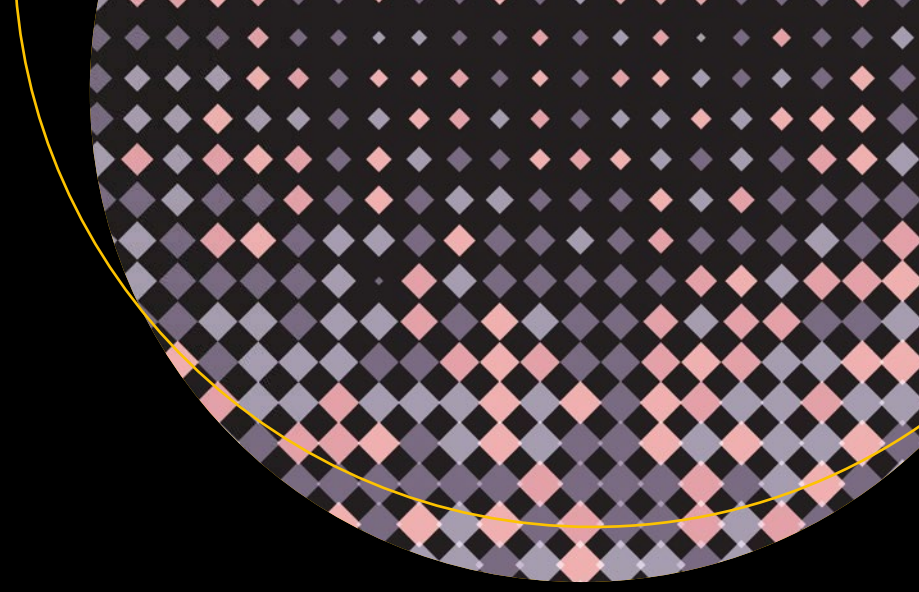

Docker Management Design Patterns

Swarm Mode on Amazon Web Services

Deepak Vohra

Apress®

Docker Management Design Patterns

Swarm Mode on Amazon Web Services

Deepak Vohra

Apress®

Docker Management Design Patterns: Swarm Mode on Amazon Web Services

Deepak Vohra
White Rock, British Columbia, Canada

ISBN-13 (pbk): 978-1-4842-2972-9 ISBN-13 (electronic): 978-1-4842-2973-6
https://doi.org/10.1007/978-1-4842-2973-6

Library of Congress Control Number: 2017955383

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewers: Michael Irwin and Massimo Nardone
Coordinating Editor: Mark Powers
Copy Editor: Kezia Endsley

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484229729. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Contents at a Glance

Contents

About the Author

Deepak Vohra is an Oracle certified Java programmer and web component developer. Deepak has published in several journals, including *Oracle Magazine, OTN, IBM developerWorks, ONJava, DevSource, WebLogic Developer's Journal, XML Journal, Java Developer's Journal, FTPOnline*, and *devx*. Deepak has published three other books on Docker, and a dozen other books on other topics. Deepak is also a Docker Mentor.

About the Technical Reviewers

Michael Irwin is an Application Architect at Virginia Tech (Go Hokies!) where he's both a developer and evangelist for cutting-edge technologies. He is helping Virginia Tech adopt Docker, cloud services, single-page applications, CI/CD pipelines, and other current development practices. As a Docker Captain and a local meetup organizer, he is very active in the Docker community giving presentations and trainings to help others learn how to best utilize Docker in their organizations. Find him on Twitter at @mikesir87.

Massimo Nardone has more than 23 years of experience in security, web/mobile development, and cloud and IT architecture. His true IT passions are security and Android systems.

He has been programming and teaching people how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master's of Science degree in Computing Science from the University of Salerno, Italy.

He worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years.

His technical skills include security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, and more.

He worked as a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (in the PKI, SIP, SAML, and Proxy areas).

He currently works as the Chief Information Security Office (CISO) for Cargotec Oyj and is a member of ISACA, Finland Chapter Board.

Massimo has reviewed more than 40 IT books for different publishers and he is the coauthor of *Pro Android Games* (Apress, 2015).

Introduction

Docker, made available as open source in March 2013, has become the de facto containerization platform. The Docker Engine by itself does not provide functionality to create a distributed Docker container cluster or the ability to scale a cluster of containers, schedule containers on specific nodes, or mount a volume. The book is about orchestrating Docker containers with the Docker-native Swarm mode, which was introduced July 2016 with Docker 1.12. Docker Swarm mode should not be confused with the legacy standalone Docker Swarm, which is not discussed in the book. The book discusses all aspects of orchestrating/managing Docker, including creating a Swarm, using mounts, scheduling, scaling, resource management, rolling updates, load balancing, high availability, logging and monitoring, using multiple zones, and networking. The book also discusses the managed services for Docker Swarm: Docker for AWS and Docker Cloud Swarm mode.

Docker Swarm Design Patterns

"A software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design." (Wikipedia)

Docker Swarm mode provides several features that are general-purpose solutions to issues inherent in a single Docker Engine. Each chapter starting with Chapter 2 introduces a problem and discusses a design pattern as a solution to the problem.

Why Docker Swarm Mode?

Why use the Docker Swarm mode when several container cluster managers are available? Docker Swarm mode is Docker-native and does not require the complex installation that some of the other orchestration frameworks do. A managed service Docker for AWS is available for Docker Swarm to provision a Swarm on production-ready AWS EC2 nodes. Docker Cloud may be linked to Docker for AWS to provision a new Swarm or connect to an existing Swarm. Docker 1.13 includes support for deploying a Docker Stack (collection of services) on Docker Swarm with Docker Compose.

What the Book Covers

Chapter 1 introduces running a Docker standalone container on CoreOS Linux. The chapter establishes the basis of the book and subsequent chapters discuss how the management design patterns provided by the Swarm mode solve problems inherent in a standalone Docker Engine.

Chapter 2 introduces the Swarm mode, including initializing a Swarm and joining worker nodes to the Swarm. Chapter 2 includes promoting/demoting a node, making a node (manager or worker) leave a Swarm, reinitializing a Swarm, and modifying node availability.

Chapter 3 discusses the managed service Docker for AWS, which provisions a Docker Swarm by supplying the Swarm parameters, including the number of managers and workers and the type of EC2 instances to use. AWS uses an AWS CloudFormation to create the resources for a Swarm. Docker for AWS makes it feasible to create a Swarm across multiple AWS zones.

Chapter 4 is about Docker services. Two types of services are defined—replicated and global. Chapter 4 discusses creating a service (replicated and global), scaling a replicated service, listing service tasks, and updating a service.

Chapter 5 discusses scaling replicated services in more detail, including scaling multiple services simultaneously. Global services are not scalable.

In Chapter 6, two types of mounts are defined: a *bind* mount and *volume* mount. This chapter discusses creating and using each type of mount.

Chapter 7 is about configuring and using resources in a Swarm. Two types of resources are supported for configuration: memory and CPU. Two types of resource configurations are defined: *reserves* and *limits*. It discusses creating a service with and without resources specification.

Chapter 8 discusses scheduling service tasks with the default and custom scheduling. Scheduling constraints are also discussed.

Chapter 9 discusses rolling updates, including setting a rolling update policy. Different types of rolling updates are provisioned, including updating to a different Docker image tag, adding/removing environment variables, updating resource limits/reserves, and updating to a different Docker image.

Chapter 10 is about networking in Swarm mode, including the built-in overlay networking called *ingress* and support for creating a custom overlay network.

Chapter 11 is about logging and monitoring in a Swarm, which does not provide a built-in support for logging and monitoring. Logging and monitoring is provided in a Swarm with a Sematext Docker agent, which sends metrics to a SPM dashboard and logs to a Logsene user interface and Kibana.

Chapter 12 discusses load balancing across service tasks with *ingress* load balancing. An external AWS elastic load balancer may also be added for distributing client requests across the EC2 instances on which a Swarm is based.

Chapter 13 discusses developing a highly available website that uses an Amazon Route 53 to create a hosted zone with resource record sets configured in a Primary/Secondary failover mode.

Chapter 14 discusses another managed service, Docker Cloud, which may be used to provision a Docker Swarm or connect to an existing Swarm.

Chapter 15 discusses Docker service stacks. A *stack* is a collection of services that have dependencies among them and are defined in a single configuration file for deployment.

Who this Book Is For

The primary audience of this book includes Docker admins, Docker application developers, and Container as a Service (CaaS) admins and developers. Some knowledge of Linux and introductory knowledge of Docker—such as using a Docker image to run a Docker container, connecting to a container using a bash shell, and stopping and removing a Docker container—is required.

CHAPTER 1

Getting Started with Docker

Docker has become the de facto containerization platform. The main appeal of Docker over virtual machines is that it is lightweight. Whereas a virtual machine packages a complete OS in addition to the application binaries, a Docker container is a lightweight abstraction at the application layer, packaging only the code and dependencies required to run an application. Multiple Docker containers run as isolated processes on the same underlying OS kernel. Docker is supported on most commonly used OSes, including several Linux distributions, Windows, and MacOS. Installing Docker on any of these platforms involves running several commands and also setting a few parameters. CoreOS Linux has Docker installed out-of-the-box. We will get started with using Docker Engine on CoreOS in this chapter. This chapter sets the context of the subsequent chapters, which discuss design patterns for managing Docker Engine using the Swarm mode. This chapter does not use Swarm mode and provides a contrast to using the Swarm mode. This chapter includes the following sections:

- Setting the environment
- Running a Docker application

Setting the Environment

We will be using CoreOS on Amazon Web Services (AWS) EC2, which you can access at `https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#`. Click on Launch Instance to lauch an EC2 instance. Next, choose an Amazon Machine Image (AMI) for CoreOS. Click on AWS Marketplace to find a CoreOS AMI. Type CoreOS in the search field to find a CoreOS AMI. Select the Container Linux by CoreOS (Stable), as shown in the EC2 wizard in Figure 1-1, to launch an instance.

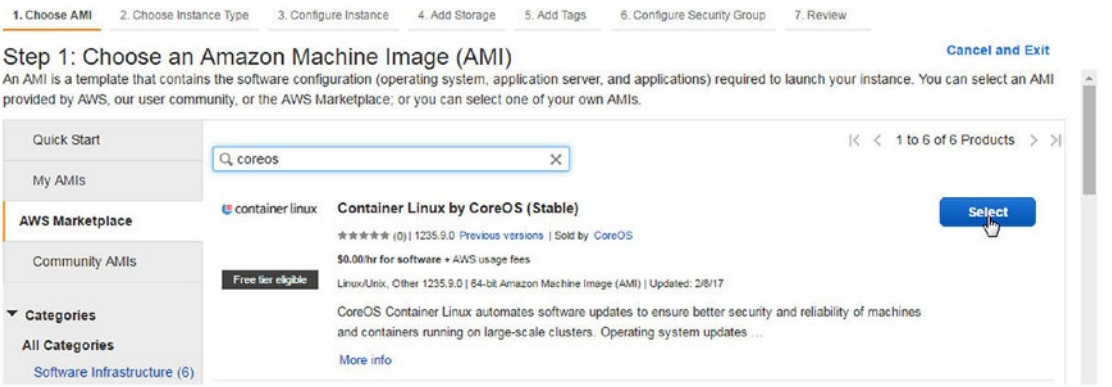

Figure 1-1. *Selecting an AMI for CoreOS Linux*

D. Vohra, *Docker Management Design Patterns*, https://doi.org/10.1007/978-1-4842-2973-6_1

From Choose an Instance Type, choose the t2.micro Type and click on Next. In Configure Instance Details, specify the number of instances as 1. Select a network or click on Create New VPC to create a new VPC. Select a subnet or click on Create New Subnet to create a new subnet. Select Enable for Auto-Assign Public IP. Click on Next.

From Add Storage, select the default settings and click on Next. In Add Tags, no tags need to be added. Click on Next. From Configure Security Group, add a security group to allow all traffic of any protocol in all port ranges from any source (0.0.0.0/0). Click on Review and Launch and subsequently click on Launch.

Select a key pair and click on Launch Instances in the Select an Existing Key Pair or Create a New Key Pair dialog, as shown in Figure 1-2.

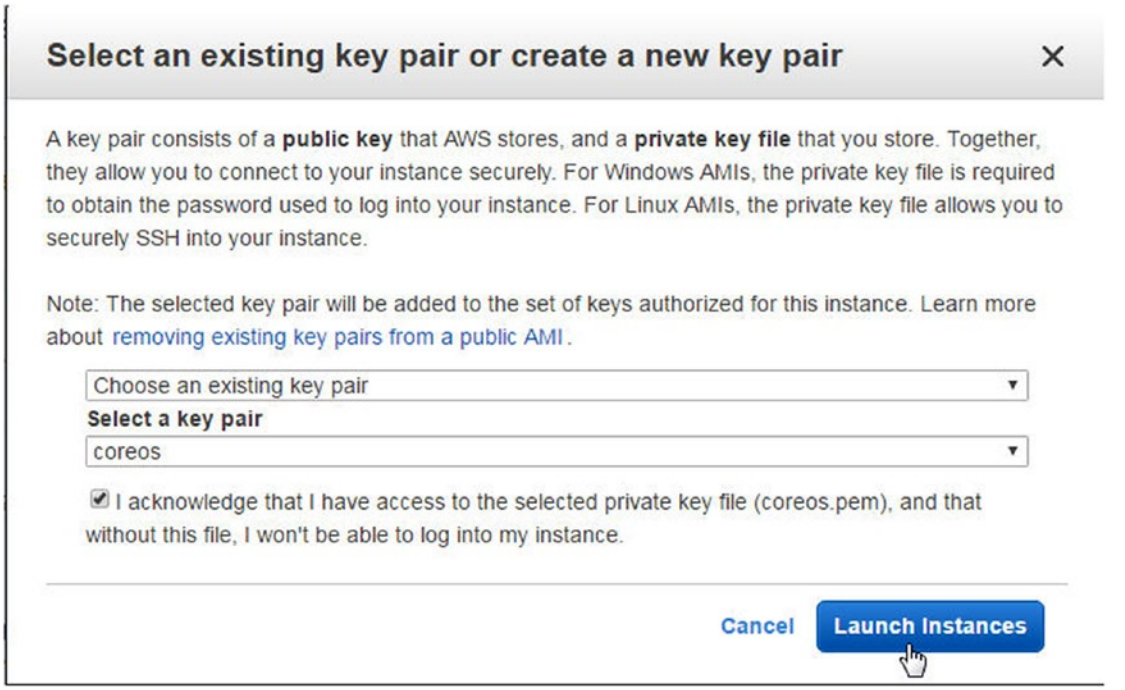

Figure 1-2. *Launch instances*

An EC2 instance with CoreOS is launched. Obtain the public DNS or IPv4 public IP address of the EC2 instance from the EC2 Console, as shown in Figure 1-3, to SSH login into the instance.

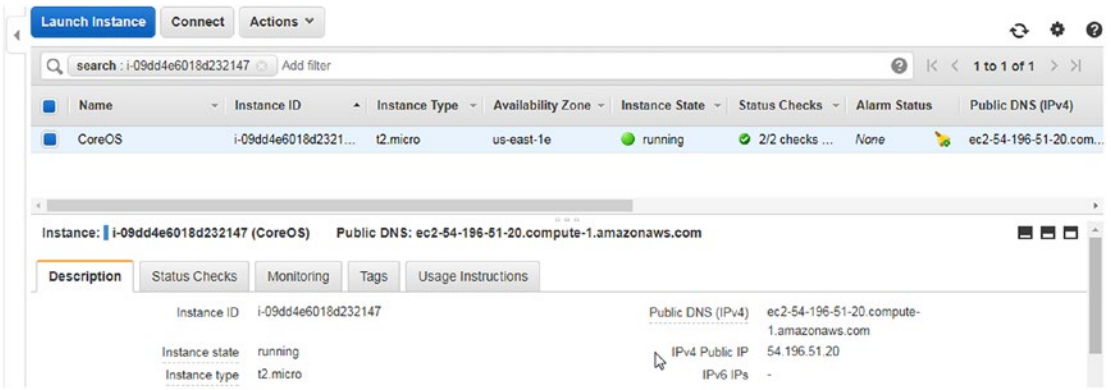

Figure 1-3. *Public DNS and public IPv4*

SSH login into the EC2 instance as user "core".

```
ssh -i "coreos.pem"  core@<public ip>
```

Running a Docker Application

As mentioned earlier, Docker is pre-installed on CoreOS. Run the `docker` command to list its usage, as shown in the following bash shell:

```
core@ip-172-30-4-75 ~ $ docker
Usage: docker [OPTIONS] COMMAND [arg...]
       docker [ --help | -v | --version ]
A self-sufficient runtime for containers.
Options:
  --config=~/.docker              Location of client config files
  -D, --debug                     Enable debug mode
  -H, --host=[]                   Daemon socket(s) to connect to
  -h, --help                      Print usage
  -l, --log-level=info            Set the logging level
  --tls                           Use TLS; implied by --tlsverify
  --tlscacert=~/.docker/ca.pem    Trust certs signed only by this CA
  --tlscert=~/.docker/cert.pem    Path to TLS certificate file
  --tlskey=~/.docker/key.pem      Path to TLS key file
  --tlsverify                     Use TLS and verify the remote
  -v, --version                   Print version information and quit

Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
```

```
    cp          Copy files/folders between a container and the local filesystem
    create      Create a new container
    diff        Inspect changes on a container's filesystem
```

Output the Docker version using the `docker version` command. For native Docker Swarm support, the Docker version must be 1.12 or later as listed in the bash shell output.

```
core@ip-172-30-4-75 ~ $ docker version
Client:
 Version:      1.12.6
 API version:  1.24
 Go version:   go1.7.5
 Git commit:   a82d35e
 Built:        Mon Jun 19 23:04:34 2017
 OS/Arch:      linux/amd64

Server:
 Version:      1.12.6
 API version:  1.24
 Go version:   go1.7.5
 Git commit:   a82d35e
 Built:        Mon Jun 19 23:04:34 2017
 OS/Arch:      linux/amd64
```

Run a Hello World app with the `tutum/hello-world` Docker image.

```
docker run -d -p 8080:80 --name helloapp tutum/hello-world
```

The Docker image is pulled and a Docker container is created, as shown in the following listing.

```
core@ip-172-30-4-75 ~ $ docker run -d -p 8080:80 --name helloapp tutum/hello-world
Unable to find image 'tutum/hello-world:latest' locally
latest: Pulling from tutum/hello-world
658bc4dc7069: Pull complete
a3ed95caeb02: Pull complete
af3cc4b92fa1: Pull complete
d0034177ece9: Pull complete
983d35417974: Pull complete
Digest: sha256:0d57def8055178aafb4c7669cbc25ec17f0acdab97cc587f30150802da8f8d85
Status: Downloaded newer image for tutum/hello-world:latest
1b7a85df6006b41ea1260b5ab957113c9505521cc8732010d663a5e2360975o2
```

List the Docker container using the `docker ps` command.

```
core@ip-172-30-4-75 ~ $ docker ps
CONTAINER ID   IMAGE               COMMAND                  CREATED          STATUS
PORTS                  NAMES
1b7a85df6006   tutum/hello-world   "/bin/sh -c 'php-fpm "   19 minutes ago   Up 19 minutes
0.0.0.0:8080->80/tcp   helloapp
```

The port mapping for the Docker container is also listed using the `docker ps` command, but it may also be obtained using the `docker port <container>` command.

```
core@ip-172-30-4-75 ~ $ docker port helloapp
80/tcp -> 0.0.0.0:8080
```

Using the 8080 port and `localhost`, invoke the Hello World application with `curl`.

```
curl localhost:8080
```

The HTML markup for the Hello World application is output, as listed shown here.

```
core@ip-172-30-4-75 ~ $ curl localhost:8080
<html>
<head>
        <title>Hello world!</title>
        <link href='http://fonts.googleapis.com/css?family=Open+Sans:400,700'
         rel='stylesheet' type='text/css'>
        <style>
        body {
                background-color: white;
                text-align: center;
                padding: 50px;
                font-family: "Open Sans","Helvetica Neue",Helvetica,Arial,sans-serif;
        }
        #logo {
                margin-bottom: 40px;
        }
        </style>
</head>
<body>
        <img id="logo" src="logo.png" />
        <h1>Hello world!</h1>
        <h3>My hostname is 1b7a85df6006</h3>
</body>
</html>
```

Using the public DNS for the EC2 instance, the Hello World application may also be invoked in a browser. This is shown in the web browser in Figure 1-4.

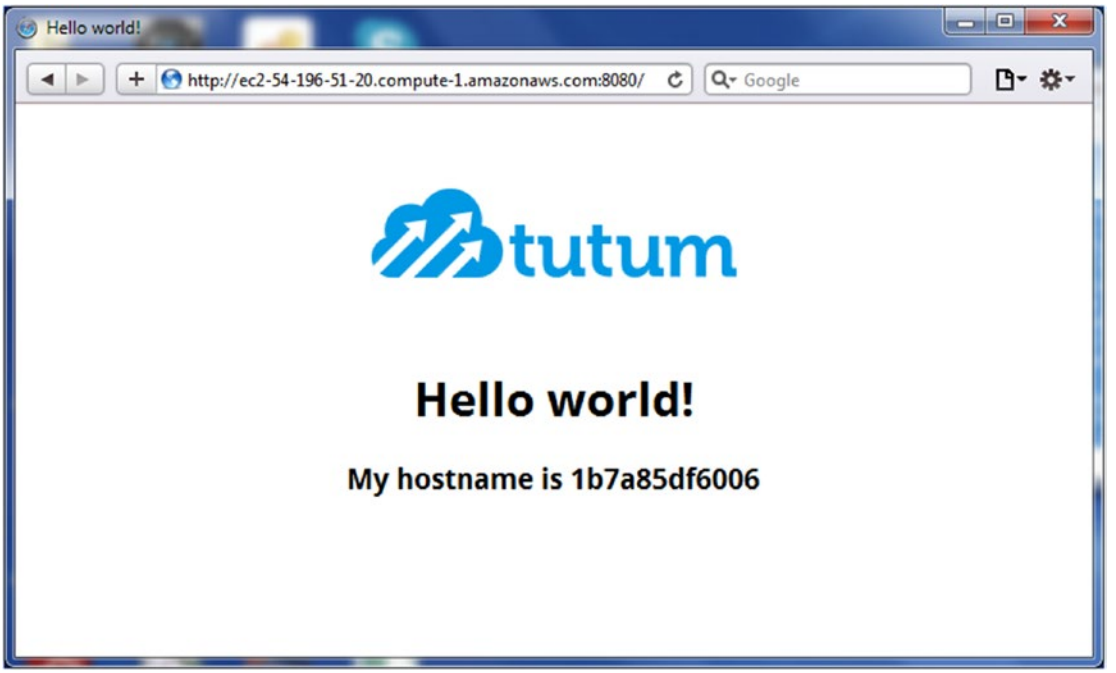

Figure 1-4. *Invoking the Hello World application in a web browser*

The `docker stop <container>` command stops a Docker container. The `docker rm <container>` command removes a Docker container. You can list Docker images using the `docker images` command. A Docker image may be removed using the `docker rmi <image>` command.

```
core@ip-172-30-4-75 ~ $ docker stop helloapp
helloapp
core@ip-172-30-4-75 ~ $ docker rm helloapp
helloapp
core@ip-172-30-4-75 ~ $ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
tutum/hello-world   latest              31e17b0746e4        19 months ago       17.79 MB
core@ip-172-30-4-75 ~ $ docker rmi tutum/hello-world
Untagged: tutum/hello-world:latest
Untagged: tutum/hello-world@sha256:0d57def8055178aafb4c7669cbc25ec17f0acdab97cc587f30150802da8f8d85
Deleted: sha256:31e17b0746e48958b27f1d3dd4fe179fbba7e8efe14ad7a51e964181a92847a6
Deleted: sha256:e1bc9d364d30cd2530cb673004dbcdf1eae0286e41a0fb217dd14397bf9debc8
Deleted: sha256:a1f3077d3071bd3eed5bbe5c9c036f15ce3f6b4b36bdd77601f8b8f03c6f874f
Deleted: sha256:ff7802c271f507dd79ad5661ef0e8c7321947c145f1e3cd434621fa869fa648d
Deleted: sha256:e38b71a2478cad712590a0eace1e08f100a293ee19a181d5f5d5a3cdb0663646
Deleted: sha256:5f27c27ccc6daedbc6ee05562f96f719d7f0bb38d8e95b1c1f23bb9696d39916
Deleted: sha256:fab20b60d8503ff0bc94ac3d25910d4a10f366d6da1f69ea53a05bdef469426b
Deleted: sha256:a58990fe25749e088fd9a9d2999c9a17b51921eb3f7df925a00205207a172b08
core@ip-172-30-4-75 ~ $
```

Summary

This chapter sets the basis for subsequent chapters by using a single Docker Engine on CoreOS. Subsequent chapters explore the different design patterns for managing distributed Docker applications in a cluster. The next chapter introduces the Docker Swarm mode.

CHAPTER 2

Using Docker in Swarm Mode

The Docker Engine is a containerization platform for running Docker containers. Multiple Docker containers run in isolation on the same underlying operating system kernel, with each container having its own network and filesystem. Each Docker container is an encapsulation of the software and dependencies required for an application and does not incur the overhead of packaging a complete OS, which could be several GB. Docker applications are run from Docker images in Docker containers, with each Docker image being specific to a particular application or software. A Docker image is built from a *Dockerfile*, with a Dockerfile defining the instruction set to be used to download and install software, set environment variables, and run commands.

The Problem

While the Docker Engine pre-1.12 (without native Swarm mode) is well designed for running applications in lightweight containers, it lacks some features, the following being the main ones.

- *No distributed computing*—No distributed computing is provided, as a Docker Engine is installed and runs on a single node or OS instance.
- *No fault tolerance*—As shown in the diagram in Figure 2-1, if the single node on which a Docker Engine is running fails, the Docker applications running on the Docker Engine fail as well.

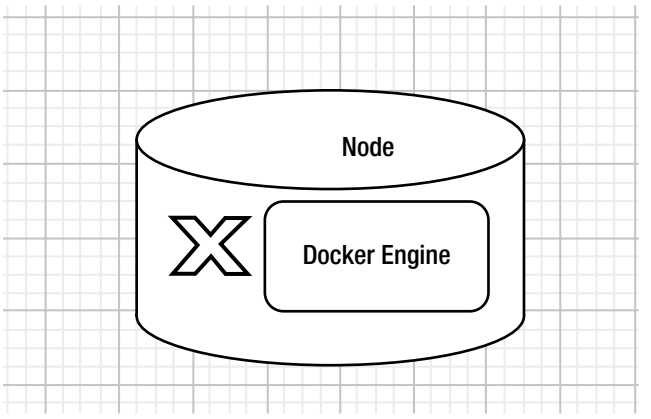

Figure 2-1. *Single node Docker cluster*

D. Vohra, *Docker Management Design Patterns*, https://doi.org/10.1007/978-1-4842-2973-6_2

The Solution

With Docker Engine version 1.12 onward, Docker container orchestration is built into the Docker Engine in *Swarm mode* and is native to the Docker Engine. Using the Swarm mode, a swarm (or cluster) of nodes distributed across multiple machines (OS instances) may be run in a master/worker/ pattern. Docker Swarm mode is not enabled in the Docker Engine by default and has to be initialized using a `docker` command. Next, as an introduction to the Docker Swarm mode, we introduce some terminology.

Docker Swarm Mode

Docker Swarm is a cluster of Docker hosts connected by an overlay networking for service discovery. A Docker Swarm includes one or more *manager nodes* and one or more *worker nodes,* as shown in Figure 2-2. In the Swarm mode, a Docker service is the unit of Docker containerization. Docker containers for a service created from a Manager node are deployed or scheduled across the cluster and the Swarm includes a built-in load balancing for scaling the services. The expected state for a service is declared on the manager, which then schedules the task to be run on a node. However, the worker node itself still pulls the image and starts the container.

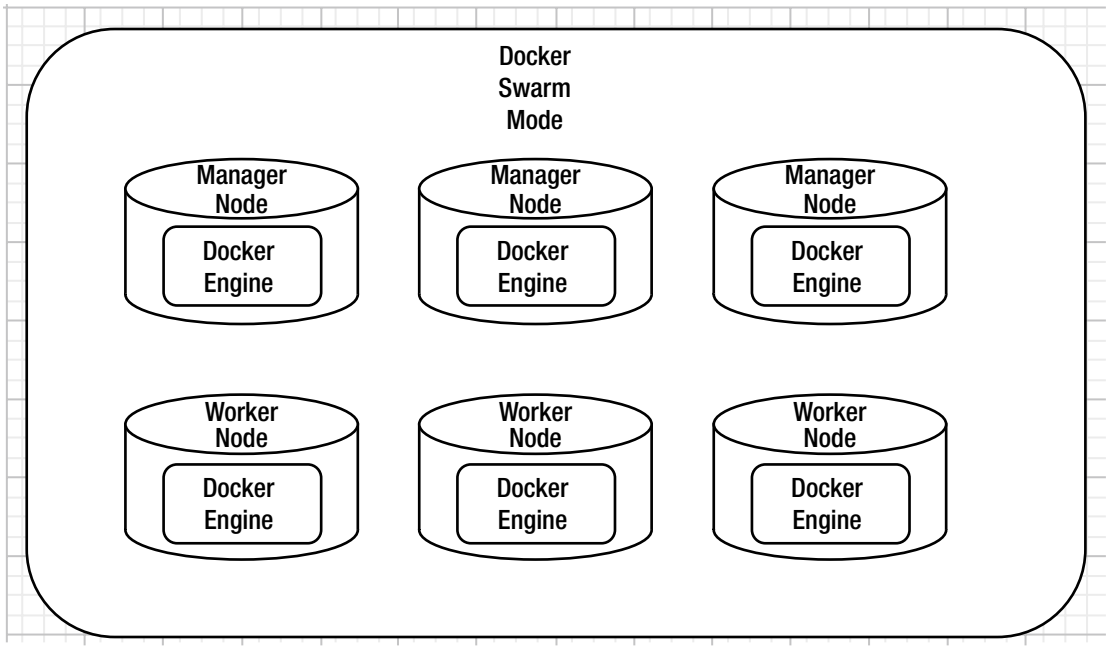

Figure 2-2. *Docker Swarm mode cluster*

Nodes

An instance of a Docker host (a Docker Engine) is called a *node*. Two types of node roles are provided: *manager nodes* and *worker nodes*.

Service

A *service* is an abstraction for a collection of tasks (also called replicas or replica tasks) distributed across a Swarm. As an example, a service could be running three replicas of an Nginx server. Default scheduling, which is discussed in Chapter 7, uses the "spread" scheduling strategy, which spreads the tasks across the nodes of the cluster based on a computed node rank. A service consists of one or more tasks that run independent of each other, implying that stopping a task or starting a new task does not affect running other tasks. The Nginx service running on three nodes could consist of three replica tasks. Each task runs a Docker container for the service. One node could be running multiple tasks for a service. A task is an abstraction for the atomic unit of scheduling, a "slot" for the scheduler to run a Docker container.

Desired State of a Service

The "desired state" of a service refers to the service state as defined in the service definition when creating the service. As an example, a service definition could define a service as consisting of three replicas of an Nginx server.

Manager Node and Raft Consensus

When the Swarm is first created, the current node becomes the first manager node. By default, all manager nodes are also workers. The manager node performs the cluster orchestration and manages the Swarm, including the initial scheduling of service tasks and subsequent reconciliation, if any, between the desired state and the actual state of services. As an example, for a service definition consisting of three replicas of an Nginx server, the manager node would create three tasks and schedule the tasks on Swarm worker nodes in the Swarm. Subsequently, if a node running a task were to fail, the Swarm manager would start a new replacement task on the worker nodes still in the Swarm. The Swarm manager accepts the service definition when a service is created and schedules the service on one or more worker nodes as service tasks. The Swarm manager node also manages the scaling of service by adding/removing service tasks. The Swarm manager assigns each service a unique DNS name and starts Docker containers via service replica tasks. The manager node monitors the cluster state. The Swarm manager is also a worker node by default, which is discussed in the next section.

To refer to "the manager node" is actually a simplification of the Swarm Manager, as a Swarm may consist of one or more manager nodes. Each manager node keeps the complete cluster state data, including which service replica tasks are running on which node and the node roles, and participates in Swarm management for the *Raft consensus*. The Raft consensus is merely an algorithm to create decisions/agreements (consensus) within a group in a distributed fashion. Swarm uses it to make decisions such as leader elections, cluster membership, service changes, etc. In the Swarm mode, Raft consensus is an agreement among the manager nodes for a global cluster state parameter such as about the state of data value stored in a database. Swarm managers share data using Raft. Raft consensus is a protocol for implementing distributed consensus among all the reachable manager nodes in a Swarm. The Raft Consensus Algorithm has several implementations and its implementation in the Swarm mode has the properties typically found in distributed systems, such as the following:

- Agreement of values for fault tolerance
- Cluster membership management
- Leader election using mutual exclusion

Only one manager node, called the *leader,* performs all the cluster orchestration and management. Only the leader node performs the service scheduling, scaling, and restarting of service tasks. The other manager nodes are for the fault tolerance of Swarm manager, which implies that if the leader node were to fail, one of the other manager nodes would be elected as the new leader and take over the cluster management. Leader election is performed by a consensus from the majority of the manager nodes.

Worker Nodes

A worker node actually runs the service replica tasks and the associated Docker containers. The differentiation between node roles as manager nodes and worker nodes is not handled at service deployment time but is handled at runtime, as node roles may be promoted/demoted. Promoting/demoting a node is discussed in a later section. Worker nodes do not affect the manager Raft consensus. Worker nodes only increase the capacity of the Swarm to run service replica tasks. The worker nodes themselves do not contribute to the voting and state held in the raft, but the fact that they are worker nodes is held within the raft. As running a service task requires resources (CPU and memory) and a node has a certain fixed allocatable resources, the capacity of a Swarm is limited by the number of worker nodes in the Swarm.

Quorum

A *quorum* refers to agreement among the majority of Swarm manager nodes or managers. If a Swarm loses quorum it cannot perform any management or orchestration functions. The service tasks already scheduled are not affected and continue to run. The new service tasks are not scheduled and other management decisions requiring a consensus, such as adding or removing a node, are not performed. All Swarm managers are counted toward determining majority consensus for fault tolerance. For leader election only the reachable manager nodes are included for Raft consensus. Any Swarm update, such as the addition or removal of a node or the election of a new leader, requires a quorum. *Raft consensus* and *quorum* are the same. For high availability, three to five Swarm managers are recommended in production. An odd number of Swarm managers is recommended in general. *Fault tolerance* refers to the tolerance for failure of Swarm manager nodes or the number of Swarm managers that may fail without making a Swarm unavailable. Mathematically, "majority" refers to more than half, but for the Swarm mode Raft consensus algorithm, Raft tolerates (N-1)/2 failures and a majority for Raft consensus is determined by (N/2)+1. N refers to the Swarm size or the number of manager nodes in the Swarm.

```
Swarm Size  = Majority + Fault Tolerance
```

As an example, Swarm sizes of 1 and 2 each have a fault tolerance of 0, as Raft consensus cannot be reached for the Swarm size if any of the Swarm managers were to fail. More manager nodes increase fault tolerance. For an odd number N, the fault tolerance is the same for a Swarm size N and N+1.

As an example, a Swarm with three managers has a fault tolerance of 1, as shown in Figure 2-3. Fault tolerance and Raft consensus do not apply to worker nodes, as Swarm capacity is based only on the worker nodes. Even if two of the three worker nodes were to fail, one Worker node, even if the manager nodes are manager-only nodes, would keep the Swarm available though a reduction in Swarm capacity and could transition some of the running tasks to non-running state.

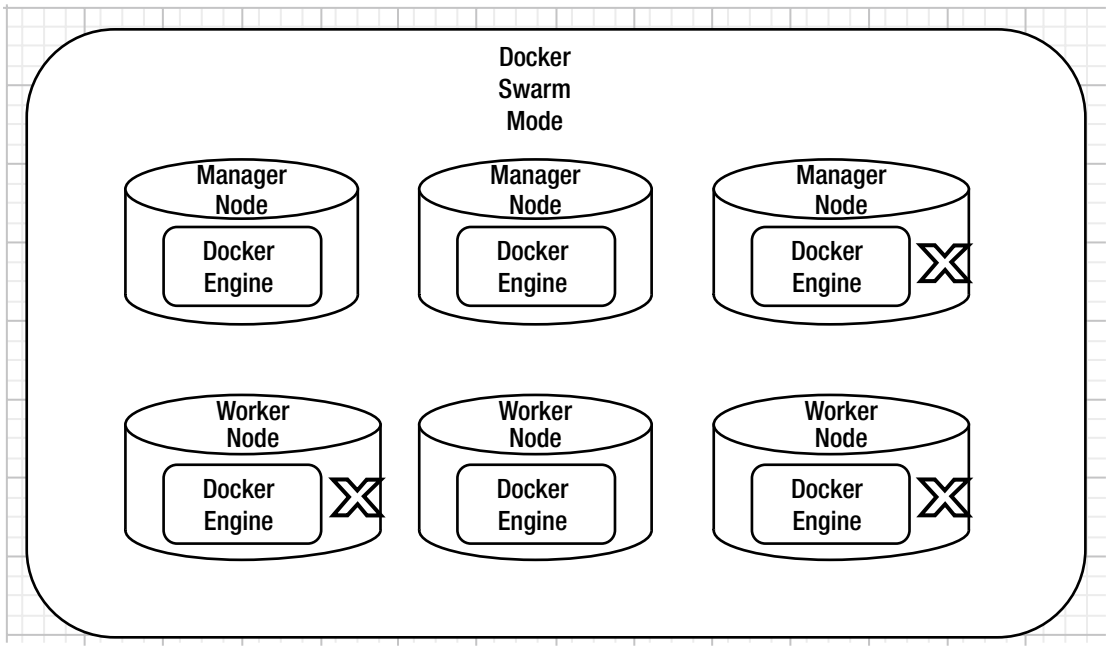

Figure 2-3. *Fault tolerance for a Swarm*

This section covers the following topics:

- Setting the environment
- Initializing the Docker Swarm mode
- Joining nodes to the Swarm cluster
- Testing the Swarm cluster
- Promoting a worker node to manager
- Demoting a manager node to worker
- Making a worker node leave the Swarm cluster
- Making A worker node rejoin the Swarm cluster
- Making a manager node leave the Swarm cluster
- Reinitializing a Swarm
- Modifying node availability
- Removing a node

Setting the Environment

This chapter shows you how to create a three-node Swarm consisting of one manager node and two worker nodes. Create three Amazon EC2 instances using CoreOS Stable AMI, as shown in the EC2 console in Figure 2-4. Enable all traffic between the EC2 instances when configuring the security group for the EC2 instances. Obtain the IP address of the EC2 instance started for the Swarm manager.

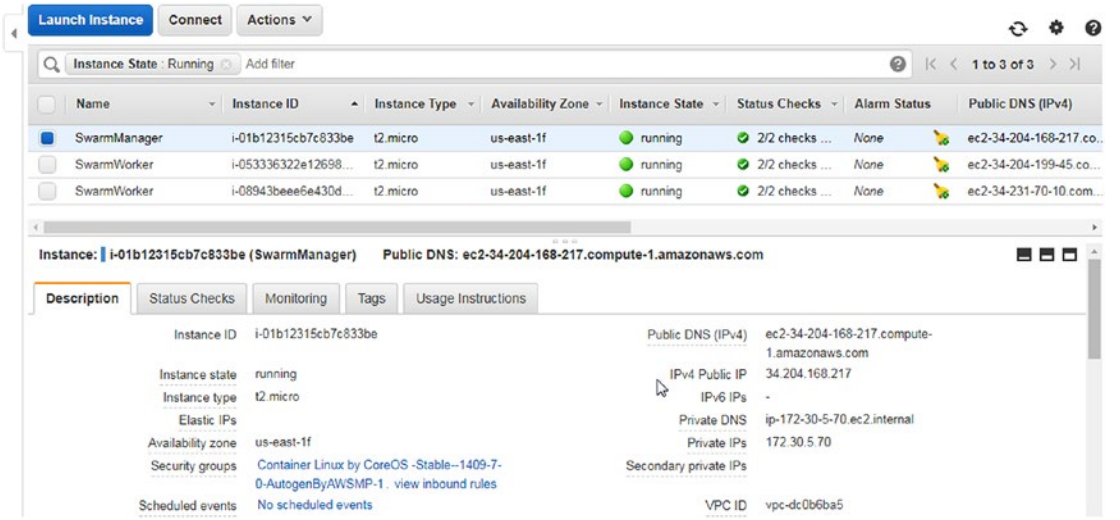

Figure 2-4. *EC2 instances*

Initializing the Docker Swarm Mode

Docker Swarm mode is not enabled by default and needs to be enabled. SSH login to the EC2 instance started for the Swarm manager using the public IP address.

```
ssh -i "coreos.pem" core@34.204.168.217
```

Docker Swarm mode is available starting with Docker version 1.12. Verify that the Docker version is at least 1.12 using the `docker --version` command.

```
[root@localhost ~]# ssh -i "coreos.pem" core@34.204.168.217
Container Linux by CoreOS stable (1409.7.0)
core@ip-172-30-5-70 ~ $ docker --version
Docker version 1.12.6, build a82d35e
```

To initialize the Swarm, use the docker `swarm init` options command. Some of the options the command supports are listed in Table 2-1.

Table 2-1. *Command Swarm init Options*

Option	Description	Default Value
`--advertise-addr`	Advertised address in the format `<ip|interface>[:port]`. The advertised address is the IP address at which other nodes may access the Swarm. If an IP address is not specified, the Docker ascertains if the system has a single IP address and, if it does, the IP address and port 2337 is used. If the system has multiple IP addresses, the `--advertise-addr` must be specified for inter-manager communication and overlay networking.	
`--availability`	Availability of the node. Should be one of `active/pause/drain`.	`active`
`--force-new-cluster`	Whether to force create a new cluster from the current state. We discuss why it may be required to force create and use the option in this chapter.	`false`
`--listen-addr`	Listen address in the format `<ip|interface>[:port]`.	`0.0.0.0:2377`

Use the default values for all options except the `--advertise-addr` for which a default value is not provided. Use the private address for the advertised address, which may be obtained from the EC2 console, as shown in Figure 2-5. If the EC2 instances on AWS were in different regions, the external public IP address should be used to access the manager node, which may also be obtained from the EC2 console.

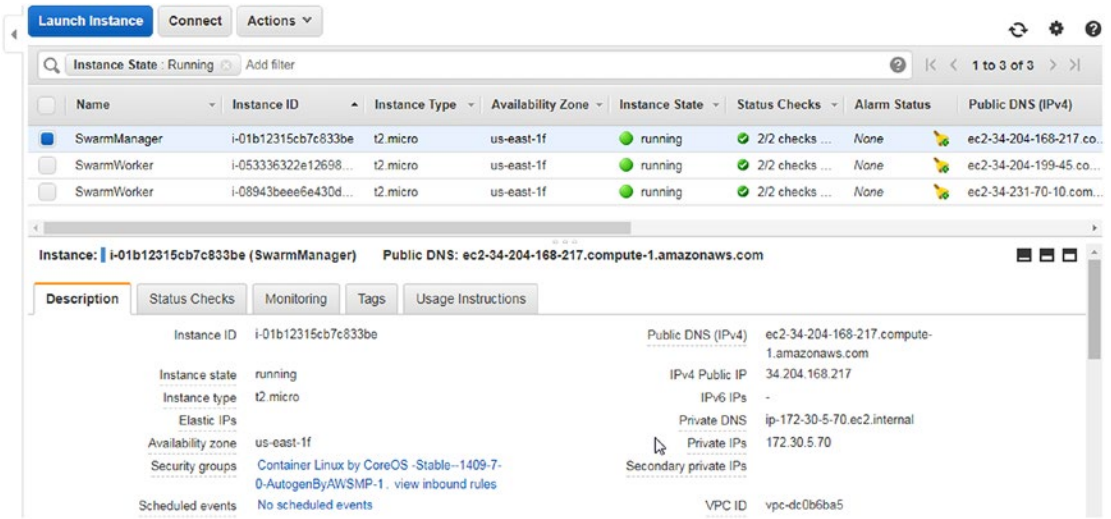

Figure 2-5. *Private IP*

Run the following command to initialize Docker Swarm mode.

```
docker swarm init --advertise-addr 172.30.5.70
```