SAMPLE CHAPTER

Mesos IN ACTION

Roger Ignazio

FOREWORD BY Florian Leibert

MANNING

Mesos in Action

by Roger Ignazio

Chapter 1

brief contents

PART 1 HELLO, MESOS 1

1 ■ Introducing Mesos 3

2 ■ Managing datacenter resources with Mesos 17

PART 2 CORE MESOS 31

3 ■ Setting up Mesos 33

4 ■ Mesos fundamentals 58

5 ■ Logging and debugging 79

6 ■ Mesos in production 97

PART 3 RUNNING ON MESOS.....117

7 ■ Deploying applications with Marathon 119

8 ■ Managing scheduled tasks with Chronos 147

9 ■ Deploying applications and managing scheduled tasks with Aurora 169

10 ■ Developing a framework 196

1 Introducing Mesos

This chapter covers

- Introducing Mesos
- Comparing Mesos with a traditional datacenter
- Understanding when and why to use Mesos
- Working with Mesos's distributed architecture

Traditionally, physical—and virtual—machines have been the typical units of computing in a datacenter. Machines are provisioned with various configuration management tools to later have applications deployed. These machines are usually organized into clusters providing individual services, and systems administrators oversee their day-to-day operations. Eventually, these clusters reach their maximum capacity, and more machines are brought online to handle the load.

In 2010, a project at the University of California, Berkeley, aimed to solve the scaling problem. The software project, now known as *Apache Mesos*, abstracts CPU, memory, and disk resources in a way that allows datacenters to function as if they were one large machine. Mesos creates a single underlying cluster to provide applications with the resources they need, without the overhead of virtual machines and operating systems. You can see a simplified example of this in figure 1.1.

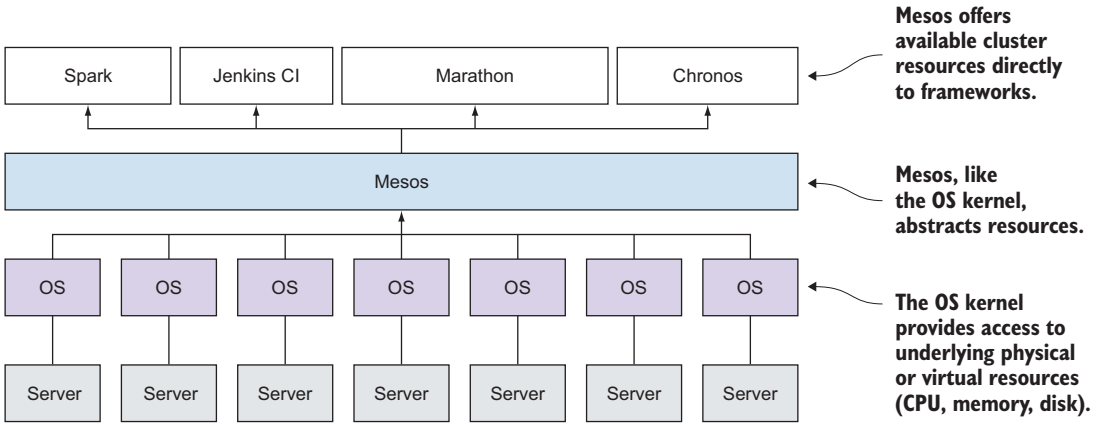

Figure 1.1 Frameworks sharing datacenter resources offered by Mesos

This book introduces Apache Mesos, an open source cluster manager that allows systems administrators and developers to focus less on individual servers and more on the applications that run on them. You'll see how to get up and running with Mesos in your environment, how it shares resources and handles failure, and—perhaps most important—how to use it as a platform to deploy applications.

1.1 Meet Mesos

Mesos works by introducing a layer of abstraction that provides a means to use entire datacenters as if they were a single, large server. Instead of focusing on one application running on a specific server, Mesos's resource isolation allows for multitenancy—the ability to run multiple applications on a single machine—leading to more efficient use of computing resources.

To better understand this concept, you might think of Mesos as being similar to today's virtualization solutions: just as a hypervisor abstracts physical CPU, memory, and storage resources and presents them to virtual machines, Mesos does the same but offers these resources directly to applications. Another way to think about this is in the context of multicore processors: when you launch an application on your laptop, it runs on one or more cores, but in most cases it doesn't particularly matter which one. Mesos applies this same concept to datacenters.

In addition to improving overall resource use, Mesos is distributed, highly available, and fault-tolerant right out of the box. It has built-in support for isolating processes using containers, such as Linux control groups (cgroups) and Docker, allowing multiple applications to run alongside each other on a single machine. Where you once might have set up three clusters—one each to run Memcached, Jenkins CI, and your Ruby on Rails apps—you can instead deploy a single Mesos cluster to run all of these applications.

In the next few sections, you're going to look at how Mesos works to provide all of these features and how it compares to a traditional datacenter.

1.1.1 Understanding how it works

Using a combination of concepts referred to as resource offers, two-tier scheduling, and resource isolation, Mesos provides a means for the cluster to act as a single supercomputer on which to run tasks. Before digging in too deeply here, let's take a look at figure 1.2. This diagram demonstrates the logic Mesos follows when offering resources to running applications. This particular example references the Apache Spark data-processing framework.

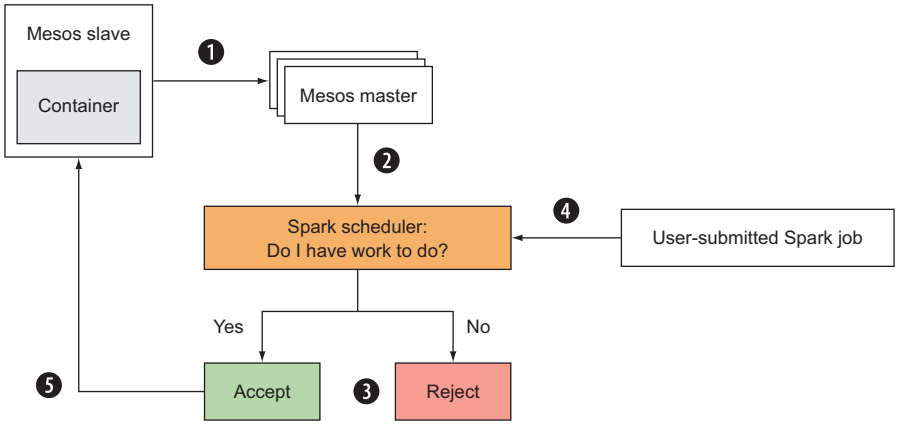

Figure 1.2 Mesos advertises the available CPU, memory, and disk as resource offers to frameworks.

Let's break it down:

1. The Mesos slave offers its available CPU, memory, and disk to the Mesos *master* in the form of a *resource offer*.
2. The Mesos master's *allocation module*—or scheduling algorithm—decides which *frameworks*—or applications—to offer the resources to.
3. In this particular case, the Spark *scheduler* doesn't have any jobs to run on the cluster. It rejects the resource offer, allowing the master to offer the resources to another framework that might have some work to do.
4. Now consider a user submitting a Spark job to be run on the cluster. The scheduler accepts the job and waits for a resource offer that satisfies the workload.
5. The Spark scheduler accepts a resource offer from the Mesos master, and launches one or more *tasks* on an available Mesos *slave*. These tasks are launched

within a container, providing isolation between the various tasks that might be running on a given Mesos slave.

Seems simple, right? Now that you've learned how Mesos uses resource offers to advertise resources to frameworks, and how two-tier scheduling allows frameworks to accept and reject resource offers as needed, let's take a closer look at some of these fundamental concepts.

NOTE An effort is underway to rename the Mesos *slave* role to *agent* for future versions of Mesos. Because this book covers Mesos 0.22.2, it uses the terminology of that specific release, so as to not create any unnecessary confusion. For more information, see https://issues.apache.org/jira/browse/MESOS-1478.

Resource Offers

Like many other cluster managers, Mesos clusters are made up of groups of machines called *masters* and *slaves*. Each Mesos slave in a cluster advertises its available CPU, memory, and storage in the form of resource offers. As you saw in figure 1.2, these resource offers are periodically sent from the slaves to the Mesos masters, processed by a scheduling algorithm, and then offered to a framework's scheduler running on the Mesos cluster.

Two-tier Scheduling

In a Mesos cluster, resource scheduling is the responsibility of the Mesos master's allocation module and the framework's scheduler, a concept known as *two-tier scheduling*. As previously demonstrated, resource offers from Mesos slaves are sent to the master's allocation module, which is then responsible for offering resources to various framework schedulers. The framework schedulers can accept or reject the resources based on their workload.

The allocation module is a pluggable component of the Mesos master that implements an algorithm to determine which offers are sent to which frameworks (and when). The modular nature of this component allows systems engineers to implement their own resource-sharing policies for their organization. By default, Mesos uses an algorithm developed at UC Berkeley known as Dominant Resource Fairness (DRF):

> In a nutshell, DRF seeks to maximize the minimum dominant share across all users. For example, if user A runs CPU-heavy tasks and user B runs memory-heavy tasks, DRF attempts to equalize user A's share of CPUs with user B's share of memory. In the single-resource case, DRF reduces to max-min fairness for that resource.[1]

[1] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types." NSDI, vol. 11, 2011.

Mesos's use of the DRF algorithm by default is fine for most deployments. Chances are you won't need to write your own allocation algorithm, so this book doesn't go into much detail about DRF. If you're interested in learning more about this research, you can find the paper online at www.usenix.org/legacy/events/nsdi11/tech/full_papers/Ghodsi.pdf.

RESOURCE ISOLATION

Using Linux cgroups or Docker containers to isolate processes, Mesos allows for *multi-tenancy*, or for multiple processes to be executed on a single Mesos slave. A framework then executes its tasks within the container, using a Mesos *containerizer*. If you're not familiar with containers, think of them as a lightweight approach to how a hypervisor runs multiple virtual machines on a single physical host, but without the overhead or need to run an entire operating system.

> **NOTE** In addition to Docker and cgroups, Mesos provides another means of isolation for other POSIX-compliant operating systems: `posix/cpu`, `posix/mem`, and `posix/disk`. It's worth noting that these isolation methods don't *isolate* resources, but instead monitor resource use.

Now that you have a clearer understanding of how Mesos works, you can move on to understanding how this technology compares to the traditional datacenter. More specifically, the next section introduces the concept of an application-centric datacenter, where the focus is more on applications than on the servers and operating systems that run them.

1.1.2 Comparing virtual machines and containers

When thinking about applications deployed in a traditional datacenter, virtual machines often come to mind. In recent years, virtualization providers (VMware, OpenStack, Xen, and KVM, to name a few) have become commonplace in many organizations. Similar to how a hypervisor allows a physical host's resources to be abstracted and shared among virtual machines, Mesos provides a layer of abstraction, albeit at a different level. The resources are presented to applications themselves, and in turn consumed by containers.

To illustrate this point, consider figure 1.3, which compares the various layers of infrastructure required to deploy four applications.

VIRTUAL MACHINES

When thinking about traditional virtual machine–based application deployments, consider for a moment the operational overhead of maintaining the operating systems on each of them: installing packages, applying security updates, maintaining user access, identifying and remediating configuration drift; the list goes on. What's the added benefit of running applications atop an entire operating system when you're more concerned with deploying the application itself? Not to mention the overhead of the operating system, which consumes added CPU, memory, and disk. At

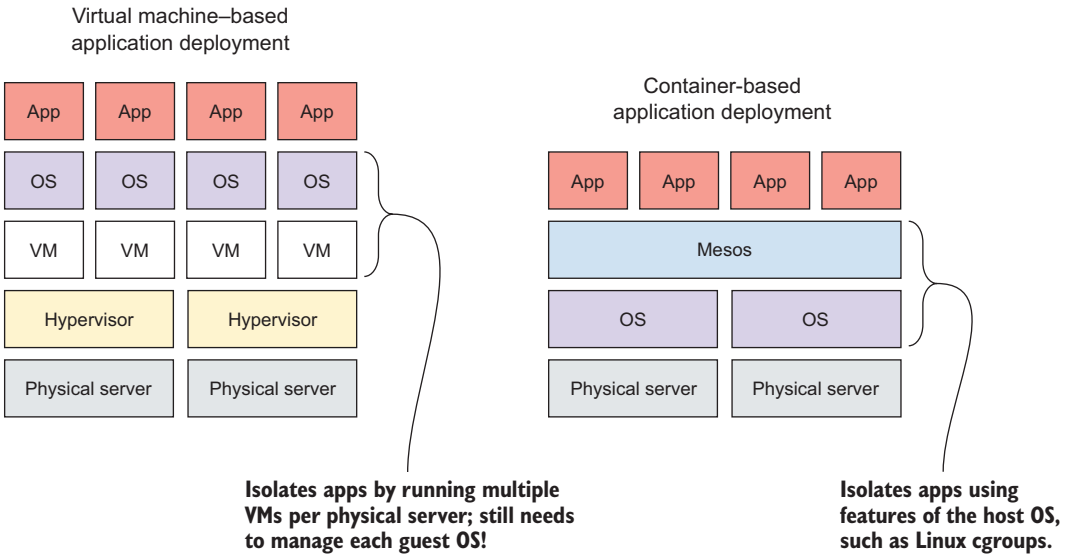

Figure 1.3 Comparing VM-based and container-based application deployments

a large-enough scale, this becomes wasteful. With an application-centric approach to managing datacenters, Mesos allows you to simplify your stack—and your application deployments—using lightweight containers.

CONTAINERS

As you learned previously, Mesos uses containers for resource isolation between processes. In the context of Mesos, the two most important resource-isolation methods to know about are the control groups (cgroups) built into the Linux kernel, and Docker.

Around 2007, support for control groups (referred to as *cgroups* throughout this text) was made available in the Linux kernel, beginning with version 2.6.24. This allows the execution of processes in a way that's *sandboxed* from other processes. In the context of Mesos, cgroups provide resource constraints for running processes, ensuring that they don't interfere with other processes running on the system. When using cgroups, any packages or libraries that the tasks might depend on (a specific version of Python, a working C++ compiler, and so on) must be already present on the host operating system. If your workloads, packages, and required tools and libraries are fairly standardized or don't readily conflict with each other, this might not be a problem. But consider figure 1.4, which demonstrates how using Docker can overcome these sorts of problems and allow you to run applications and workloads in a more isolated manner.

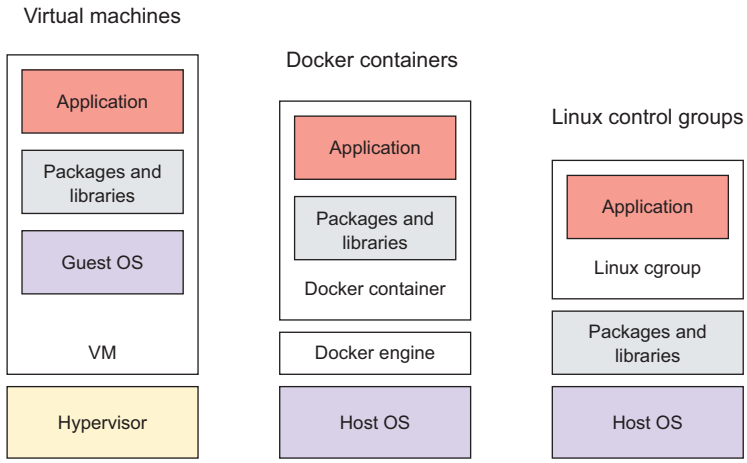

Figure 1.4 Comparing virtual machines, Docker containers, and Linux cgroups

Using low-level primitives in the Linux kernel, including cgroups and namespaces, Docker provides a means to build and deploy containers almost as if they were virtual machines. The application and all of its dependencies are packaged within the container and deployed atop a host operating system. They take a concept from the freight industry—the standardized industrial shipping container—and apply this to application deployment. In recent years, this new unit of software delivery has grown in popularity as it's generally considered to be more lightweight than deploying an entire virtual machine.

You don't need to understand all the implementation details and intricacies of building and deploying containers to use Mesos, though. If you'd like more information, please consult the following online resources:

- Linux control groups: www.kernel.org/doc/documentation/cgroup-v1/cgroups.txt
- Docker: https://docs.docker.com

1.1.3 Knowing when (and why) to use Mesos

Running applications at scale isn't reserved for large enterprises anymore. Startups with only a handful of employees are creating apps that easily attract millions of users. Re-architecting applications and datacenters is a nontrivial task, but certain components that are in a typical stack are already great candidates to run on Mesos. By taking some of these technologies and moving them (and their workloads) to a Mesos cluster, you can scale them more easily and run your datacenter more efficiently.

NOTE This book covers Mesos version 0.22.2, which provides an environment for running stateless and distributed applications. Beginning in version 0.23,

Mesos will begin work to support persistent resources, thus enabling support for stateful frameworks. For more information on this effort, see https://issues.apache.org/jira/browse/MESOS-1554.

For example, consider the stateless, distributed, and stateful technologies in table 1.1.

Table 1.1 Technologies that are—and aren't—good candidates to run on Mesos

Service type	Examples	Should you use Mesos?
Stateless—no need to persist data to disk	Web apps (Ruby on Rails, Play, Django), Memcached, Jenkins CI build slaves	Yes
Distributed out of the box	Cassandra, Elasticsearch, Hadoop Distributed File System (HDFS)	Yes, provided the correct level of redundancy is in place
Stateful—needs to persist data to disk	MySQL, PostgreSQL, Jenkins CI masters	No (version 0.22); potentially (version 0.23+)

The real value of Mesos is realized when running stateless services and applications—applications that will handle incoming loads but that could go offline at any time without negatively impacting the service as a whole, or services that run a job and report the result to another system. As noted previously, examples of some of these applications include Ruby on Rails and Jenkins CI build slaves.

Progress has been made running distributed databases (such as Cassandra and Elasticsearch) and distributed filesystems (such as Hadoop Distributed File System, or HDFS) as Mesos frameworks. But this is feasible only if the correct level of redundancy is in place. Although certain distributed databases and filesystems have data replication and fault tolerance built in, your data might not survive if the entire Mesos cluster fails (because of natural disasters, redundant power/cooling systems failures, or human error). In the real world, you should weigh the risks and benefits of deploying services that persist data on a Mesos cluster.

As I mentioned earlier, Mesos excels at running stateless, distributed services. Stateful applications that need to persist data to disk aren't good candidates for running on Mesos as of this writing. Although possible, it's not yet advisable to run certain databases such as MySQL and PostgreSQL atop a Mesos cluster. When you do need to persist data, it's preferable to do so by deploying a traditional database cluster outside the Mesos cluster.

1.2 Why we need to rethink the datacenter

Deploying applications within a datacenter has traditionally involved one or more physical (or virtual) servers. The introduction and mainstream adoption of virtualization has allowed us to run multiple virtual machines on a single physical server and make better use of physical resources. But running applications this way also means you're usually running a full operating system on each of those virtual machines, which consumes resources and brings along its own maintenance overhead.

This section presents two primary reasons that you should rethink how datacenters are managed: the administrative overhead of statically partitioning resources, and the need to focus more on applications instead of infrastructure.

1.2.1 Partitioning of resources

When you consider the traditional virtual machine–based model of deploying applications and statically partitioning clusters, you quickly find this deployment model inefficient and cumbersome to maintain. By maximizing the use of each server in a datacenter, operations teams maximize their return on investment and can keep the total cost of ownership as reasonable as possible.

In computing circles, teams generally refer to a *cluster* as a group of servers that work together as a single system to provide a service. Traditionally, the deployment of these services has been largely node-centric: you dedicate a certain number of machines to provide a given service. But as the infrastructure footprint expands and service offerings increase, it's difficult to continue statically partitioning these services.

But now consider the demand for these services doubling. To continue scaling, a systems administrator needs to provision new machines and join them to the individual clusters. Perhaps the operations team, anticipating the need for additional capacity, scales each of those clusters to three times its current size. Although you've managed to scale each of those services, you now have machines in your datacenter sitting idle, waiting to be used. As such, if a single machine in any of those clusters fails, it quickly needs to be brought back online for the service to continue operating at full capacity, as shown in figure 1.5.

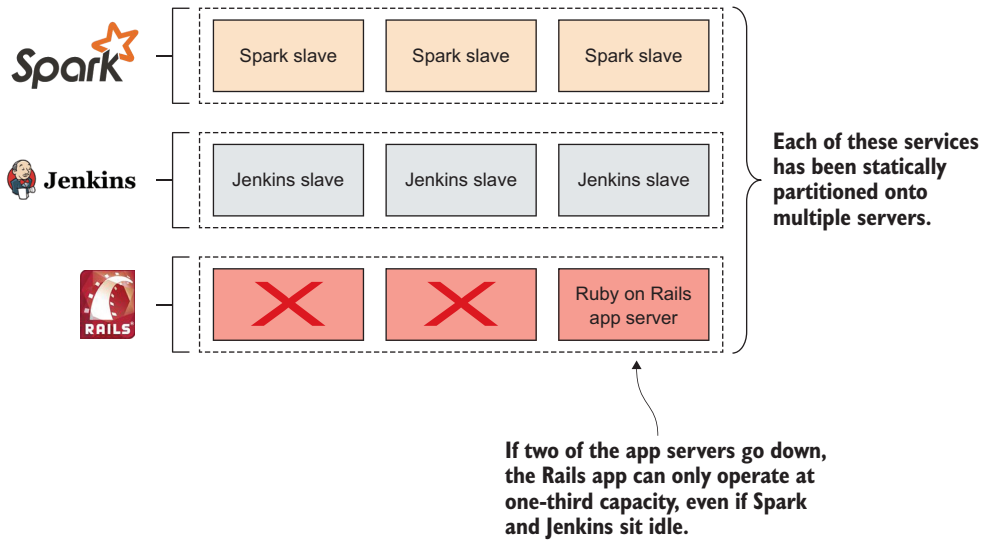

Figure 1.5 Three applications statically partitioned in a datacenter

Now consider solving the aforementioned scaling scenario by using Mesos, as shown in figure 1.6. You can see that you'd use these same machines in the datacenter to focus on running applications instead of virtual machines. The applications could run on any machine with available resources. If you need to scale, you add servers to the *Mesos* cluster, instead of adding machines to multiple clusters. If a single Mesos node goes offline, no particular impact occurs to any one service.

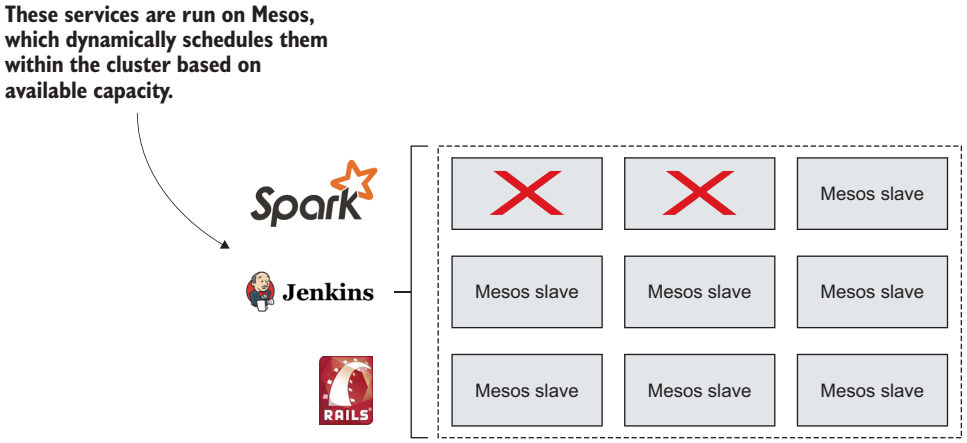

Figure 1.6 Three applications running on a Mesos cluster

Consider these small differences across hundreds or thousands of servers. Instead of trying to guess how many servers you need for each service and provision them into several static clusters, you're able to allow these services to dynamically request the compute, memory, and storage resources they need to run. To continue scaling, you add new machines to your Mesos cluster, and the applications running on the cluster scale to the new infrastructure. Operating a single, large computing cluster in this manner has several advantages:

- You can easily provision additional cluster capacity.
- You can be less concerned about where services are running.
- You can scale from several nodes to thousands.
- The loss of several servers doesn't severely degrade any one service.

1.2.2 Deploying applications

As we discussed previously, one of the major differences—and benefits—of deploying applications on a Mesos cluster is multitenancy. Not unlike a virtualization hypervisor running multiple virtual machines on a physical server, Mesos allows multiple applications to run on a single server in isolated environments, using either Linux *cgroups* or

Docker *containers*. Instead of having multiple environments (one each for development, staging, and production), the entire datacenter becomes a platform on which to deploy applications.

Where Mesos is commonly referred to—and acts as—a distributed *kernel*, other Mesos frameworks help users run long-running and scheduled tasks, similar to the init and Cron systems, respectively. You'll learn more about these frameworks (Marathon, Chronos, and Aurora) and how to deploy applications on them later in this book.

Consider the power of what I've described so far: Mesos provides fault tolerance out of the box. Instead of a systems administrator getting paged when a single server goes offline, the cluster will automatically start the failed job elsewhere. The sysadmin needs to be concerned only if a certain percentage of machines goes offline in the datacenter, as that might signal a larger problem. As such, with the correct placement and redundancy in place, scheduled maintenance can occur at any time.

1.3 The Mesos distributed architecture

To provide services at scale, Mesos provides a distributed, fault-tolerant architecture that enables fine-grained resource scheduling. This architecture comprises three components: *masters*, *slaves*, and the applications (commonly referred to as *frameworks*) that run on them. Mesos relies on Apache ZooKeeper, a distributed database used specifically for coordinating leader election within the cluster, and for leader detection by other Mesos masters, slaves, and frameworks.

In figure 1.7, you can see how each of these architecture components works together to provide a stable platform on which to deploy applications. I'll break it down for you in the sections that follow the diagram.

1.3.1 Masters

One or more Mesos masters are responsible for managing the Mesos slave daemons running on each machine in the cluster. Using ZooKeeper, they coordinate which node will be the *leading master*, and which masters will be on standby, ready to take over if the leading master goes offline.

The leading master is responsible for deciding which resources to offer to a particular framework using a pluggable *allocation module,* or scheduling algorithm, to distribute *resource offers* to the various schedulers. The scheduler can then either accept or reject the offer based on whether it has any work to be performed at that time.

A Mesos cluster requires a minimum of one master, and three or more are recommended for production deployments to ensure that the services are highly available. You can run ZooKeeper on the same machines as the Mesos masters themselves, or use a standalone ZooKeeper cluster. Chapter 3 goes into more detail about the sizing and deploying of Mesos masters.

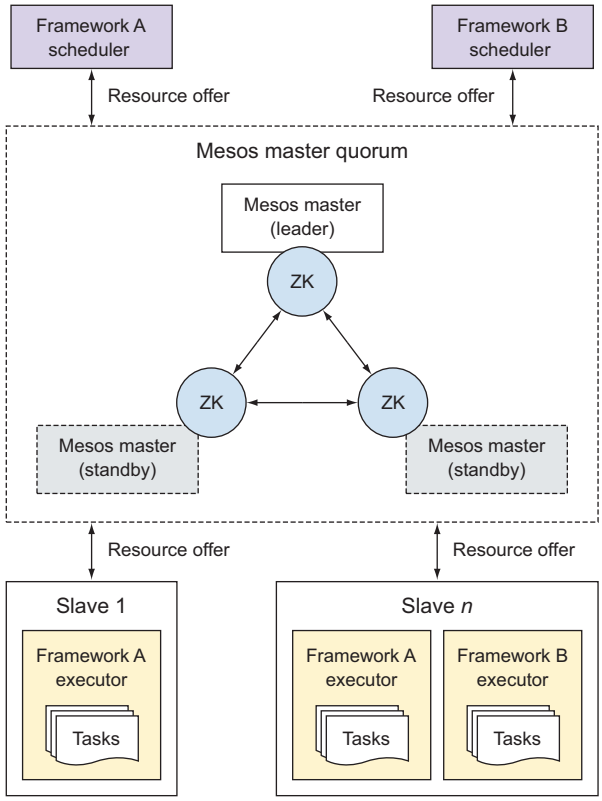

Figure 1.7 The Mesos architecture consists of one or more masters, slaves, and frameworks.

1.3.2 Slaves

The machines in a cluster responsible for executing a framework's tasks are referred to as Mesos *slaves*. They query ZooKeeper to determine the leading Mesos master and advertise their available CPU, memory, and storage resources to the leading master in the form of a resource offer. When a scheduler accepts a resource offer from the Mesos master, it then launches one or more *executors* on the slave, which are responsible for running the framework's tasks.

Mesos slaves can also be configured with certain attributes and resources, which allow them to be customized for a given environment. *Attributes* refer to key/value pairs that might contain information about the node's location in a datacenter, and *resources* allow a particular slave's advertised CPU, memory, and disk to be overridden with user-provided values, instead of Mesos automatically detecting the available resources on the slave. Consider the following example attributes and resources:

```
--attributes='datacenter:pdx1;rack:1-1;os:rhel7'
--resources='cpu:24;mem:24576;disk:409600'
```

I've configured this particular Mesos slave to advertise its datacenter; location within the datacenter; operating system; and user-provided CPU, memory, and disk resources. This information is especially useful when trying to ensure that applications stay online during scheduled maintenance. Using this information, a datacenter operator could take an entire rack (or an entire row!) of machines offline for scheduled maintenance without impacting users. Chapter 4 covers this (and more) in the Mesos slave configuration section.

1.3.3 Frameworks

As you learned earlier, a *framework* is the term given to any Mesos application that's responsible for scheduling and executing tasks on a cluster. A framework is made up of two components: a scheduler and an executor.

> **TIP** A list of frameworks known to exist at the time of writing is included in appendix B.

SCHEDULER

A *scheduler* is typically a long-running service responsible for connecting to a Mesos master and accepting or rejecting resource offers. Mesos delegates the responsibility of scheduling to the framework, instead of attempting to schedule all the work for a cluster itself. The scheduler can then accept or reject a resource offer based on whether it has any tasks to run at the time of the offer. The scheduler detects the leading master by communicating with the ZooKeeper cluster, and then registers itself to that master accordingly.

EXECUTOR

An *executor* is a process launched on a Mesos slave that runs a framework's tasks on a slave. As of this writing, the built-in Mesos executors allow frameworks to execute shell scripts or run Docker containers. New executors can be written using Mesos's various language bindings and bundled with the framework, to be fetched by the Mesos slave when a task requires it.

As you've learned, Mesos provides a distributed, highly available architecture. Masters schedule work to be performed on the cluster, and slaves advertise available resources to the schedulers, which in turn execute tasks on the cluster.

1.4 Summary

In this chapter, you've been introduced to the Apache Mesos project, its architecture, and how it attempts to solve scaling problems and make clustering simple. You've also learned how Mesos deployments compare and contrast with the traditional datacenter, and how an application-centric approach can lead to using resources more efficiently. We've discussed when (and when not) to use Mesos for a given workload, and

where you can get help and find more information, should you need it. Here are a few things to remember:

- Mesos abstracts CPU, memory, and disk resources away from underlying systems and presents multiple machines as a single entity.
- Mesos slaves advertise their available CPUs, memory, and disk in the form of resource offers.
- A Mesos framework comprises two primary components: a scheduler and an executor.
- Containers are a lightweight method to provide resource isolation to individual processes.

In the next chapter, I'll walk you through a real-world example of how Mesos allows for more efficient resource use, and how you might run applications in your own datacenter by building on projects in the Mesos ecosystem.

PROGRAMMING

Mesos IN ACTION

Roger Ignazio

Modern datacenters are complex environments, and when you throw Docker and other container-based systems into the mix, there's a great need to simplify. Mesos is an open source cluster management platform that transforms the whole datacenter into a single pool of compute, memory, and storage resources that you can allocate, automate, and scale as if you're working with a single supercomputer.

Mesos in Action introduces readers to the Apache Mesos cluster manager and the concept of application-centric infrastructure. Filled with helpful figures and hands-on instructions, this book guides you from your first steps creating a highly-available Mesos cluster through deploying applications in production and writing native Mesos frameworks. You'll learn how to scale to thousands of nodes, while providing resource isolation between processes using Linux and Docker containers. You'll also learn practical techniques for deploying applications using popular key frameworks.

What's Inside

- Spinning up your first Mesos cluster
- Scheduling, resource administration, and logging
- Deploying containerized applications with Marathon, Chronos, and Aurora
- Writing Mesos frameworks using Python

Readers need to be familiar with the core ideas of datacenter administration and need a basic knowledge of Python or a similar programming language.

Roger Ignazio is an experienced systems engineer with a focus on distributed, fault-tolerant, and scalable infrastructure. He is currently a technical lead at Mesosphere.

"You would be hard-pressed to find a better guide than Roger Ignazio and a better book than *Mesos in Action*."
—From the Foreword by Florian Leibert, Mesosphere

"Helps to illuminate best practices and avoid hidden pitfalls when deploying Apache Mesos."
—Marco Massenzio, Apple

"Ignazio knows his stuff, but more importantly he knows how to explain it."
—Morgan Nelson, getaroom.com

"You will not only learn Mesos, you will learn a whole ecosystem."
—Thomas Peklak, Emakina CEE

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

ISBN-13: 978-1-61729-292-7
ISBN-10: 1-61729-292-3
54499
9 781617 292927

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]