SRV

;; Query time: 3 msec
;; SERVER: 172.17.42.1#8600(172.17.42.1)
;; WHEN: Tue May 05 17:59:35 EDT 2015
;; MSG SIZE  rcvd: 38
```

If you are looking for a simple easy to plug in solution to service discovery, `registrator` can provide you with just that with almost no effort, albeit it requires you to run some storage backend like `consul` or `etcd`. However, the benefits of simple deployment and native docker integration outweigh the downsides.

We will now conclude this chapter and move on to the solutions that do not rely on any consensus algorithm that guarantees strong consistency of data, but nevertheless still provide interesting alternatives to implement service discovery in your infrastructure.

Eureka

Over the past few years **Netflix engineering** teams have produced such big amounts of open source tools to help them master their microservices cloud architecture that to make it easy for users to browse they had to create their own **Open Source Software Center**. Most of their tools are written in the Java programming language and very often are not easy to integrate into private infrastructures without reusing many other tools from their OSS toolkit. In this chapter we will have a look at **Eureka** which provides an interesting alternative to implement service discovery.

Eureka is a REST based service that was primarily designed to provide "mid-tier" load balancing, service discovery and failover service. The recommended use case is if you run your infrastructure in AWS cloud, where Eureka was battle-tested, and you have a fair amount of internal services which you do not want to register with AWS ELB or expose to the traffic from outside world. Arguably, inability of ELBs to provide internal load balancing was the primary motivation to create Eureka. Ideally the services discovered via Eureka are stateless as it does not provide sticky session. From architecture point of view Eureka has two components:

- *Server* - provides service registry
- *Client* - handles service registration, and provides basic round robin load balancing and failover

The recommended deployment is to have one Eureka server cluster per **AWS** region, or at least one server per AWS zone. Eureka server does not know about any servers in other AWS regions. Its primary purpose of holding information is for load balancing **within one AWS region**. Servers in the Eureka cluster replicate their service registries between each other in asynchronous fashion. Replication may take some time to reflect in the Eureka servers. This can take sometimes several minutes due to the caching on the server side. In comparison to Zookeeper, etcd or consul, **Eureka favours service availability over strong data consistency**, so there is always a chance that the client might have to deal with the stale reads. This is by design. Eureka focusses on resilience in frequently failing cloud environments. Taking this approach means that Eureka can operate even when the cluster gets partitioned due to some failure but sacrifices data consistency.

Eureka clients register with the server and then renew their "lease" every 30 seconds. If the client does not renew its lease for 90 seconds it is automatically removed from server's registry and must register again. This is similar to a mechanism implemented via `TTL` in `etcd` although with Eureka you don't have a choice of choosing the lifetime of the registry entry. Eureka clients are resilient to server failures. They keep local registry caches, so they can operate reasonably well even if the registry servers fail; this obviously puts some requirements on the client side to deal with service failover. Once the partition heals, local state is merged with the server state. To deal with service failures easily, there is another library in Netflix OSS arsenal called **Ribbon**.

Older versions of client are pull based, however the latest Eureka release brings a lot of improvements both on the server and client side. It separates **write and read clusters**, which improves performance and scalability. Clients can create a subscription to a certain group of services and they can be notified about any changes by the server just like it is the case with the previously described tools. It also provides a simple dashboard and makes deployment to other cloud providers easier. You can read about motivations for the new design at the following **link**.

Service discovery with Eureka

Like we have already mentioned, the design goal of Eureka is client side mid-tier load balancing. In order to load balance between the services you must first look them up in the server registry. Since Eureka is written in Java programming language, it should be fairly easy to integrate the Java client to your application codebase. The native client provides full feature coverage including simple round robin load balancing. Service registers itself with the Eureka server on its start and continuously sends heart beats every 30 seconds. Unregistration happens automatically after 90 seconds of inactivity.

Eureka exposes REST API which allows you to implement your own client easily. There is a few client libraries available in various programming languages, but none of them really offers the quality and feature coverage of the native client implementation. Another option you have is to implement a small sidekick java program running alongside your service which handles the service registration and heartbeats using the native client library. This would mean an extra work and unnecessary maintenance complexity, however you gain a full power of native client library.

The biggest advantage of using Eureka for service discovery is its resilience to failures which makes it a great solution to run in cloud environments as long as you can deal with service failover and stale reads. Indeed, the cloud deployment was the primary driver of its design. Unfortunately, you have no control of the lifetime of service registry entries and must continuously send heartbeats which can cause a fair amount of traffic in your infrastructure and puts a bit of extra load on the registry servers. Clients always retrieve full service list; there is also no filtering or search granularity in service lookups. This will improve in 2.0 release, which also introduces a concept of service subscriptions that you can get server notifications about.

Eureka was designed with auto scaling in mind so it scales reasonably well. Again, the latest release improves the scaling capability even more. The biggest Achilles heel though is that if you want to take a full advantage of Eureka you must use few other Netflix OSS components such as already mentioned Ribbon library or **Archaius** configuration server which in turn depends on Zookeeper. This, as you may have already realized, introduces a lot of often unnecessary complexity in your infrastructure.

We will now move on from the realms of Netflix OSS and discuss a solution different variations of which have become very popular and convenient way of implementing service discovery and which has a rather intriguing name. Meet **smartstack**!

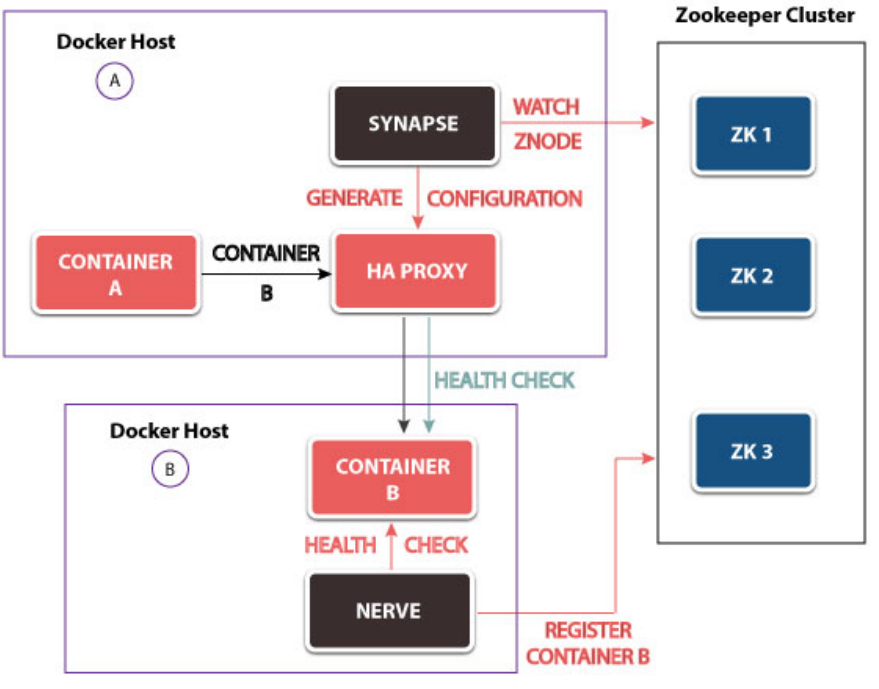
Smartstack

Smartstack is a service discovery solution created by the engineering team of **AirBnb**. Smartstack has a special place in the service discovery ecosystem since its design inspired a whole suite of other solutions. As its name suggests, smartstack is really a **stack** of smart services, which comprises **nerve** and **synapse**.

Nerve and synapse are both written in the Ruby programming language and are available as Ruby gems. They interact with **HAproxy** and the already described Zookeeper. You can read more about the motivations behind creating Smartstack on the excellent introductory **blog post**. In this chapter we will discuss its main features and provide a short summary at the end. Don't hesitate to check the extensive online documentation before you decide to deploy the smartstack in your infrastructure.

Service discovery with Smartstack

Smartstack champions the sidekick process model of service discovery on both registration and discovery side: `synapse` and `nerve` run both as separate standalone processes alongside the application service and handle the service registration and service lookup automatically on behalf of the application service. In production setups running one instance of synapse per host machine should suffice. Smartstack leverages Zookeeper as a service catalogue backend and HAproxy as a single point of entry and a load balancer for the discovered services. Smartstack is quite easy to integrate with your docker infrastructure. You can see a simple architecture using smartstack for service discovery on the following diagram:



Application developers don't need to write any service discovery code and they get load balancing and automatic service failover for free out of the box. Let's have a closer look at both of the smartstack's core services to get a better understanding of how the smartstack service discovery works.

NERVE

Nerve is a simple utility to monitor the health of machines and services. It stores the service health status in some distributed storage. At the moment only Zookeeper backend is fully supported, but there is also an experimental support for etcd. In the context of service discovery nerve handles the **service registration** by creating and removing `znodes` in a Zookeeper cluster based on the service health. If you use `etcd` as a backend storage, nerve will create an K-V entry in `etcd` cluster and set its `TTL` to 30s. It will then continuously update it based on the service health.

Nerve encourages a good practice of service development, which requires the application developers to provide some mechanism to **properly** monitor the service health. This is important not only for reliable service discovery implementation. Nerve leverages the available health monitoring options provided by application service to drive the service

registration process. Finally, nerve can also be used as a standalone service monitoring watchdog without being a part of a service discovery solution.

If you want to find out more about nerve you should head out to Github to read the official **documentation**.

SYNAPSE

Synapse is a simple service discovery implementation that defines a concept of service `watcher` and allows you to watch for events on a given backend. Synapse generates HAproxy configuration file based on the received events. There are several service `watchers` available in synapse:

- **stub** - no watch is used - a list of services is specified manually
- **zookeeper** - zookeeper watches are registered on particular `znodes` in the cluster
- **docker** - docker API events are watched
- **EC2** - watches servers via AWS EC2 tags

Synapse rewrites HAproxy configuration files every time a watched service becomes un/available and reloads the HAproxy process afterward. All client requests are proxied via HAproxy, which takes care of properly routing the requests to the particular application services. This is a win for both application developers as well as operation engineers:

- developers don't need to write any service discovery code
- operators can rely on the battle-tested solution with regards to service load balancing and failover

Again, do check out the official Github **documentation**.

Smartstack is a great technology agnostic solution to implement service discovery in your infrastructure. It does not require to write any extra application code and can be easily deployed on bare metal, virtual machine or in docker containers. However, while particular smartstack components are quite simple, overall the whole setup requires to maintain at least four different pieces of technology: Zookeeper, HAproxy, synapse and nerve. For example, if you aren't already running Zookeeper in your infrastructure you might find it hard to justify running full smartstack setup. Furthermore, whilst running HAproxy in front of your services gives you a nice service abstraction along with load balancing and service failover, managing at least one HAproxy instance on every host machine introduces certain amount of complexity and often requires a fair amount of maintenance.

nsqlookupd

We will finish this chapter with a short description of a tool developed by the engineering team of **bitly** called **nsqlookupd**. **nsqlookupd does not provide full service discovery**

solution, it merely offers an innovative way of discovering instances of **nsqd**, or distributed message queue daemons running in your infrastructure at application runtime.

Service registration is done by the actual `nsqd` daemons that advertise their presence to arbitrary number of `nsqlookupd` instances when they start and then periodically send heartbeats with their status to them.

`nsqlookupd` instances serve as service registries queried directly by clients. They only provide a weekly consistent database of the `nsqd` instances on the network. Clients normally query each available instance and then merge the results.

If you are looking for a distributed message queue solution that can run in a wide range of infrastructure topologies you can read more about `nsqd` and `nsqlookupd` at the project official **documentation**.

Summary

In this chapter we have discussed a number of service discovery solutions. As it always happens, there is no silver bullet and picking the right tool for a job depends on what the job is and what requirements does it have to satisfy. Instead of recommending a particular solution, we will finish this chapter by providing a table overview which, along with the content of this chapter, will hopefully help you decide to make the right decision when picking the best service discovery tool for your infrastructure:

Name	Registration	Data Consistency	Language
SkyDNS	client	strong	Go
weave-dns	auto	strong	Go
ZooKeeper	client	strong	java
etcd	sidekick+client	strong	Go
consul	client+config+auto	strong	Go
eureka	client	eventual	java
nsqlookupd	client	eventual	Go

In the next and final chapter in this book we will examine logging and monitoring in Docker.

15 Logging and Monitoring

When the age of virtualization came about many companies needed a way to monitor their new abstract environments. The industry of monitoring tools needed to be updated to support monitoring a host and guest environment. The same type of life cycle is happening again with the container infrastructure deployments. Logging and monitoring in a Docker environment can be hard to wrap your head around at first but breaking down the environment and using newly available tools will allow you to get the most of this new abstract environment.

The systems and applications we put in containers are no different from the ones we deploy on virtual servers today. For many it can be hard to think about monitoring and logging in a context of another virtual layer on top of a virtual system. Regardless of the complexity and abstraction we still need to setup monitoring and logging to understand the health and performance of our applications as well as for various audit requirements. The biggest issue it seems for companies is the fact that there is a layer of abstraction our current tools lack visibility on. Let's break down logging and monitoring into a couple scenarios so you can be successful at running Docker containers inside your environment.

Logging

You have a couple different options when you need to view logs of a Docker container. The options will vary depending on whether the container is running or is destroyed/removed. When you break down logging within running containers your options are the following:

1. Native Docker logging support
2. Extracting logs by attaching to a running container
3. Exporting logs to the host
4. Sending logs to a central logging system
5. Loading the logs to another Docker container

As for a destroyed/removed container your options are limited. Since containers by default are ephemeral, you'll lose data when they terminate. Losing logging data can be detrimental to an investigation of a failed application. In order to keep logs around after a con-

tainer terminates you'll need to export those logs to a host or network based system from the container.

Native Docker logging

Docker automatically captures the `stdout` and `stderr` of the process running in Docker container and directs in into a given log path on the container root file system: container log path is accessible directly from the host for a user with `root` privileges. You can find out the container log path on the host by running the following command:

```
docker inspect -f '{{.LogPath}}' <container_id>
```

Because of this, running processes inside Docker containers in a foreground mode easily allows you to leverage the native Docker logging capabilities.

As of version 1.6 there are a few log drivers available in Docker out of the box. By default Docker uses `json-file`, which stores the application logs in `json` formatted files in the earlier mention file system path. This is very convenient as many centralized logging solutions require `json` formatted streams for easy search indexing.

You can view the contents of the container log by running the built in Docker client `logs` command. Be aware that running the `logs` command will output all the available logs from the start of the container, so it's better to redirect it to some viewer utility like `more` or `less`:

```
docker logs redis
```

As you will notice, the Docker command line client automatically decodes the `json` encoded files and present them in the plain text format on your screen.

If you are investigating an issue of a running container this is typically the first command to run. This command tells Docker to get the logs of a container named `redis` from the host system where its recorded the logs. Like we said, the container logs by default, if not changed, are logged to the container root file system where the containers files are kept. Keep in mind when running containers in production you'll typically need to rotate your log files or store the log files on a scalable disk store if your application logs a lot of output. If you run the following command you'll get a constant stream of logs as they are written to the file system:

`docker logs redis --follow` (or `-f` for short)

Docker also includes an option within its `logs` command to show the time stamp of a file and limit the size of the viewable lines of logs. You can use the `--timestamps` (or `-t`) command to show the time stamp of the log lines if you need to debug a time sensitive issue. You can also leverage the `--tail` with a following numerical value such as `--tail 10` to show the last 10 lines of output from the containers stdout and stderr.

The `logs` command provided by Docker can only get you so far. Let's say for instance you're running a `redis` database server that logs to `/logs/redis.log` inside the container. Running `docker logs redis` will not show you the logs of this file as

docker only captures `stdout` and `stderr` output. We will explain how to deal with these logs later on in this chapter.

The second log driver option available in docker is `syslog`. When used, Docker sends all the application logs to the syslog running on the host machine. You have to explicitly specify this option when starting the container. In our model `redis` example the command you would want to run would look like this:

```
docker run -d --log-driver=syslog redis
```

If you now inspect the contents of the host machine `syslog` file you will see the contents of the log produced by the above started `redis` container. This is a very convenient log option as many open source log shipping utilities are built to parse and ship the logs from the `syslog`. You can easily reuse these tools in a traditional way by running them directly on the host machine and feeding them the host `syslog` as you would normally in non-Docker world.

Finally, Docker provides the `none` log driver, which discards all of the application generated logs. Again, just like with `syslog` driver, this option must be explicitly enabled when you start the container. While not capturing any logs is not a recommended way of running applications in production environments, it is a convenient option in some setups for some containers which generate huge amount of arbitrary logs and that can cause a real I/O havoc on the host file system.

Attaching to Docker containers

Its typically not recommended to look at log files within the container or the host operating system. However when troubleshooting, you might need to continue your investigation and look at additional log files inside the running container. The next step typically is to attach to a running container in order to continue your investigation. Attaching to a running container requires additional security access (`root` and `ssh`) of a container which can be cumbersome to do repeatedly. However, if you need to do so you can use the attach command that comes with Docker. Here is an example of the command in action.

```
docker attach redis
```

Attaching to the container will attach to the shell of the running process inside the container. When attaching to the container you will be limited to the commands you can run in the container operating system since most images are lightweight to run a single process.

In version **1.3** Docker introduced the `docker exec` subcommand, which allows you to execute any command inside the running Docker container including your favorite shell. This is a better way than attaching to your TTY to the running containers. You can easily run `bash` (if installed in the container image) by running the following command and inspect the logs or any other files on the container file system:

```
docker exec -it redis /bin/bash
```

This will create a new process inside the running container that runs alongside the `redis` process. Once you are inside the container you can look at additional log files that

might not record to `stderr` or `stdout`. This is useful to help debug a failing application or to look at performance based log files. However, this is not a scalable solution to use in production systems. Production systems typically use a centralized logging system to view the streams of logs.

Exporting logs to host

Companies today usually run or use centralized logging systems. In a traditional server environment there is some kind of agent software (e.g. `syslog`) that reads a file system folder or file location for logs and then ships them off to a centralized system. In order to use the centralized logging system you'll need to get the log files out of the container and onto the host system. Getting the logs to the host system with Docker is not hard. There are two common ways you can get logs to record to the host system. You can use the `VOLUME` instruction within the `Dockerfile` image or use the `docker run -v` option to mount a file location from inside the container to a location on the host file system.

The `docker run -v volume` option is the recommended way to use when getting logs to the host system. The `-v` option gives you flexibility on where to redirect the file system output. The `VOLUME` instruction within the `Dockerfile` by default uses the location of the container that is typically a root file system location. If your root filesystem fills up you can run into major issues on your host machine and it could be hard to clean up. For instance, let's take a container configured to run `redis` to log to `/redislogs/redis.log` and get the log files to the host system.

If the `redis` configuration file is set to log the files to `/redislogs/redis.log`, using the VOLUME command in the Dockerfile can easily be achieved by `VOLUME /redislogs`. The `VOLUME` instruction in the image causes docker run to create a new mount point at `/redislogs` location within the container and copy the contents to the newly created volume folder on the host system. The volume folder on the host system is not in a well known location however. #TODO (get example of log location). In this case it will be hard to configure a centralized logging agent on the host to pick up the files in the volume. This is why it's recommended to use the `docker run -v` option for persistent logs.

By using the `docker run -v` option when starting the container you can essentially redirect a folder or file location within the container to a location on the host file system. A common practice is to "redirect" the logs directory within the container to the host system log location like this `docker run -v /logs:/apps/logs`.

When redirecting the logs to a central place on the host machine such as `/logs/apps` you can easily configure a logging agent to pick up all logs from `/logs/apps` and send them to the centralized system. This is also useful when running multiple containers on a host. Let's take for instance you're running `nginx`, `redis`, and a worker container on the same host. By running the following docker commands you'll get three log files in your `/apps` directory:

1. `docker run -v /logs/apps/nginx/nginx.log nginx`
2. `docker run -v /logs/apps/redis/redis.log redis`
3. `docker run -v /logs/apps/worker/worker.log worker`

The logging agent that runs on the host, if configured to read from `/logs/apps/**/*.log`, will pick up all three of the log files and ship them off. This is an easy way to model your container and log agents so your dev/ops teams can easily remember where to look for log files.

Sending logs to a centralized logging system

Sending logs to a centralized logging system is similar to exporting to the host except in one way. You won't redirect the log files to the host system. Instead you can use another method of exporting logs by running a logging agent inside the container alongside the application process and shipping the logs directly over the network. If your operation's team is not against running multiple processes within a container, you can treat your container just like a server by installing and configuring the agent on startup in the container.

Let's take for instance a standard `syslog` setup. Inside the container your application will log to `/logs/apps` and a `syslog` daemon is installed and configured to read any logs from that folder location. It's then configured to ship the logs to a centralized logging server. As new logs are appended to files within that folder the `syslog` daemon reads and ships the logs as intended.

This model is useful, however it has some drawbacks. The drawbacks include running multiple processes inside a container making the container heavier in process utilization, it adds more complexity to the container start scripts and removes the convenience of application isolation. If you have 10 containers on a system running this model you'll essentially have 10 running `syslog` daemons on the host. If these `syslog` agents were consolidated to a single process you might achieve better performance.

Another option to export your logs to a network system is directly from the application. Let's say you have a Java application running inside your container. The Java code could be configured to write any log files to a function within the code. The function could be setup to ship logs to a centralized queue (e.g. **kafka**) on a remote server to be processed by another system later. This option gives developers an easy to configure logging model that could scale to any containerized system. This model however is only available to the application sharing the same Java code with the remote logging function. If another container is running a Python script another remote logging function would need to be written.

Side mounting logs from another container

The last option for getting logs from a container is starting to become popular. The ability to use shared volumes from another container on the system will allow you to pull data (in

this case logs) into another running container. We like to think of this as side mounting logs from another container just like a side mounted seat on a motorcycle. This option could save processing time on a system that has many containers. Let's say if each container on the system runs a service to send logs it would waste resources since every container runs the duplicate service. If you were able to pull all the logs from each container into a single container, then collect and send the logs to a centralized system. You would save a bunch of resources by consolidating the log collector process.

Let's use a simple example of two containers. One container runs a redis server that logs events to /logs/redis.log inside its container on a volume /logs. If we start another container and use the docker run flag of `--volumes-from` you could then pull the /logs volume into the new container. This might be hard to conceptualize so here are some commands.

First we start up a new container named redis. We create a new volume with -v and since we named the container with `--name` it will be easy to pull it from another.

```
docker run -d -v /logs --name redis registry/redis
```

Then we'll start a log collector container to mount the /logs volume from the first container using `--volumes-from` flag. Take note of the `redis` name. `docker run -d --volumes-from redis --name log_collector registry/logcollector`

This will allow a system that runs many Docker containers the ability to use a shared resource. This practice has been found to be common in grid like deployments such as Mesosphere.

Docker allows many different options to collect and view logs. We suggest picking a solution that your comfortable with maintaining and scales for your infrastructure. Companies use the model that works best in their environment. Now let's take a look at monitoring Docker containers.

Monitoring

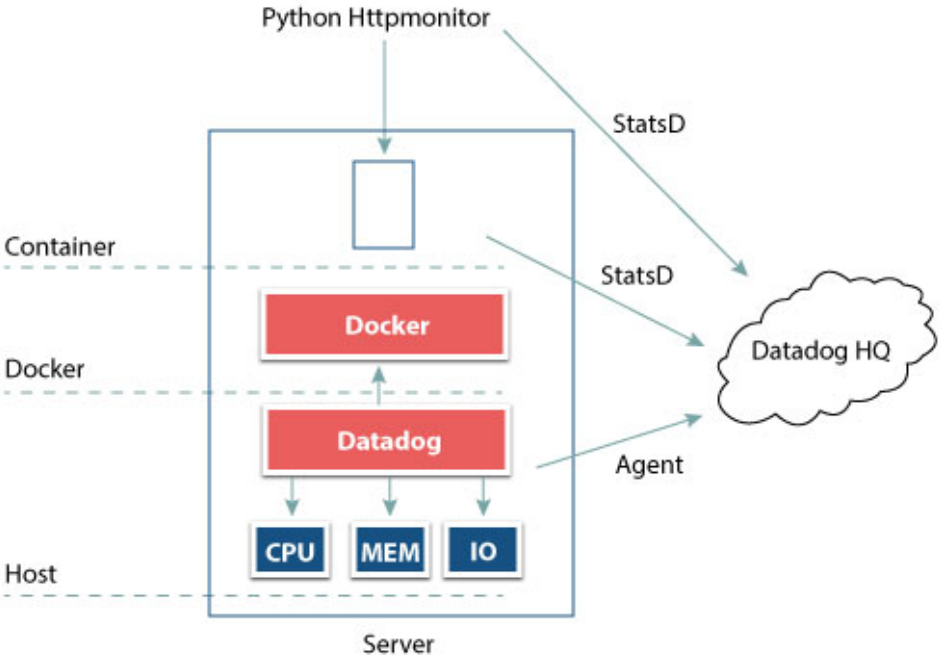
Monitoring Docker containers really comes down to how you want to monitor the services running in the containers and what metrics you need to collect. Your approach to monitoring Docker containers really depends on your current tools and your style of monitoring. Our recommendation is to pick the tools that you are comfortable with and your team enjoys using. Enterprise organizations will most likely have mature tools like Nagios where monitoring a new technology such as Docker might require getting creative. Startups often times start out with the latest greatest technology which might already have Docker support such as New Relic, Datadog, or Sysdig. Either tool will work great in most sceneries so it's just a matter of what your team prefers to use and its effectiveness.

Monitoring can be thought of in a very similar approach to getting logs from your containers. You can take the approach of monitoring the running containers just like a service that's running on the system with an init file. Another option is to monitor the Docker sub-

system (using docker ps or docker stats) and as long as the container is still running it could be considered healthy. Some companies have even put monitoring agents within the container itself just like a traditional server. You could monitor the health of a container by accessing the application layer with a **http://<service>/health** endpoint or some other measure. Finally, if you take the Etsy engineering approach, you will measure everything about your system that you might choose to monitor by collecting metrics on the host, Docker process, and container health.

Many companies who have started to use Docker containers have switched to a Service Oriented Architecture (SOA), which may require a new style of monitoring. We won't go in depth in how to monitor SOA environments, but you should be aware of the the differences. Services in a SOA architecture are typically run in an ephemeral way. Meaning if the running containers would die, thats ok. A service will most likely have multiple containers of the same service running load balanced and if they terminate they will be automatically restarted, re-register themselves to service discovery, and then they'll be backup and running with no downtime. The emphasis of monitoring in a load balanced ephemeral system is typically for application service health and not a service on a server. Most of the time this means the service will be monitored through an outside system at the application layer or port monitor instead which might require new monitoring tools.

There are about a million ways to monitor servers, services, and applications. Were going to walk through a simple environment with a monitoring tool called Datadog, and a simple Python script for HTTP endpoint monitoring. Datadog has a host based agent for monitoring the server and host services along with a statsd backend for application level monitoring. This example environment has a single server running a single Docker container. We'll need to monitor the hosts CPU, Memory, and DiskIO. We'll get some statistics about the Docker process. Then we'll use a Python script running on another system to monitor the application http endpoints which will send metrics to the statsd backend. In the end we'll have end to end monitoring of a system running a Docker container.

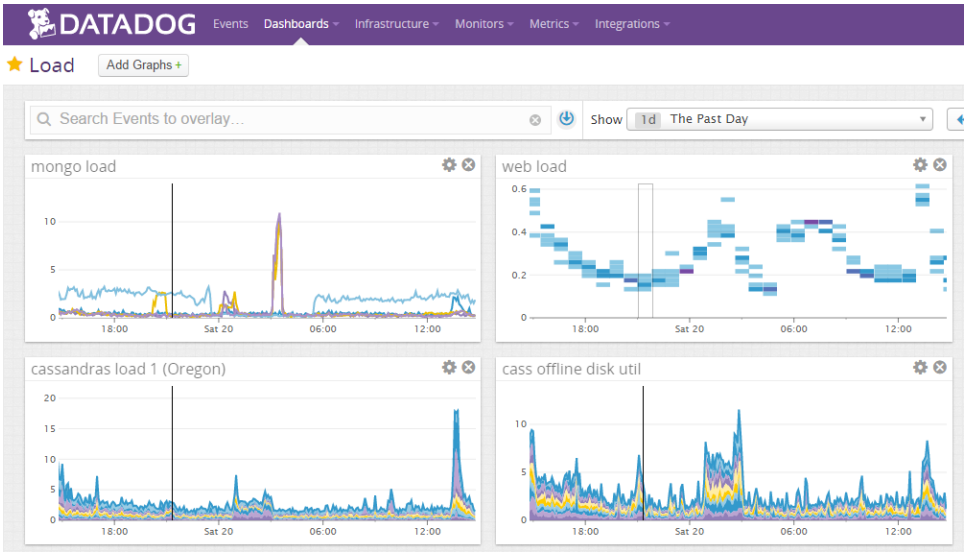


Host based monitoring

Most server environments will have a monitoring system that will measure and alert on CPU, Memory, and DiskIO the traditional metrics used for monitoring. When monitoring the host for these metrics most monitoring systems will work just fine. Since services will now be running in a single process under Docker they won't be easily monitored through the traditional command calls like `ps aux | grep nagios` or `service nagios status`. You'll need to look at your monitoring system to see if it can monitor the Docker processes or pull it's API for health. If it doesn't have Docker integration then you'll need to monitor it yourself if you feel the need to do so.

Monitoring the host with Datadog is pretty straight forward in this example environment. The Datadog agent is just like most traditional monitoring system agents where it installs locally and ships the metrics off to a central system. Much in the same way that Nagios, Zabbix, or Hyperic monitoring agents work. Installing the agent is pretty simple, just follow Datadog's setup instructions to quickly install their host agent (Ubuntu in this example). Here is what we use currently `DD_API_KEY=cdfadffdada9a... bash -c "$(curl -L https://raw.githubusercontent.com/DataDog/dd-agent/master/packaging/datadog-agent/source/install_agent.sh)"`. After the agent is installed, it will start sending data immedi-

ately to the Datadog infrastructure. Within a couple minutes the host will have all your CPU, Memory, and DiskIO metrics available in their metrics explorer and dashboards. Once data is in the system, alerts can be then setup to email or page an employee if needed. Here is an example of data collected in a simple dashboard.



At this point you can only see one third of what most infrastructure teams want to monitor in application environments. We only have visibility into the host metrics like CPU, Memory, and DiskIO at this point. What we don't get is the ability to see individual process or service usage on the system. Most likely if you have a situation where you have high CPU you'll want to know what container, process, or service is taking up the majority of the resources. It will be critical to gather the performance metrics of each Docker container running on the system in order to reduce the mean time to repair (MTTR). Currently on the system we know nothing about the Docker process, the running containers performance, or health. So let's take a look at monitoring the Docker process and how we can achieve gathering that data.

Docker deamon based monitoring

Docker provides several ways to monitor containers and get information out of the system. We'll cover a couple different commands Docker provides to get you started. Then we'll cover how to use Datadog to monitor the Docker deamon and gather container metrics for you. By default Datadog does not collect Docker metrics from their API. You will need to enable the integration on their web site. The integration will gather metrics from Docker so

you can monitor containers at scale. Before we jump into the example let's first look at getting the the state of containers by using `docker ps`.

When running services in Docker containers you won't be able to rely on some well known tools to monitor container processes anymore. By running your services or processes as a container the host system won't have visibility into their health, only the Docker service. Tools for checking if a process is running such as `ps` will be abstracted now under the Docker service that now will require running `docker ps` or `docker info` to get detailed process information now running as containers.

The command `docker ps` is a simple command which returns the status of running containers. Running it is simple. Just type `docker ps` on the system you have Docker installed on. You're not going to get a ton of information about containers, but you'll get the most important metric, container state. Some companies get all they need from just running `docker ps`. It allows you to see if a specific container is running and restart it if its not. A simple example could be running a bash script to monitor for a specific redis container's running state. If the container is not running then start it.

```
#!/bin/bash

# Check for running Redis container, if its not running then
start it
STATE=$(docker inspect redis | jq ".[0].State.Running")
if [[ "$STATE" != "true" ]]; then
    docker run -p 6379:6379 --name redis -v /logs/apps/redis/:/
logs -d hub.docker.com/redis
fi
```

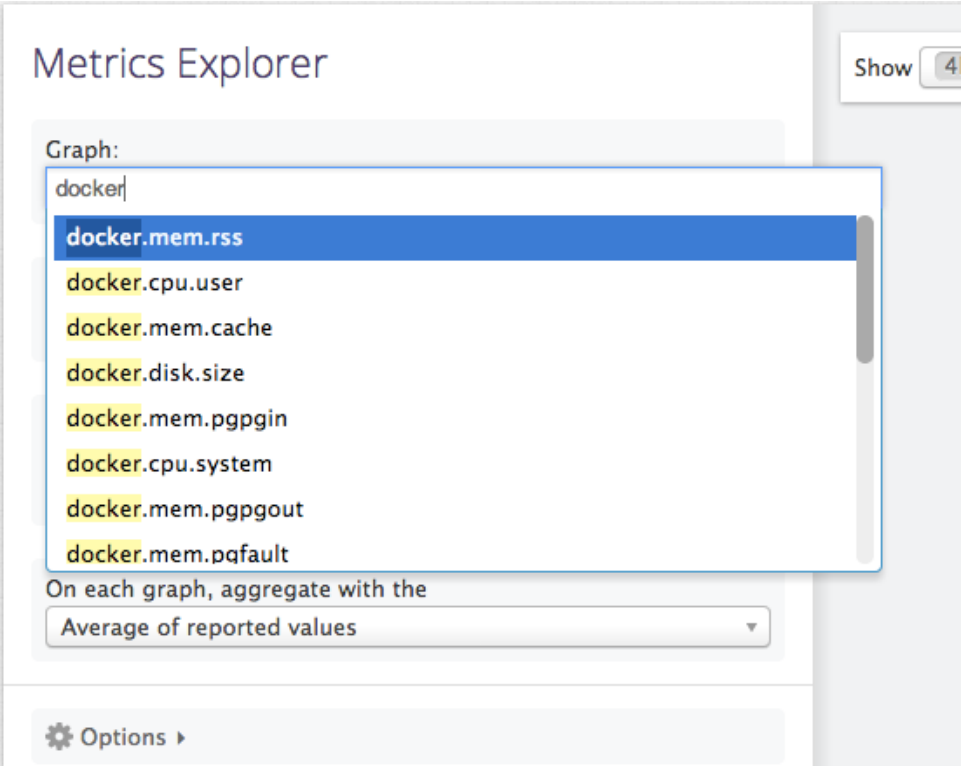
This is a very simple and rudimentary example of how to monitor the running state of a container, but this is how many companies do it today. The orchestration components of most production environments use very similar examples to maintain and automate the health of their systems. As you scale your Docker deployments and have many container running at any given time you'll most likely need better tools and orchestration. If you want to build your own (many companies have already), there are other ways to get the state of the containers other than using bash scripts such as using the Docker python API **docker-py** or using the **Docker Remote API**. You can even use a Docker cluster management system such as **Shipyard** to see the status of multiple containers on multiple systems.

You now have an understanding of getting the state of containers running on your system. We'll need to get metrics about the performance of our containers. Using the earlier example of a system with very high CPU. We'll need to find which container on the system is grabbing the most CPU. For this use case there is another Docker command you should get familiar with. Docker version 1.5 or greater provides a fantastic stats API and command line tool in `docker stats` to get live metrics about your containers such as CPU, Memory, NetworkIO, DiskIO and BlockIO. It's essentially a raw dump of the cgroup metrics some

used before the API was available. This new API is what most monitoring companies are starting to use to get more metrics from the Docker system. You can also roll your own monitoring system using this command. The `docker stats` command gives you a ton of information so you could build your own very sophisticated dashboard to provide exactly what your looking for.

Using the `docker ps` and `docker stats` commands will get you pretty far in monitoring the basic status of your containers. However, when you start to scale containers out beyond a single system you'll need a tool to help aggregate and scale the metrics in an easy to use dashboard. Let's revisit our example of using Datadog to provide the the monitoring at scale.

Datadog's Docker integration uses the Docker daemon socket to reach the Docker API. By using the API, Datadog is able to monitor how many containers are running on the host system, Docker CPU and Memory usage, events on Docker container status changes, uncommon metrics from cgroups, Docker image stats, and more. This can provide a wealth of knowledge on how your container health and performance is on your running containerized services. Datadog does not monitor the Docker API by default so you'll need to enable it on their website. Once you turn on the integration you'll need to make it so the Datadog agent can access Docker by adding it to the Docker group on the system. Once everything is working Datadog will consume the metrics from the system and automatically send data to their infrastructure. At this point you'll be able to setup dashboards and alerts for specific docker metrics you team needs. Here is an example of the metrics you'll be able to view using their metrics explorer from Datadog's **example**.



By using Datadog's Docker integration we're able to see how many containers are running on a single host or even across many hosts, events on container state changes, and even metrics on specific images. Operations and development teams can use this information to now monitor their container level services state and performance. If there is a large change we could notify an operations team of an issue and then alert them of them problem. By using the Docker monitoring integration that Datadog provides you're now able to get the abstracted metrics and information beyond what the normal host level agent can provide. Now that we host based metrics and Docker metrics you can see we have quite a bit of visibility on the system. The only piece were missing is the health of the application itself within the container. We can now move on to the final component of this example.

Container based monitoring

There are a number of ways you can monitor the health of the application inside of your container. Since a container can run any type of application since it in it self is a full blow OS. You could treat your running container just like a server if you wanted to by installing a host base agent inside the container during the image creation process. However, we

strongly recommend keeping your containers as light as possible, a single process even. It's not recommended, but some companies do it successfully and sometimes its easy to integrate into your current monitoring infrastructure. In this example our container on the system is running a simple http process with a /health endpoint. In order to make sure the application health is good inside the container we can monitor it several ways. You can use StatsD or some custom monitoring framework such as a bash or python script. Let's first look at the StatsD approach.

StatsD is a lightweight monitoring tool for easy stats aggregation created by Etsy detailed **here**. It's become a very popular application monitoring service since its light weight and can scale in large environments. One option for monitoring application health inside a container is to send application metrics from the code itself by using a StatsD library. For instance, let's say this web application inside the container gets a POST request to process some credit card data. It would receive the POST request and send a metric to the StatsD server in order to track how many post requests it has received. The server would then process the credit card by connecting to an API at a credit card processor. Right before connecting the code could be setup to record the time. After the call returns and the credit card is processed then the code could again record the time. The code could then take the difference between the recorded times to get how long the processing took and fire off another StatsD metric. When it responds then to the client submitting the POST request a final StatsD metric is sent with a successful metric in order to show the number of total credit cards processed. In this example if all of the StatsD metrics are graphed on a dashboard you can see the health of the application in near real time. You would be able to see how many POST requests the container is getting, processing time to hit the API, and number of completed credit card processes. This would be one way to measure the health of the application within the container. If the dashboard for instance take a sudden drop in completed credit card processes you could then send an alert to get it looked at.

Since we've eluded to a /health endpoint in our example, we'll cover this in some detail. StatsD (and Datadog's implementation) will take any string input along with a metric and then aggregate the stats for you. For instance, if we want to monitor the http endpoint at **http://myservice.corp/health**, we could use `echo "myservice.statuscode:`curl -sL -w "%{http_code}" "http://myservice.corp/health"`|c" | nc -u -w0 statsd.server.com 8125`. This command would check the status code of the health endpoint on the server and then net cat the metric to StatsD. Datadog has integration with StatsD that can then be used to monitor and alert on the metrics you send statsd. In this example if the service was healthy it would return a status code of 200. Datadog would get a metric of myservice.statuscode and the result would be 200. If the status code would return a 500 or 404 error we could then use Datadog to send a failure notice via email or a pager system. Here is an example Python script that utilizes Datadog's StatsD implementation.

```
import requests # For URL monitoring
import statsd # We installed the Datadog statsd module
import sys
```

```
import time

sites = ['http://myservice.corp/health']

def check_web_response_code(url):
    r = requests.get(url,allow_redirects=True,veri-
fy=False,stream=True)
    return str(r.status_code)

def send_dogstatsd(options,site):
    c = statsd.DogStatsd(options.statsd, options.statsport)
    c.connect(host=options.statsd, port=options.statsport)
    statname = 'httpmonitor'
    tags = []
    tags += ['site:'+site]
    r = check_web_response_code(site)
    c.gauge(statname, r, tags=tags)

def monitor_sites(options):
    for site in sites:
        send_dogstatsd(options,site)

def main():
    while True:
        monitor_sites(options)
        time.sleep(30);

if __name__ == '__main__':
    sys.exit(main())
```

This simple Python script can run in a container and monitor the health of many other containers. This is a very simple and easy way to get the application health. Check out the StatsD github project for more details and additional code libraries.

Summary

In conclusion, this example has complete end-to-end service monitoring of a system with Docker and a running container. With a single tool we were able to monitor the server it self, the Docker daemon process, and the container application. By using Datadog, our example can be measured in a consolidated dashboard on Datadog that can also provide alerts on host, Docker process, and application health. This is a very basic example, but we hope it shows you how to get started monitoring your applications running in a Docker infrastructure.

And that wraps up our book. We hope you now have the tools and information to use Docker in a production environment.