Docker Deep Dive

Feb 2018

Nigel Poulton

Docker Deep Dive

Zero to Docker in a single book

February 2018

Now covering all **Docker Certified Associate (DCA)** exam objectives

Nigel Poulton @nigelpoulton

About this edition

This is the 5th edition of the book, published on February 6th, 2018.

I'm calling this the **Docker Certified Associate edition** as the book now **covers all DCA exam objectives!**

Adding this new content has **not** turned this into an exam-cram book! It's is still the easy-to-read real-world-focused book it's always been! It just covers a lot more material!

This edition adds ~200 pages, taking the printed version to just over 400 pages. This feels like the right length for a book like this.

This edition adds the following new chapters:

- Chapter 9: Deploying apps with Docker Compose
- Chapter 11: Docker networking
- Chapter 13: Volumes and persistent data
- Chapter 14: Deploying apps with Docker Stacks
- Chapter 16: Tools for the enterprise
- Chapter 17: Enterprise-grade features
- Appendix A: Securing client and daemon communication
- Appendix B: The DCA exam

(All typos are mine. Or is that typo's)

Education is about inspiring and creating opportunities.

I hope this book, and all my Pluralsight video-training courses, inspire you and create new opportunities!

Huge thanks to my wife and kids for putting up with a geek in the house who thinks he's a bunch of software running inside a container on top of midrange biological hardware. It can't be easy living with me!

Massive thanks to everyone who watches my Pluralsight videos. I love connecting with you and appreciate all the feedback I've had over the years. This helped me decide to write this book! I hope it'll be an amazing tool to help you drive your careers forward.

*If you're planning on taking the DCA exam - **good luck!***

@nigelpoulton

Contents

10: Docker Swarm

Now that we know how to install Docker, pull images, and work with containers, the next thing we need is a way to work with things at scale. That's where Docker Swarm comes into the picture.

At a high level Swarm has two major components:

- A secure cluster
- An orchestration engine

As usual, we'll split this chapter into the usual three parts:

- The TLDR
- The deep dive
- The commands

The examples and outputs we'll use will be from a Linux-based swarm. However, most commands and features work with Docker on Windows.

Docker Swarm - The TLDR

Docker Swarm is two things: an enterprise-grade secure cluster of Docker hosts, and an engine for orchestrating microservices apps.

On the clustering front, it groups one or more Docker nodes and lets you manage them as a cluster. Out-of-the-box you get an encrypted distributed cluster store, encrypted networks, mutual TLS, secure cluster join tokens, and a PKI that makes managing and rotating certificates a breeze! And you can non-disruptively add and remove nodes. It's a beautiful thing!

On the orchestration front, swarm exposes a rich API that allows you to deploy and manage complicated microservices apps with ease. You can define your apps in declarative manifest files, and deploy them with native Docker commands. You can even perform rolling updates, rollbacks, and scaling operations. Again, all with simple commands.

In the past, Docker Swarm was a separate product that you layered on top of the Docker engine. Since Docker 1.12 it's fully integrated into the Docker engine and can be enabled with a single command. As of 2018, it has the ability to deploy and manage native swarm apps as well as Kubernetes apps. Though at the time of writing, support for Kubernetes apps is relatively new.

Docker Swarm - The Deep Dive

We'll split the deep dive part of this chapter as follows:

- Swarm primer
- Build a secure swarm cluster
- Deploy some swarm services
- Troubleshooting

The examples cited will be based on Linux, but they will also work on Windows. Where there are differences we'll be sure to point them out.

Swarm mode primer

On the clustering front, a *swarm* consists of one or more Docker *nodes*. These can be physical servers, VMs, Raspberry Pi's, or cloud instances. The only requirement is that all nodes can communicate over reliable networks.

Nodes are configured as *managers* or *workers*. *Managers* look after the control plane of the cluster, meaning things like the state of the cluster and dispatching tasks to *workers*. *Workers* accept tasks from *managers* and execute them.

The configuration and state of the *swarm* is held in a distributed *etcd* database located on all managers. It's kept in memory and is extremely up-to-date. But the best thing

about it is the fact that it requires zero configuration — it's installed as part of the swarm and just takes care of itself.

Something that's game changing on the clustering front is the approach to security. TLS is so tightly integrated that it's impossible to build a swarm without it. In today's security conscious world, things like this deserve all the props they get! Anyway, *swarm* uses TLS to encrypt communications, authenticate nodes, and authorize roles. Automatic key rotation is also thrown in as the icing on the cake! And it all happens so smoothly that you wouldn't even know it was there!

On the application orchestration front, the atomic unit of scheduling on a swarm is the *service*. This is a new object in the API, introduced along with swarm, and is a higher level construct that wraps some advanced features around containers.

When a container is wrapped in a service we call it a *task* or a *replica*, and the service construct adds things like scaling, rolling updates, and simple rollbacks.

The high-level view is shown in Figure 10.1.

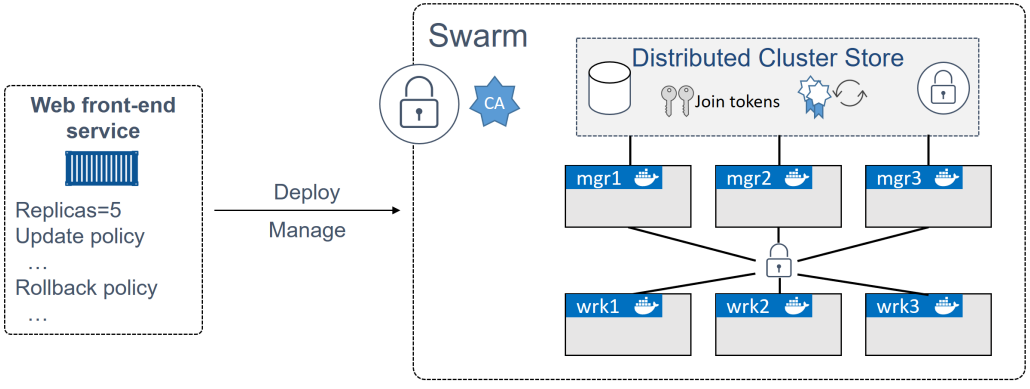

Figure 10.1 High-level swarm

That's enough of a primer. Let's get our hands dirty with some examples.

Build a secure Swarm cluster

In this section we'll build a secure swarm cluster with three *manager nodes* and three *worker nodes*. You can use a different lab with different numbers of *managers* and

workers, and with different names and IPs, but the examples that follow will use the values in Figure 10.2.

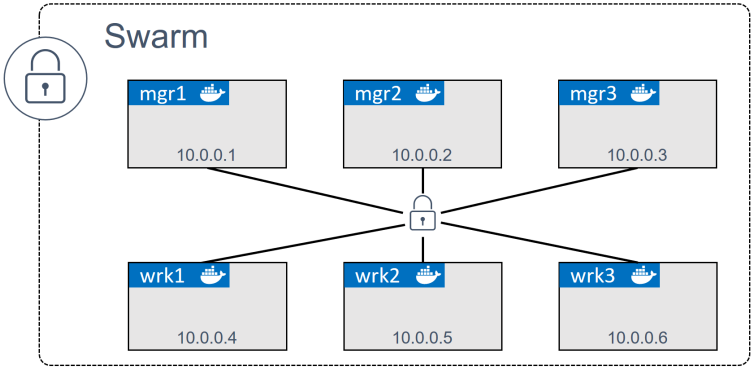

Figure 10.2

Each of the nodes needs Docker installed and needs to be able to communicate with the rest of the swarm. It' also beneficial if name resolution is configured — it makes it easier to identify nodes in command outputs and helps when troubleshooting.

On the networking front, you should have the following ports open on routers and firewalls:

- `2377/tcp`: for secure client-to-swarm communication
- `7946/tcp and 7946/udp`: for control plane gossip
- `4789/udp`: for VXLAN-based overlay networks

Once you've satisfied the pre-requisites, you can go ahead and build a swarm.

The process of building a swarm is sometimes called *initializing a swarm*, and the high-level process is this: Initialize the first manager node > Join additional manager nodes > Join worker nodes > Done.

Initializing a brand new swarm

Docker nodes that are not part of a swarm are said to be in *single-engine mode*. Once they're added to a swarm they're switched into *swarm mode*.

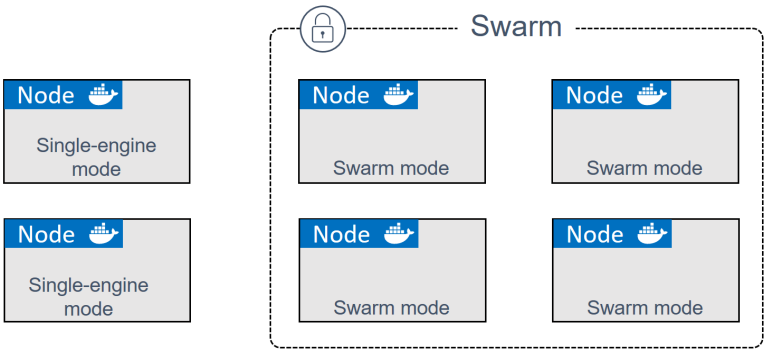

Figure 10.3 Swarm mode vs single-engine mode

Running `docker swarm init` on a Docker host in *single-engine mode* will switch that node into *swarm mode*, create a new *swarm*, and make the node the first *manager* of the swarm.

Additional nodes can then be *joined* as workers and managers. This obviously switches them into *swarm mode* as part of the operation.

The following steps will put **mgr1** into *swarm mode* and initialize a new swarm. It will then join **wrk1**, **wrk2**, and **wrk3** as worker nodes — automatically putting them into *swarm mode*. Finally, it will add **mgr2** and **mgr3** as additional managers and switch them into *swarm mode*. At the end of the procedure all 6 nodes will be in *swarm mode* and operating as part of the same swarm.

This example will use the IP addresses and DNS names of the nodes shown in Figure 10.2. Yours may be different.

1. Log on to **mgr1** and initialize a new swarm (don't forget to use backticks instead of backslashes if you're following along with Windows in a PowerShell terminal).

```
$ docker swarm init \
  --advertise-addr 10.0.0.1:2377 \
  --listen-addr 10.0.0.1:2377

Swarm initialized: current node (d21lyz...c79qzkx) is now a manager.
```

The command can be broken down as follows:

- `docker swarm init` tells Docker to initialize a new swarm and make this node the first manager. It also enables swarm mode on the node.
- `--advertise-addr` is the IP and port that other nodes should use to connect to this manager. It's an optional flag, but it gives you control over which IP gets used on nodes with multiple IPs. It also gives you the chance to specify an IP address that does not exist on the node, such as a load balancer IP.
- `--listen-addr` lets you specify which IP and port you want to listen on for swarm traffic. This will usually match the `--advertise-addr`, but is useful in situations where you want to restrict swarm to a particular IP on a system with multiple IPs. It's also required in situations where the `--advertise-addr` refers to a remote IP address like a load balancer.

I recommend you be specific and always use both flags.

The default port that swarm mode operates on is **2377**. This is customizable, but it's convention to use `2377/tcp` for secured (HTTPS) client-to-swarm connections.

2. List the nodes in the swarm

```
$ docker node ls
ID            HOSTNAME   STATUS  AVAILABILITY  MANAGER STATUS
d21...qzkx *  mgr1       Ready   Active        Leader
```

Notice that **mgr1** is currently the only node in the swarm, and is listed as the *Leader*. We'll come back to this in a second.

3. From **mgr1** run the `docker swarm join-token` command to extract the commands and tokens required to add new workers and managers to the swarm.

```
$ docker swarm join-token worker
To add a manager to this swarm, run the following command:
   docker swarm join \
   --token SWMTKN-1-0uahebax...c87tu8dx2c \
   10.0.0.1:2377

$ docker swarm join-token manager
To add a manager to this swarm, run the following command:
   docker swarm join \
   --token SWMTKN-1-0uahebax...ue4hv6ps3p \
   10.0.0.1:2377
```

Notice that the commands to join a worker and a manager are identical apart from the join tokens (`SWMTKN...`). This means that whether a node joins as a worker or a manager depends entirely on which token you use when joining it. **You should ensure that your join tokens are protected, as they are all that is required to join a node to a swarm!**

4. Log on to **wrk1** and join it to the swarm using the `docker swarm join` command with the worker join token.

```
$ docker swarm join \
    --token SWMTKN-1-0uahebax...c87tu8dx2c \
    10.0.0.1:2377 \
    --advertise-addr 10.0.0.4:2377 \
    --listen-addr 10.0.0.4:2377

This node joined a swarm as a worker.
```

The `--advertise-addr`, and `--listen-addr` flags optional. I've added them as I consider it best practice to be as specific as possible when it comes to network configuration.

5. Repeat the previous step on **wrk2** and **wrk3** so that they join the swarm as workers. Make sure you use **wrk2** and **wrk3's** own IP addresses for the `--advertise-addr` and `--listen-addr` flags.
6. Log on to **mgr2** and join it to the swarm as a manager using the `docker swarm join` command with the token used for joining managers.

```
$ docker swarm join \
    --token SWMTKN-1-0uahebax...ue4hv6ps3p \
    10.0.0.1:2377 \
    --advertise-addr 10.0.0.2:2377 \
    --listen-addr 10.0.0.1:2377

This node joined a swarm as a manager.
```

7. Repeat the previous step on **mgr3**, remembering to use **mgr3's** IP address for the `advertise-addr` and `--listen-addr` flags.
8. List the nodes in the swarm by running `docker node ls` from any of the manager nodes in the swarm.

```
$ docker node ls
ID                HOSTNAME     STATUS  AVAILABILITY  MANAGER STATUS
0g4rl...babl8 *   mgr2         Ready   Active        Reachable
2xlti...l0nyp     mgr3         Ready   Active        Reachable
8yv0b...wmr67     wrk1         Ready   Active
9mzwf...e4m4n     wrk3         Ready   Active
d21ly...9qzkx     mgr1         Ready   Active        Leader
e62gf...l5wt6     wrk2         Ready   Active
```

Congratulations! You've just created a 6-node swarm with 3 managers and 3 workers. As part of the process you put the Docker Engine on each node into *swarm mode*. As a bonus, the *swarm* is automatically secured with TLS.

If you look in the `MANAGER STATUS` column you'll see that the three manager nodes are showing as either "Reachable" or "Leader". We'll learn more about leaders shortly. Nodes with nothing in the `MANAGER STATUS` column are *workers*. Also note the asterisk (*) after the ID on the line showing **mgr2**. This shows us which node we ran the `docker node ls` command from. In this instance the command was issued from **mgr2**.

> **Note**: It's a pain to specify the `--advertise-addr` and `--listen-addr` flags every time you join a node to the swarm. However, it can be a much bigger pain if you get the network configuration of your swarm wrong. Also, manually adding nodes to a swarm is unlikely to be a daily

> task, so I think it's worth the extra up-front effort to use the flags. It's your choice though. In lab environments or nodes with only a single IP you probably don't need to use them.

Now that we have a *swarm* up and running, let's take a look at manager high availability (HA).

Swarm manager high availability (HA)

So far, we've added three manager nodes to a swarm. Why did we add three, and how do they work together? We'll answer all of this, plus more in this section.

Swarm *managers* have native support for high availability (HA). This means one or more can fail, and the survivors will keep the swarm running.

Technically speaking, swarm implements a form of active-passive multi-manager HA. This means that although you might — and should — have multiple *managers*, only one of them is ever considered *active*. We call this active manager the "*leader*", and the leader's the only one that will ever issue live commands against the *swarm*. So it's only ever the leader that changes the config, or issues tasks to workers. If a passive (non-active) manager receives commands for the swarm, it proxies them across to the leader.

This process is shown in Figure 10.4. Step 1 is the command coming in to a *manager* from a remote Docker client. Step 2 is the non-leader manager proxying the command to the leader. Step 3 is the leader executing the command on the swarm.

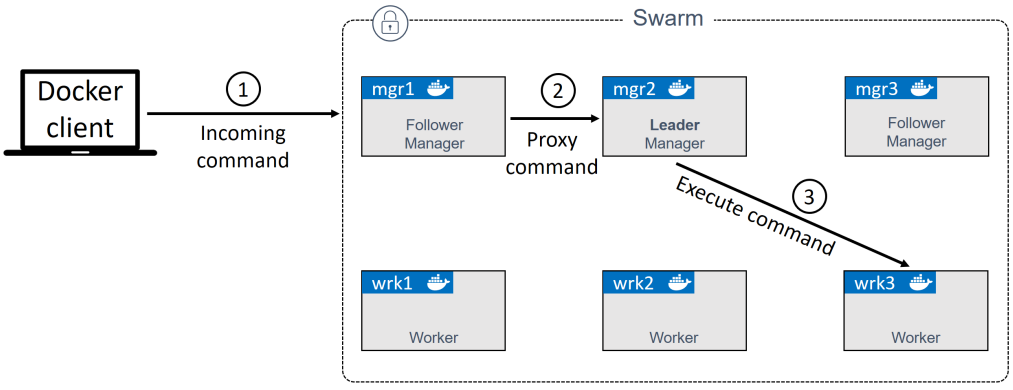

Figure 10.4

If you look closely at Figure 10.4 you'll notice that managers are either *leaders* or *followers*. This is Raft terminology, because swarm uses an implementation of the Raft consensus algorithm[22] to power manager HA. And on the topic of HA, the following two best practices apply:

1. Deploy an odd number of managers.
2. Don't deploy too many managers (3 or 5 is recommended)

Having an odd number of *managers* reduces the chances of split-brain conditions. For example, if you had 4 managers and the network partitioned, you could be left with two managers on each side of the partition. This is known as a split brain — each side knows there used to be 4 but can now only see 2. But crucially, neither side has any way of knowing if the other two are still alive and whether it holds a majority (quorum). The cluster continues to operate during split-brain conditions, but you are no longer able to alter the configuration or add and manage application workloads.

However, if you had 3 or 5 managers and the same network partition occurred, it would be impossible to have the same number of managers on both sides of the partition. This means that one side achieve quorum and cluster management would remain available. The example on the right side of Figure 10.5 shows a partitioned cluster where the left side of the split knows it has a majority of managers.

[22] https://raft.github.io/

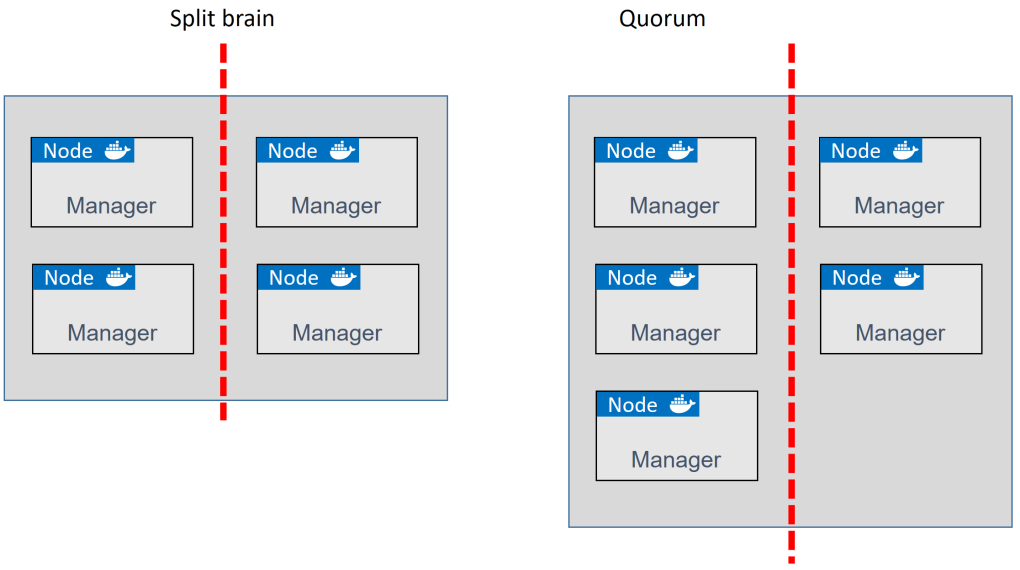

Figure 10.5

As with all consensus algorithms, more participants means more time required to achieve consensus. It's like deciding where to eat — it's always quicker and easier to decide with 3 people than it is with 33! With this in mind, it's a best practice to have either 3 or 5 managers for HA. 7 might work, but it's generally accepted that 3 or 5 is optimal. You definitely don't want more than 7, as the time taken to achieve consensus will be longer.

A final word of caution regarding manager HA. While it's obviously a good practice to spread your managers across availability zones within your network, you need to make sure that the networks connecting them are reliable! Network partitions can be a royal pain in the backside! This means, at the time of writing, the nirvana of hosting your active production applications and infrastructure across multiple cloud providers such as AWS and Azure is a bit of a daydream. Take time to make sure your managers are connected via reliable high-speed networks!

Built-in Swarm security

Swarm clusters have a ton of built-in security that's configured out-of-the-box with sensible defaults — CA settings, join tokens, mutual TLS, encrypted cluster store,

encrypted networks, cryptographic node ID's and more. See **Chapter 15: Security in Docker** for a detailed look at these.

Locking a Swarm

Despite all of this built-in native security, restarting an older manager or restoring an old backup has the potential to compromise the cluster. Old managers re-joining a swarm automatically decrypt and gain access to the Raft log time-series database — this can pose security concerns. Restoring old backups can wipe the current swarm configuration.

To prevent situations like these, Docker allows you to lock a swarm with the Autolock feature. This forces managers that have been restarted to present the cluster unlock key before being permitted back into the cluster.

It's possible to apply a lock directly to a new swarm you are creating by passing the `--autolock` flag to the `docker swarm init` command. However, we've already built a swarm, so we'll lock our existing swarm with the `docker swarm update` command.

Run the following command from a swarm manager.

```
$ docker swarm update --autolock=true
Swarm updated.
To unlock a swarm manager after it restarts, run the
`docker swarm unlock`command and provide the following key:

    SWMKEY-1-5+ICW2kRxPxZrVyBDWzBkzZdSd0Yc7C12o4Uuf9NPU4

Please remember to store this key in a password manager, since without
it you will not be able to restart the manager.
```

Be sure to keep the unlock key in a secure place!

Restart one of your manager nodes to see if it automatically re-joins the cluster. You may need to prepend the command with `sudo`.

```
$ service docker restart
```

Try and list the nodes in the swarm.

```
$ docker node ls
Error response from daemon: Swarm is encrypted and needs to be unlocked

before it can be used.
```

Although the Docker service has restarted on the manager, it has not been allowed to re-join the cluster. You can prove this even further by running the `docker node ls` command on another manager node. The restarted manager will show as `down` and `unreachable`.

Use the `docker swarm unlock` command to unlock the swarm for the restarted manager. You'll need to run this command on the restarted manager, and you'll need to provide the unlock key.

```
$ docker swarm unlock
Please enter unlock key: <enter your key>
```

The node will be allowed to re-join the swarm, and will show as `ready` and `reachable` if you run another `docker node ls`.

Locking your swarm and protecting the unlock key is recommended for production environments.

Now that we've got our *swarm* built, and we understand the concepts of *leaders* and *manager HA*, let's move on to *services*.

Swarm services

Everything we do in this section of the chapter gets improved on by Docker Stacks (Chapter 14). However, it's important that you learn the concepts here so that you're prepared for Chapter 14.

Like we said in the *swarm primer*... *services* are a new construct introduced with Docker 1.12, and they only exist in *swarm mode*.

They let us specify most of the familiar container options, such as *name, port mappings, attaching to networks,* and *images*. But they add things, like letting us declare the *desired state* for an application service, feed that to Docker, and let Docker

take care of deploying it and managing it. For example, assume you've got an app with a web front-end. You have an image for it, and testing has shown that you'll need 5 instances to handle normal daily traffic. You would translate this requirement into a single *service* declaring the image the containers should use, and that the service should always have 5 running replicas.

We'll see some of the other things that can be declared as part of a service in a minute, but before we do that, let's see how to create what we just described.

You create a new service with the `docker service create` command.

> **Note**: The command to create a new service is the same on Windows. However, the image used in this example is a Linux image and will not work on Windows. You can substitute the image for a Windows web server image and the command will work. Remember, if you are typing Windows commands from a PowerShell terminal you will need to use the backtick (‘) to indicate continuation on the next line.

```
$ docker service create --name web-fe \
   -p 8080:8080 \
   --replicas 5 \
   nigelpoulton/pluralsight-docker-ci

z7ovearqmruwk0u2vc5o7ql0p
```

Notice that many of the familiar `docker container run` arguments are the same. In the example, we specified `--name` and `-p` which work the same for standalone containers as well as services.

Let's review the command and output.

We used `docker service create` to tell Docker we are declaring a new service, and we used the `--name` flag to name it **web-fe**. We told Docker to map port 8080 on every node in the swarm to 8080 inside of each service replica. Next, we used the `--replicas` flag to tell Docker that there should always be 5 replicas of this service. Finally, we told Docker which image to use for the replicas — it's important to understand that all service replicas use the same image and config!

After we hit `Return`, the manager acting as leader instantiated 5 replicas across the *swarm* — remember that swarm managers also act as workers. Each worker or manager then pulled the image and started a container from it running on port 8080. The swarm leader also ensured a copy of the service's desired state was stored on the cluster and replicated to every manager in the swarm.

But this isn't the end. All *services* are constantly monitored by the swarm — the swarm runs a background *reconciliation loop* that constantly compares the *actual state* of the service to the *desired state*. If the two states match, the world is a happy place and no further action is needed. If they don't match, swarm takes actions so that they do. Put another way, the swarm is constantly making sure that *actual state* matches *desired state*.

As an example, if a *worker* hosting one of the 5 **web-fe** replicas fails, the *actual state* for the **web-fe** service will drop from 5 replicas to 4. This will no longer match the *desired state* of 5, so Docker will start a new **web-fe** replica to bring *actual state* back in line with *desired state*. This behavior is very powerful and allows the service to self-heal in the event of node failures and the likes.

Viewing and inspecting services

You can use the `docker service ls` command to see a list of all services running on a swarm.

```
$ docker service ls
ID        NAME     MODE        REPLICAS  IMAGE              PORTS
z7o...uw  web-fe   replicated  5/5       nigel...ci:latest  *:8080->8080/tcp
```

The output above shows a single running service as well as some basic information about state. Among other things, we can see the name of the service and that 5 out of the 5 desired replicas are in the running state. If you run this command soon after deploying the service it might not show all tasks/replicas as running. This is often due to the time it takes to pull the image on each node.

You can use the `docker service ps` command to see a list of service replicas and the state of each.

```
$ docker service ps web-fe
ID         NAME      IMAGE             NODE  DESIRED  CURRENT
817...f6z  web-fe.1  nigelpoulton/...  mgr2  Running  Running 2 mins
a1d...mzn  web-fe.2  nigelpoulton/...  wrk1  Running  Running 2 mins
cc0...ar0  web-fe.3  nigelpoulton/...  wrk2  Running  Running 2 mins
6f0...azu  web-fe.4  nigelpoulton/...  mgr3  Running  Running 2 mins
dyl...p3e  web-fe.5  nigelpoulton/...  mgr1  Running  Running 2 mins
```

The format of the command is `docker service ps <service-name or service-id>`. The output displays each replica (container) on its own line, shows which node in the swarm it's executing on, and shows desired state and actual state.

For detailed information about a service, use the `docker service inspect` command.

```
$ docker service inspect --pretty web-fe
ID:             z7ovearqmruwk0u2vc5o7ql0p
Name: Service   web-fe
Mode:           Replicated
 Replicas:      5
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:   nigelpoulton/pluralsight-docker-ci:latest@sha256:7a6b01...d8d3d
Resources: Endpoint
Mode:  vip Ports:
```

```
 PublishedPort = 8080
  Protocol = tcp
  TargetPort = 8080
  PublishMode = ingress
```

The example above uses the `--pretty` flag to limit the output to the most interesting items printed in an easy-to-read format. Leaving off the `--pretty` flag will give a more verbose output. I highly recommend you read through the output of `docker inspect` commands as they're a great source of information and a great way to learn what's going on under the hood.

We'll come back to some of these outputs later.

Replicated vs global services

The default replication mode of a service is `replicated`. This will deploy a desired number of replicas and distribute them as evenly as possible across the cluster.

The other mode is `global`, which runs a single replica on every node in the swarm.

To deploy a *global service* you need to pass the `--mode global` flag to the `docker service create` command.

Scaling a service

Another powerful feature of *services* is the ability to easily scale them up and down.

Let's assume business is booming and we're seeing double the amount of traffic hitting the web front-end. Fortunately, scaling the **web-fe** service is as simple as running the `docker service scale` command.

```
$ docker service scale web-fe=10
web-fe scaled to 10
```

This command will scale the number of service replicas from 5 to 10. In the background it's updating the service's *desired state* from 5 to 10. Run another `docker service ls` command to verify the operation was successful.

```
$ docker service ls
ID        NAME     MODE        REPLICAS  IMAGE              PORTS
z7o...uw  web-fe   replicated  10/10     nigel...ci:latest  *:8080->8080/tcp
```

Running a `docker service ps` command will show that the service replicas are balanced across all nodes in the swarm evenly.

```
$ docker service ps web-fe
ID         NAME       IMAGE             NODE  DESIRED  CURRENT
nwf...tpn  web-fe.1   nigelpoulton/...  mgr1  Running  Running 7 mins
yb0...e3e  web-fe.2   nigelpoulton/...  wrk3  Running  Running 7 mins
mos...gf6  web-fe.3   nigelpoulton/...  wrk2  Running  Running 7 mins
utn...6ak  web-fe.4   nigelpoulton/...  wrk3  Running  Running 7 mins
2ge...fyy  web-fe.5   nigelpoulton/...  mgr3  Running  Running 7 mins
64y...m49  web-fe.6   igelpoulton/...   wrk3  Running  Running about a min
ild...51s  web-fe.7   nigelpoulton/...  mgr1  Running  Running about a min
vah...rjf  web-fe.8   nigelpoulton/...  wrk2  Running  Running about a mins
xe7...fvu  web-fe.9   nigelpoulton/...  mgr2  Running  Running 45 seconds ago
l7k...jkv  web-fe.10  nigelpoulton/...  mgr2  Running  Running 46 seconds ago
```

Behind the scenes, swarm runs a scheduling algorithm that defaults to balancing replicas as evenly as possible across the nodes in the swarm. At the time of writing, this amounts to running an equal number of replicas on each node without taking into consideration things like CPU load etc.

Run another `docker service scale` command to bring the number back down from 10 to 5.

```
$ docker service scale web-fe=5
web-fe scaled to 5
```

Now that we know how to scale a service, let's see how we remove one.

Removing a service

Removing a service is simple — may be too simple.

The following `docker service rm` command will delete the service deployed earlier.

```
$ docker service rm web-fe
web-fe
```

Confirm it's gone with the `docker service ls` command.

```
$ docker service ls
ID      NAME    MODE    REPLICAS    IMAGE      PORTS
```

Be careful using the `docker service rm` command, as it deletes all service replicas without asking for confirmation.

Now that the service is deleted from the system, let's look at how to push rolling updates to one.

Rolling updates

Pushing updates to deployed applications is a fact of life. And for the longest time it's been really painful. I've lost more than enough weekends to major application updates, and I've no intention of doing it again.

Well… thanks to Docker *services*, pushing updates to well-designed apps just got a lot easier!

To see this, we're going to deploy a new service. But before we do that we're going to create a new overlay network for the service. This isn't necessary, but I want you to see how it is done and how to attach the service to it.

```
$ docker network create -d overlay uber-net
43wfp6pzea470et4d57udn9ws
```

This creates a new overlay network called "uber-net" that we'll be able to leverage with the service we're about to create. An overlay network creates a new layer 2 network that we can place containers on, and all containers on it will be able to communicate. This works even if the Docker hosts the containers are running on are on different underlying networks. Basically, the overlay network creates a new layer 2 container network on top of potentially multiple different underlying networks.

Figure 10.6 shows two underlay networks connected by a layer 3 router. There is then a single overlay network across both. Docker hosts are connected to the two underlay networks and containers are connected to the overlay. All containers on the overlay can communicate even if they are on Docker hosts plumbed into different underlay networks.

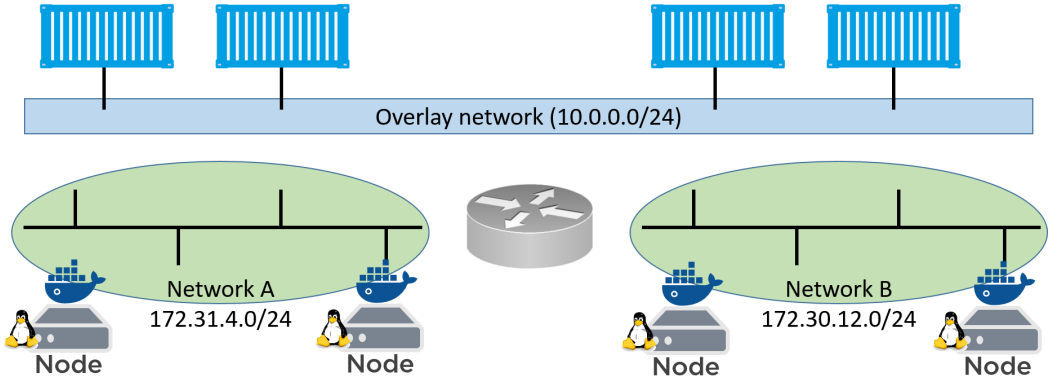

Figure 10.6

Run a `docker network ls` to verify that the network created properly and is visible on the Docker host.

```
$ docker network ls
NETWORK ID          NAME                DRIVER      SCOPE
<Snip>
43wfp6pzea47        uber-net            overlay     swarm
```

The `uber-net` network was successfully created with the `swarm` scope and is *currently* only visible on manager nodes in the swarm.

Let's create a new service and attach it to the network.

```
$ docker service create --name uber-svc \
   --network uber-net \
   -p 80:80 --replicas 12 \
   nigelpoulton/tu-demo:v1

dhbtgvqrg2q4sg07ttfuhg8nz
```

Let's see what we just declared with that `docker service create` command.

The first thing we did was name the service and then use the `--network` flag to tell it to place all replicas on the new `uber-net` network. We then exposed port 80 across the entire swarm and mapped it to port 80 inside of each of the 12 replicas we asked it to run. Finally, we told it to base all replicas on the nigelpoulton/tu-demo:v1 image.

Run a `docker service ls` and a `docker service ps` command to verify the state of the new service.

```
$ docker service ls
ID            NAME      REPLICAS  IMAGE
dhbtgvqrg2q4  uber-svc  12/12     nigelpoulton/tu-demo:v1

$ docker service ps uber-svc
ID        NAME          IMAGE                NODE  DESIRED   CURRENT STATE
0v...7e5  uber-svc.1    nigelpoulton/...:v1  wrk3  Running   Running 1 min
bh...wa0  uber-svc.2    nigelpoulton/...:v1  wrk2  Running   Running 1 min
23...u97  uber-svc.3    nigelpoulton/...:v1  wrk2  Running   Running 1 min
82...5y1  uber-svc.4    nigelpoulton/...:v1  mgr2  Running   Running 1 min
c3...gny  uber-svc.5    nigelpoulton/...:v1  wrk3  Running   Running 1 min
e6...3u0  uber-svc.6    nigelpoulton/...:v1  wrk1  Running   Running 1 min
78...r7z  uber-svc.7    nigelpoulton/...:v1  wrk1  Running   Running 1 min
2m...kdz  uber-svc.8    nigelpoulton/...:v1  mgr3  Running   Running 1 min
b9...k7w  uber-svc.9    nigelpoulton/...:v1  mgr3  Running   Running 1 min
ag...v16  uber-svc.10   nigelpoulton/...:v1  mgr2  Running   Running 1 min
e6...dfk  uber-svc.11   nigelpoulton/...:v1  mgr1  Running   Running 1 min
e2...k1j  uber-svc.12   nigelpoulton/...:v1  mgr1  Running   Running 1 min
```

Passing the service the `-p 80:80` flag will ensure that a **swarm-wide** mapping is created that maps all traffic, coming in to any node in the swarm on port 80, through to port 80 inside of any service replica.

This mode of publishing a port on every node in the swarm — even nodes not running service replicas — is called *ingress mode* and is the default. The alternative mode is *host mode* which only publishes the service on swarm nodes running replicas. Publishing a service in *host mode* requires the long-form syntax and looks like the following:

```
docker service create --name uber-svc \
   --network uber-net \
   --publish published=80,target=80,mode=host \
   --replicas 12 \
   nigelpoulton/tu-demo:v1
```

Open a web browser and point it to the IP address of any of the nodes in the swarm on port 80 to see the service running.

Figure 10.7

As you can see, it's a simple voting application that will register votes for either "football" or "soccer". Feel free to point your web browser to other nodes in the swarm. You'll be able to reach the web service from any node because the `-p 80:80` flag creates an *ingress mode* mapping on every swarm node. This is true even on

nodes that are not running a replica for the service — **every node gets a mapping and can therefore redirect your request to a node that runs the service**.

Now let's assume that this particular vote has come to an end and your company is wants to run a new poll. A new image has been created for the new poll and has been added to the same Docker Hub repository, but this one is tagged as `v2` instead of `v1`.

Let's also assume that you've been tasked with pushing the updated image to the swarm in a staged manner — 2 replicas at a time with a 20 second delay between each. We can use the following `docker service update` command to accomplish this.

```
$ docker service update \
   --image nigelpoulton/tu-demo:v2 \
   --update-parallelism 2 \
   --update-delay 20s uber-svc
```

Let's review the command. `docker service update` lets us make updates to running services by updating the service's desired state. This time we gave it a new image tag `v2` instead of `v1`. And we used the `--update-parallelism` and the `--update-delay` flags to make sure that the new image was pushed to 2 replicas at a time with a 20 second cool-off period in between each. Finally, we told Docker to make these changes to the `uber-svc` service.

If we run a `docker service ps` against the service we'll see that some of the replicas are at `v2` while some are still at `v1`. If we give the operation enough time to complete (4 minutes) all replicas will eventually reach the new desired state of using the `v2` image.

```
$ docker service ps uber-svc
ID        NAME              IMAGE        NODE   DESIRED   CURRENT STATE
7z...nys  uber-svc.1        nigel...v2   mgr2   Running   Running 13 secs
0v...7e5  \_uber-svc.1      nigel...v1   wrk3   Shutdown  Shutdown 13 secs
bh...wa0  uber-svc.2        nigel...v1   wrk2   Running   Running 1 min
e3...gr2  uber-svc.3        nigel...v2   wrk2   Running   Running 13 secs
23...u97  \_uber-svc.3      nigel...v1   wrk2   Shutdown  Shutdown 13 secs
82...5y1  uber-svc.4        nigel...v1   mgr2   Running   Running 1 min
c3...gny  uber-svc.5        nigel...v1   wrk3   Running   Running 1 min
e6...3u0  uber-svc.6        nigel...v1   wrk1   Running   Running 1 min
78...r7z  uber-svc.7        nigel...v1   wrk1   Running   Running 1 min
2m...kdz  uber-svc.8        nigel...v1   mgr3   Running   Running 1 min
b9...k7w  uber-svc.9        nigel...v1   mgr3   Running   Running 1 min
ag...v16  uber-svc.10       nigel...v1   mgr2   Running   Running 1 min
e6...dfk  uber-svc.11       nigel...v1   mgr1   Running   Running 1 min
e2...k1j  uber-svc.12       nigel...v1   mgr1   Running   Running 1 min
```

You can witness the update happening in real-time by opening a web browser to any node in the swarm and hitting refresh several times. Some of the requests will be serviced by replicas running the old version and some will be serviced by replicas running the new version. After enough time, all requests will be serviced by replicas running the updated version of the service.

Congratulations. You've just pushed a rolling update to a live containerized application. Remember, Docker Stacks take all of this to the next level in Chapter 14.

If you run a `docker inspect --pretty` command against the service, you'll see the update parallelism and update delay settings are now part of the service definition. This means future updates will automatically use these settings unless you override them as part of the `docker service update` command.

```
$ docker service inspect --pretty uber-svc
ID:             mub0dgtc8szm80ez5bs8wlt19
Name: Service   uber-svc
Mode:           Replicated
 Replicas:      12
UpdateStatus:
 State:         updating
 Started:       About a minute
 Message:       update in progress
Placement:
UpdateConfig:
 Parallelism:   2
 Delay:         20s
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:    nigelpoulton/tu-demo:v2@sha256:d3c0d8c9...cf0ef2ba5eb74c
Resources: Networks:
uber-net Endpoint
Mode:  vip Ports:

 PublishedPort = 80
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress
```

You should also note a couple of things about the service's network config. All nodes in the swarm that are running a replica for the service will have the `uber-net` overlay network that we created earlier. We can verify this by running `docker network ls` on any node running a replica.

You should also note the `Networks` portion of the `docker inspect` output. This shows the `uber-net` network as well as the swarm-wide `80:80` port mapping.

Troubleshooting

Swarm Service logs can be viewed with the `docker service logs` command. However, not all logging drivers support the command.

By default, Docker nodes configure services to use the `json-file` log driver, but other drivers exist, including:

- `journald` (only works on Linux hosts running `systemd`)
- `syslog`
- `splunk`
- `gelf`

`json-file` and `journald` are the easiest to configure, and both work with the `docker service logs` command. The format of the command is `docker service logs <service-name>`.

If you're using 3rd-party logging drivers you should view those logs using the logging platform's native tools.

The following snippet from a `daemon.json` configuration file shows a Docker host configured to use `syslog`.

```
{
  "log-driver": "syslog"
}
```

You can force individual services to use a different driver by passing the `--log-driver` and `--log-opts` flags to the `docker service create` command. These will override anything set in `daemon.json`.

Service logs work on the premise that your application is running as PID 1 in its container and sending logs to `STDOUT`, and errors to `STDERR`. The logging driver forwards these "logs" to the locations configured via the logging driver.

The following `docker service logs` command shows the logs for all replicas in the `svc1` service that experienced a couple of failures starting a replica.

```
$ docker service logs seastack_reverse_proxy
svc1.1.zhc3cjeti9d4@wrk-2 | [emerg] 1#1: host not found...
svc1.1.6m1nmbzmwh2d@wrk-2 | [emerg] 1#1: host not found...
svc1.1.6m1nmbzmwh2d@wrk-2 | nginx: [emerg] host not found..
svc1.1.zhc3cjeti9d4@wrk-2 | nginx: [emerg] host not found..
svc1.1.1tmya243m5um@mgr-1 | 10.255.0.2 "GET / HTTP/1.1" 302
```

The output is trimmed to fit the page, but you can see that logs from all three service replicas are shown (the two that failed and the one that's running). Each line starts with the name of the replica, which includes the service name, replica number, replica ID, and name of host that it's scheduled on. Following that is the log output.

It's hard to tell because it's trimmed to fit the book, but it looks like the first two replicas failed because they were trying to connect to another service that was still starting (a sort of race condition when dependent services are starting).

You can follow the logs (`--follow`), tail them (`--tail`), and get extra details (`--details`).

Docker Swarm - The Commands

- `docker swarm init` is the command to create a new swarm. The node that you run the command on becomes the first manager and is switched to run in *swarm mode*.
- `docker swarm join-token` reveals the commands and tokens needed to join workers and managers to existing swarms. To expose the command to join a new manager, use the `docker swarm join-token manager` command. To get the command to join a worker, use the `docker swarm join-token worker` command.
- `docker node ls` lists all nodes in the swarm including which are managers and which is the leader.
- `docker service create` is the command to create a new service.
- `docker service ls` lists running services in the swarm and gives basic info on the state of the service and any replicas it's running.
- `docker service ps <service>` gives more detailed information about individual service replicas.

- `docker service inspect` gives very detailed information on a service. It accepts the `--pretty` flag to limit the information returned to the most important information.
- `docker service scale` lets you scale the number of replicas in a service up and down.
- `docker service update` lets you update many of the properties of a running service.
- `docker service logs` lets you view the logs of a service.
- `docker service rm` is the command to delete a service from the swarm. Use it with caution as it deletes all service replicas without asking for confirmation.

Chapter summary

Docker swarm is key to the operation of Docker at scale.

At its core, swarm has a secure clustering component, and an orchestration component.

The secure clustering component is enterprise-grade and offers a wealth of security and HA features that are automatically configured and extremely simple to modify.

The orchestration component allows you to deploy and manage microservices applications in a simple declarative manner. Native Docker Swarm apps are supported, and so are Kubernetes apps.

We'll dig deeper into deploying microservices apps in a declarative manner in Chapter 14.

<Snip>

The following is a snippet from Chapter 15: Security in Docker. It relates to the Docker Swarm sample chapter included in this file.

Security in Swarm Mode

Swarm Mode is the future of Docker. It lets you cluster multiple Docker hosts and deploy your applications in a declarative way. Every Swarm is comprised of *managers* and *workers* that can be Linux or Windows. Managers make up the control plane of the cluster and are responsible for configuring the cluster and dispatching work to it. Workers are the nodes that run your application code as containers.

As expected, Swarm Mode includes many security features that are enabled out-of-the-box with sensible defaults. These include:

- Cryptographic node IDs
- Mutual authentication via TLS
- Secure join tokens
- CA configuration with automatic certificate rotation
- Encrypted cluster store (config DB)
- Encrypted networks

Let's walk through the process of building a secure Swarm and configuring some of the security aspects.

To follow along, you will need at least three Docker hosts running Docker 1.13 or higher. The examples cited use three Docker hosts called "mgr1", "mgr2", and "wrk1". Each one is running Docker 18.01.0-ce on Ubuntu 16.04. There is network connectivity between all three hosts, and all three can ping each other by name. The setup is shown in Figure 15.6.

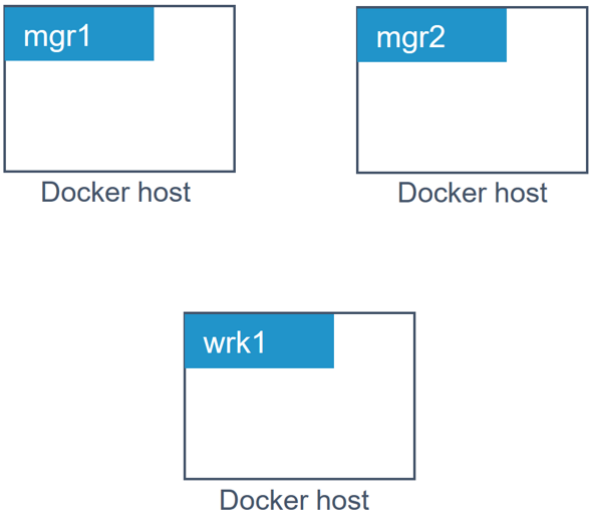

Figure 15.6

Configure a secure Swarm

Run the following command from the node you want to be the first manager in the new Swarm. In the example, we will run it from "mgr1".

```
$ docker swarm init
Swarm initialized: current node (7xam...662z) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token \
     SWMTKN-1-1dmtwu...r17stb-ehp8g...hw738q 172.31.5.251:2377

To add a manager to this swarm, run 'docker swarm join-token manager'
and follow the instructions.
```

That's it! That is literally all you need to do to configure a secure Swarm!

"mgr1" is configured as the first manager of the Swarm and also as the root CA. The Swarm has been given a cryptographic Swarm ID, and "mgr1" has issued itself with

a client certificate that identifies it as a manager in the Swarm. Certificate rotation has been configured with the default value of 90 days, and a cluster config database has been configured and encrypted. A set of secure tokens have also been created so that new managers and new workers can be joined to the Swarm. And all of this with a **single command!**

Figure 15.7 shows how the lab looks now.

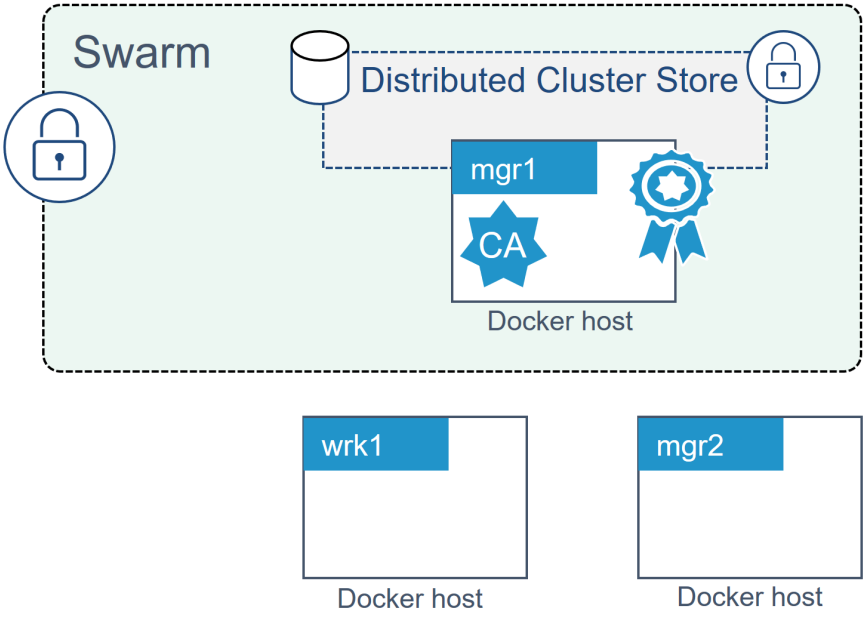

Figure 15.7

Now let's join "mgr2" as an additional manager.

Joining new managers to a Swarm is a two-step process. In the first step, you'll extract the token required to join new managers to the Swarm. In the second, you'll run a `docker swarm join` command on "mgr2". As long as you include the manager join token as part of the `docker swarm join` command, "mgr2" will join the Swarm as a manager.

Run the following command from "mgr1" to extract the manager join token.

```
$ docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join --token \
    SWMTKN-1-1dmtwu...r17stb-2axi5...8p7glz \
    172.31.5.251:2377
```

The output of the command gives you the exact command you need to run on nodes that you want to join the Swarm as managers. The join token and IP address will be different in your lab.

Copy the command and run it on "mgr2":

```
$ docker swarm join --token SWMTKN-1-1dmtwu...r17stb-2axi5...8p7glz \
> 172.31.5.251:2377

This node joined a swarm as a manager.
```

"mgr2" has now joined the Swarm as an additional manager.

> The format of the join command is `docker swarm join --token <manager-join-token> <ip-of-existing-manager>:<swarm-port>`.

You can verify the operation by running a `docker node ls` on either of the two managers.

```
$ docker node ls
ID                HOSTNAME   STATUS    AVAILABILITY    MANAGER STATUS
7xamk...ge662z    mgr1       Ready     Active          Leader
i0ue4...zcjm7f *  mgr2       Ready     Active          Reachable
```

The output above shows that "mgr1" and "mgr2" are both part of the Swarm and are both Swarm managers. The updated configuration is shown in Figure 15.8.

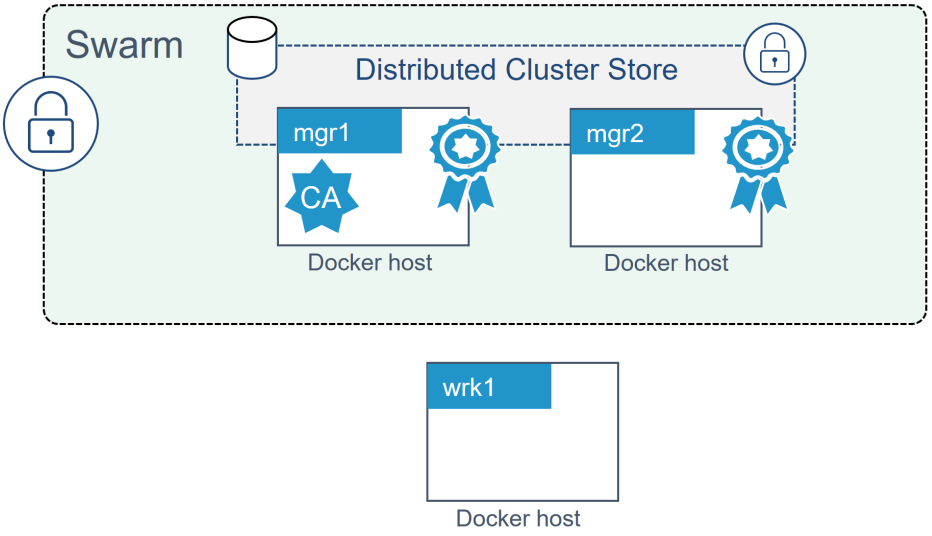

Figure 15.8

Two managers is possibly the worst number you can have. However, we're just messing about in a demo lab, not building a business critical production environment ;-)

Adding a Swarm worker is a similar two-step process. Step 1 is to extract the join token for new workers, and step 2 is to run a `docker swarm join` command on the node you want to join as a worker.

Run the following command on either of the managers to expose the worker join token.

```
$ docker swarm join-token worker

To add a worker to this swarm, run the following command:

    docker swarm join --token \
    SWMTKN-1-1dmtw...17stb-ehp8g...w738q \
    172.31.5.251:2377
```

Again, you get the exact command you need to run on nodes that you want to join as workers. The join token and IP address will be different in your lab.

Copy the command and run it on "wrk1" as shown:

```
$ docker swarm join --token SWMTKN-1-1dmtw...17stb-ehp8g...w738q \
> 172.31.5.251:2377

This node joined a swarm as a worker.
```

Run another `docker node ls` command from either of the Swarm managers.

```
$ docker node ls
ID                  HOSTNAME      STATUS      AVAILABILITY   MANAGER STATUS
7xamk...ge662z *    mgr1          Ready       Active         Leader
ailrd...ofzv1u      wrk1          Ready       Active
i0ue4...zcjm7f      mgr2          Ready       Active         Reachable
```

You now have a Swarm with two managers and one worker. The managers are configured for high availability (HA) and the cluster store is replicated to them both. This updated configuration is shown in Figure 15.9.

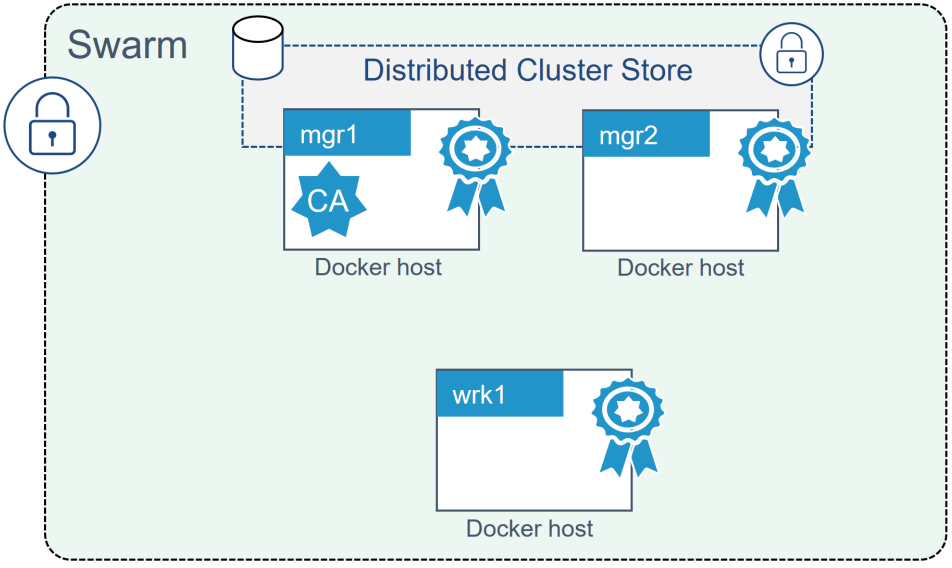

Figure 15.9

Looking behind the scenes at Swarm security

Now that we've built a secure Swarm let's take a minute to look behind the scenes at some of the security technologies involved.

Swarm join tokens

The only thing that is needed to join managers and workers to an existing Swarm is the relevant join token. For this reason, it is vital that you keep your join-tokens safe! No posting them on public GitHub repos!

Every Swarm maintains two distinct join tokens:

- One for joining new managers
- One for joining new workers

It's worth understanding the format of the Swarm join token. Every join token is comprised of 4 distinct fields separated by dashes (-):

```
PREFIX - VERSION - SWARM ID - TOKEN
```

The prefix is always "SWMTKN", enabling you to pattern match against it and prevent people from accidentally posting it publicly. The version field indicates the version of the Swarm. The Swarm ID field is a hash of the Swarm's certificate. The token portion is the part that determines if it can be used to join nodes as managers or workers.

As the following shows, the manager and worker join tokens for a given Swarm are identical except for the final TOKEN field.

- MANAGER: SWMTKN-1-1dmtwusdc…r17stb-**2axi53zjbs45lqxykaw8p7glz**
- WORKER: SWMTKN-1-1dmtwusdc…r17stb-**ehp8gltji64jbl45zl6hw738q**

If you suspect that either of your join tokens has been compromised you can revoke them and issue new ones with a single command. The following example revokes the existing *manager* join token and issues a new one.

```
$ docker swarm join-token --rotate manager

Successfully rotated manager join token.

To add a manager to this swarm, run the following command:

    docker swarm join --token \
     SWMTKN-1-1dmtwu...r17stb-1i7txlh6k3hb921z3yjtcjrc7 \
     172.31.5.251:2377
```

Notice that the only difference between the old and new join tokens is the last field. The Swarm ID remains the same.

Join tokens are stored in the cluster config database which is encrypted by default.

TLS and mutual authentication

Every manager and worker that joins a Swarm is issued a client certificate. This certificate is used for mutual authentication. It identifies the node, which Swarm the node is a member of, and role the node performs in the Swarm (manager or worker).

On a Linux host, you can inspect a node's client certificate with the following command.

```
$ sudo openssl x509 \
  -in /var/lib/docker/swarm/certificates/swarm-node.crt \
  -text

  Certificate:
      Data:
          Version: 3 (0x2)
          Serial Number:
              80:2c:a7:b1:28...a8:af:89:a1:2a:51:89
      Signature Algorithm: ecdsa-with-SHA256
          Issuer: CN=swarm-ca
          Validity
              Not Before: Jul 19 07:56:00 2017 GMT
              Not After : Oct 17 08:56:00 2017 GMT
```

```
        Subject: O=mfbkgjm2tlametbnfqt2zid8x, OU=swarm-manager,
        CN=7xamk8w3hz9q5kgr7xyge662z
        Subject Public Key Info:
<SNIP>
```

The `Subject` data in the output above uses the standard `O`, `OU`, and `CN` fields to specify the Swarm ID, the node's role, and the node ID.

- The Organization `O` field stores the Swarm ID
- The Organizational Unit `OU` field stores the node's role in the Swarm
- The Canonical Name `CN` field stores the node's crypto ID.

This is shown in Figure 15.10.

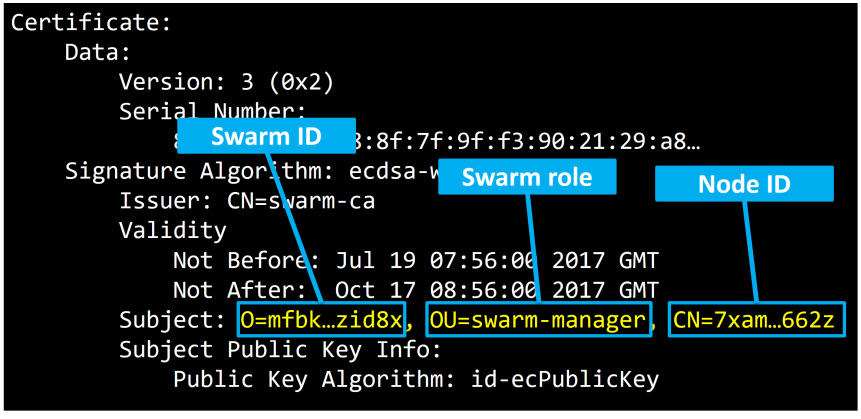

Figure 15.10

We can also see the certificate rotation period in the `Validity` section directly above.

We can match these values to the corresponding values shown in the output of a `docker system info` command.

```
$ docker system info
<SNIP>
Swarm: active
 NodeID: 7xamk8w3hz9q5kgr7xyge662z    << Relates to the CN field
 Is Manager: true                     << Relates to the OU field
 ClusterID: mfbkgjm2tlametbnfqt2zid8x << Relates to the O field ...
 <SNIP>
 ...
 CA Configuration:

  Expiry Duration: 3 months           << Relates to Validity field
 Force Rotate: 0
 Root Rotation In Progress: false
 <SNIP>
```

Configuring some CA settings

You can configure the certificate rotation period for the Swarm with the `docker swarm update` command. The following example changes the certificate rotation period to 30 days.

```
$ docker swarm update --cert-expiry 720h
```

Swarm allows nodes to renew certificates early (slightly before they expire) so that not all nodes in the Swarm try and update their certificates at the same time.

You can configure an external CA when creating a Swarm by passing the `--external-ca` flag to the `docker swarm init` command.

The new `docker swarm ca` sub-command can be used to manage CA related configuration. Run the command with the `--help` flag to see a list of things it can do.

<Snip>

Appendix B: The DCA Exam

This appendix will be updated over time with tips and advice for taking the DCA exam.

I'm also starting a new website and LinkedIn group for you to share your exam experiences and tips.

- The website is www.dockercerts.com and is currently under development
- The LinkedIn group[28] is called **Docker Certified Associate (DCA)

Other resources to help with the exam

At the time of writing, **this is the only resource available that covers all DCA exam objectives**.

I also have an excellent video training course[29] that covers most of the exam objectives and is a great way to help you remember what you've learned in this book.

The video course is fast-paced, fun, and has excellent reviews!

Jose Gomez @pipoe2h · Jan 13

@Docker Deep Dive by @nigelpoulton in @pluralsight is a master piece. Great job, IMHO your best course at the moment. #Docker #Containers #DevOps

[28] https://www.linkedin.com/groups/13578221

[29] https://app.pluralsight.com/library/courses/docker-deep-dive-update/table-of-contents

Deepak koshal @Deepakkoshal · Jan 16

Replying to @nigelpoulton @Docker

I feel so lucky to have you as my trainer. You got me from zero to Docker in true sense and now DCA. In-between is it normal to see flying whales 🐳 in the dream?

Rubén Campos @rucamzu · Jan 18

I completed the first three @pluralsight courses in the @Docker path by @nigelpoulton, including the Deep Dive!! Tons of good stuff in there, and extra kudos to Nigel for making the courses so much the more enjoyable. Thanks!!

Elias Valdez @sailevc · Jan 28

Just finished your Docker Deep Dive Architecture & Theory module recap. Answering to your "Is this good?" question: yeah, it's awesome. It's always good to know the how and why and you made it not boring at all. It was quite the contrary, actually. @nigelpoulton

Emmanuel Ballerini @emballerini · Feb 4

Just finished "Docker deep dive" on @pluralsight by @nigelpoulton Great course! Highly recommended to anyone who wants to learn how Docker really works!

Justin Hartman @STLJHartman · Jan 19

From zero to Proficient in a few weeks, thanks @nigelpoulton for creating amazing @pluralsight #docker courses. Keep on creating!! Your training style is excellent and enthusiasm is addictive. I'm all in and looking forward to getting an expert score in the near future.

Let me say two things if you're unsure about spending money on a video course:

1. It's worth it if it helps you pass the DCA exam!
2. Pluralsight always has a free trial. Sign up for the trial and see if you like it — I think you'll love it!

Mapping exam objectives to chapters

Here is a list of the exam objectives and which chapters they are covered in. Almost all objectives will be covered in more chapters than shown here, but these are the main chapters where they are covered in the most detail.

Domain 1: Orchestration (25% of exam)

- Complete the setup of a swarm mode cluster, with managers and worker nodes: **CHAPTER 10**
- State the differences between running a container vs running a service: **CHAPTERS 10 and 14**
- Demonstrate steps to lock a swarm cluster: **CHAPTER 10**
- Extend the instructions to run individual containers into running services under swarm: **CHAPTERS 10 and 14**
- Interpret the output of "docker inspect" commands: **Several chapters**
- Convert an application deployment into a stack file using a YAML compose file with docker stack deploy: **CHAPTER 14**
- Increase # of replicas: **CHAPTERS 10 and 14**
- Add networks, publish ports: **CHAPTERS 9, 11, 12, and 14**
- Mount volumes: **CHAPTERS 9 and 13**
- Illustrate running a replicated vs global service: **CHAPTER 10**
- Identify the steps needed to troubleshoot a service not deploying: **CHAPTER 14**
- Apply node labels to demonstrate placement of tasks: **CHAPTER 14**
- Sketch how a Dockerized application communicates with legacy systems: **CHAPTER 11**
- Paraphrase the importance of quorum in a swarm cluster: **CHAPTERS 10 and 16**
- Demonstrate the usage of templates with "docker service create": **CHAPTER 10**

Domain 2: Image Creation, Management, and Registry (20% of exam)

- Describe Dockerfile options [add, copy, volumes, expose, entrypoint, etc): **CHAPTERS 8 and 9**
- Show the main parts of a Dockerfile: **CHAPTERS 8 and 9**
- Give examples on how to create an efficient image via a Dockerfile: **CHAPTER 8**
- Use CLI commands such as list, delete, prune, rmi, etc to manage images: **CHAPTER 6**
- Inspect images and report specific attributes using filter and format: **CHAPTER 6**
- Demonstrate tagging an image: **CHAPTERS 6 and 17**
- Utilize a registry to store an image: **CHAPTER 17**
- Display layers of a Docker image: **CHAPTER 6**
- Apply a file to create a Docker image: **CHAPTER 8**
- Modify an image to a single layer: **CHAPTER 8**
- Describe how image layers work: **CHAPTER 8**
- Deploy a registry (not architect): **CHAPTER 16**
- Configure a registry: **CHAPTERS 16 and 17**
- Log into a registry: **CHAPTERS 6 and 17**
- Utilize search in a registry: **CHAPTER 6**
- Tag an image: **CHAPTERS 6 and 17**
- Push an image to a registry: **CHAPTERS 8 and 17**
- Sign an image in a registry: **CHAPTER 17**
- Pull an image from a registry: **CHAPTER 6**
- Describe how image deletion works: **CHAPTERS 6 and 17**
- Delete an image from a registry: **CHAPTER 17**

Domain 3: Installation and Configuration (15% of exam)

- Demonstrate the ability to upgrade the Docker engine: **CHAPTER 3**
- Complete setup of repo, select a storage driver, and complete installation of Docker engine on multiple platforms: **CHAPTER 3**
- Setup swarm, configure managers, add nodes, and setup backup schedule: **CHAPTERS 10 and 16**
- Create and manager user and teams: **CHAPTERS 16 and 17**
- Outline the sizing requirements prior to installation: **CHAPTER 16**
- Understand namespaces, cgroups, and configuration of certificates: **CHAPTERS 5, 15, 16 and Appendix A**
- Use certificate-based client-server authentication to ensure a Docker daemon has the rights to access images on a registry: **CHAPTER 17**
- Consistently repeat steps to deploy Docker engine, UCP, and DTR on AWS and on premises in an HA config: **CHAPTERS 16 and 17**
- Complete configuration of backups for UCP and DTR: **CHAPTER 16**
- Configure the Docker daemon to start on boot: **CHAPTER 3**

Domain 4: Networking (15% of exam)

- Create a Docker bridge network for a developer to use for their containers: **CHAPTER 11**
- Troubleshoot container and engine logs to understand a connectivity issue between containers: **CHAPTER 11**
- Publish a port so that an application is accessible externally: **CHAPTERS 7, 9, 10, 11, 14, and 17**
- Identify which IP and port a container is externally accessible on: **CHAPTERS 7, 9, 11, 17**
- Describe the different types and use cases for the built-in network drivers: **CHAPTER 11**
- Understand the Container Network Model and how it interfaces with the Docker engine and network and IPAM drivers: **CHAPTER 11**

- Configure Docker to use external DNS: **CHAPTER 11**
- Use Docker to load balance HTTP/HTTPs traffic to an application (Configure L7 load balancing with Docker EE): **CHAPTER 17**
- Understand and describe the types of traffic that flow between the Docker engine, registry, and UCP controllers: **CHAPTERS 6, 17, and Appendix A**
- Deploy a service on a Docker overlay network: **CHAPTERS 10, 12, and 14**
- Describe the difference between "host" and "ingress" port publishing mode: **CHAPTERS 11 and 14**

Domain 5: Security (15% of exam)

- Describe the process of signing an image: **CHAPTERS 6, 15, and 17**
- Demonstrate that an image passes a security scan: **CHAPTERS 15 and 17**
- Enable Docker Content Trust: **CHAPTERS 15 and 17**
- Configure RBAC in UCP: **CHAPTER 17**
- Integrate UCP with LDAP/AD: **CHAPTER 17**
- Demonstrate creation of UCP client bundles: **CHAPTERS 16 and 17**
- Demonstrate creation of UCP client bundles: **CHAPTER 15**
- Describe swarm default security: **CHAPTERS 10 and 15**
- Describe MTLS: **CHAPTERS 15 and 17**
- Identity roles: **CHAPTER 17**
- Describe the difference between UCP workers and managers: **CHAPTERS 16 and 17**
- Describe process to use external certificates with UCP and DTR: **CHAPTERS 16 and 17**

Domain 6: Storage and Volumes (10% of exam)

- State which graph driver should be used on which OS: **CHAPTERS 3 and 13**
- Demonstrate how to configure devicemapper: **CHAPTER 3**
- Compare object storage to block storage, and explain which one is preferable when available: **CHAPTER 13**

- Summarize how an application is composed of layers and where those layers reside on the filesystem: **CHAPTERS 6 and 13**
- Describe how volumes are used with Docker for persistent storage: **CHAPTERS 13 and 14**
- Identify the steps you would take to clean up unused images on a filesystem, also on DTR: **CHAPTERS 13 and 17**
- Demonstrate how storage can be used across cluster nodes: **CHAPTERS 13 and 17**

… of the most exciting chapter of your career!