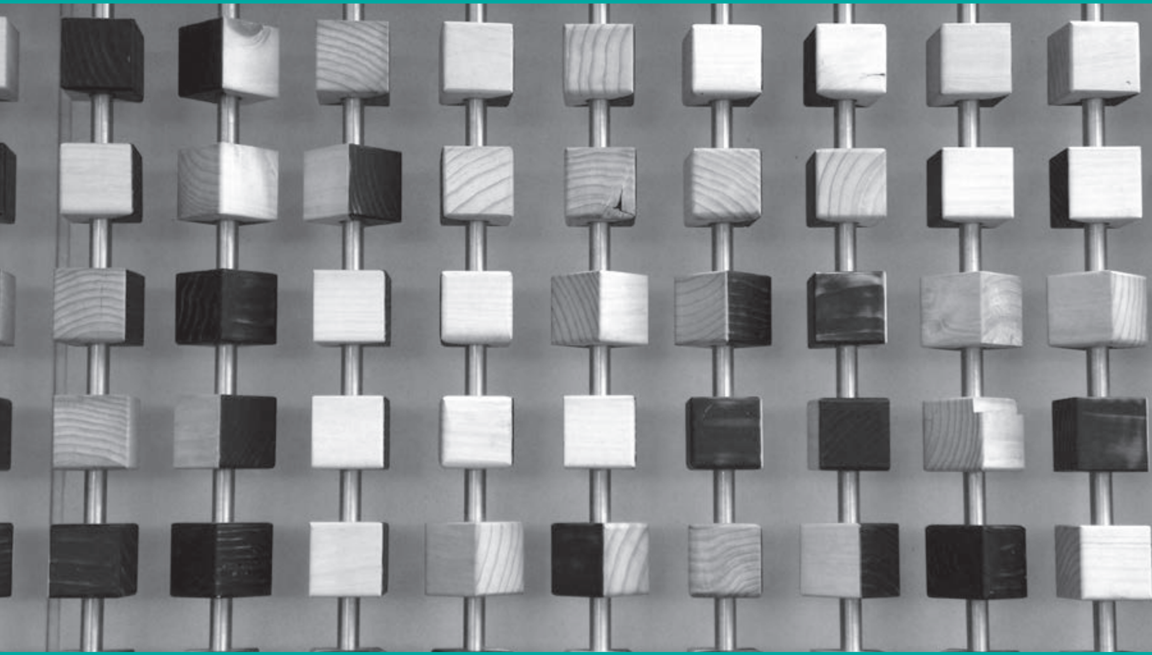
O'REILLY®

Compliments of MESOSPHERE

# Application Delivery with DC/OS

## Building and Running Modern Data-Driven Apps



Andrew Jefferson

# Application Delivery with DC/OS

*Building and Running Modern Data-Driven Apps*

*Andrew Jefferson*

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

**Application Delivery with DC/OS**
by Andrew Jefferson

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

**Editors:** Brian Anderson and Virginia Wilson
**Production Editor:** Nicholas Adams
**Copyeditor:** Octal Publishing, Inc.
**Interior Designer:** David Futato
**Cover Designer:** Randy Comer
**Illustrator:** Rebecca Demarest

April 2017: First Edition

**Revision History for the First Edition**
2017-03-28: First Release

978-1-491-98342-3

[LSI]

# Table of Contents

# Foreword

In 2009, my UC Berkeley colleagues and I observed that the world of computing was changing from small applications powered by large machines (where VM-partitioning made sense), to larger apps powered by clusters of low-cost machines. The explosion of data and users meant that modern enterprise apps had to become distributed systems, and we needed a way to easily run this new type of application. Later that year we published a research paper titled "The Datacenter Needs an Operating System."

Managing users and data at scale were real-world problems faced by companies like Twitter and AirBnB. VM-centric (or even container-centric) approaches were too low level—what mattered were the services running on top, e.g., Spark and Kafka. Moreover, each of these services re-implemented the same set of functionalities (e.g., failure detection, monitoring). We needed something to enable these services to run on aggregated compute resources, abstracting away the servers underneath, just like we abstract away the resources in our laptops, servers, smartphones, tablets, etc. We needed an operating system for the datacenter.

Replacing the word "computer" with "datacenter" in the Wikipedia definition of an operating system captures this need succinctly: "A collection of software that manages the *datacenter* ~~computer~~ hardware resources and provides common services for *datacenter* ~~computer~~ programs."

DC/OS—our datacenter operating system—began with the Apache Mesos distributed system kernel, which we started at UC Berkeley and then used in production at Twitter and other organizations. In April 2016, Mesosphere open sourced DC/OS. Today, 100+ services are available at the click of a mouse, including data services like Apache Spark, Apache Cassandra, Apache Kafka, and ElasticSearch—and more. Developers can choose the services they want, while operators can pick any infrastructure they'd like to run on.

I hope you enjoy this book.

*— Ben Hindman, Apache Mesos PMC Chair & Mesosphere Cofounder*

CHAPTER 1

# Introduction

In this report, I introduce DC/OS and the Modern Enterprise Architecture proposed by Mesosphere for building and operating software applications and services. I explain in detail how DC/OS works and how to build applications to run on DC/OS. I also explain how the Modern Enterprise Architecture can meet the needs of organizations from startups to large enterprises, and how using it can benefit software development, systems administration, and data strategy. Here are some brief descriptions to help familiarize you with these terms:

*DC/OS*
: This stands for Data Center Operating System, which is a system composed of Linux nodes communicating over a network to provide software-defined services. A DC/OS Cluster provides a software-defined platform on which applications can be deployed and which can scale to thousands of nodes in a datacenter. DC/OS provides an operational approach and integrated set of software tools to run complex multicomponent software systems and manage the operation of those systems.

*Mesosphere*
: Mesosphere is the company that created DC/OS. It sells Mesosphere Enterprise DC/OS (the enterprise version of DC/OS). In the words of Mesosphere CEO and cofounder Florian Leibert:

> Mesosphere is democratizing the modern infrastructure we used at Twitter, AirBnB, and other web-scale companies to quickly deliver data-driven services on any datacenter or cloud.

*Modern Enterprise Architecture*
: This is a system proposed by Mesosphere for building services using DC/OS to run multiple software applications powered by distributed microservices. Applications and microservices run in containers, and DC/OS packages are used to provide stateful and big data services.[1]

The benefits of using DC/OS and the Modern Enterprise Architecture are both tactical (improved reliability, better resource utilization, and faster software development) and strategic (collecting and extracting more value from data, having flexibility to deploy on-cloud or on-premises hardware using open source technologies).

In the central part of this report, I explain what DC/OS is and how it works. This explanation introduces the internal components of DC/OS in enough depth that you should be able to run applications on DC/OS without it seeming magical or mysterious. In the final chapter, I describe specific approaches that you can use with DC/OS to build, deploy, and operate software applications.

This report is intended for the principal users of DC/OS:

- System administrators responsible for the operation and uptime of applications and services
- Software engineers responsible for building applications and services to run on DC/OS
- Systems architects responsible for the design of systems and computing infrastructure.

This report also should be useful for you if you have any of these roles: DevOps, AppOps, QA, product manager, project manager, CTO, or CEO. For the technical sections of the report, I assume that you have experience in building and running networked (client/server) applications and using Linux.

1 *http://bit.ly/2nkXF5O*

If you read this report from cover to cover, you should learn enough to identify situations in which DC/OS could be used and what benefits it could bring. If you are interested in the details of how DC/OS works but not why you should use it, you can skip the first and last chapters and concentrate on the central part of the report.

## Glossary

The majority of the terminology used in this report is taken from the DC/OS documentation (available at *https://dcos.io/docs/1.8/overview/concepts/*). I recommend using this documentation as a reference when reading the technical sections of this report.

For now, though, there are some terms that have fairly flexible meanings in general use, but in this report, I use them in very specific ways:

- *Server* is used only to mean a software application that responds to requests from other applications.
- *Node* is a single virtual or physical machine running a Linux OS on which a Mesos agent or Mesos master process runs. DC/OS nodes are networked together to form a DC/OS *cluster*.
- *Operations* is used to refer to the activities and responsibilities of keeping a software system up and running in a live environment. *Operations* tasks are typically carried out by *systems administrators*, although different organizations use different practices or terminology.
- *Software development* is used to refer to the activities and responsibilities of creating new software or making changes to existing software. *Software development tasks* are typically carried out by *software engineers*, although different organizations use different practices or terminology.

CHAPTER 2

# Why Do We Need Modern Enterprise Architecture?

In this chapter, we explore the reasons that have motivated people to develop and use systems like DC/OS. Examples of similar systems are Google's Borg cluster-management system and tools like Kubernetes or Docker Swarm. These allow software-defined systems to control and run tasks on clusters of computing nodes (which can be virtual or physical). The reasons for the development of these systems are diverse including organizational, infrastructure, and application requirements.

We'll explore each of the different areas, and as we go through each, I will pick out specific requirements that I think DC/OS and Mesosphere's Modern Enterprise Architecture (MEA) are addressing. If you think that you have some if these requirements, you might benefit from using DC/OS.

A common question I hear—and one that I faced myself when I began considering using DC/OS—is this: "I have been making software applications successfully for years without DC/OS: what has changed that means I should change my approach?"

Here are my personal reasons for adopting DC/OS:

- The operational requirements (reliability, performance, connectivity) of the internet-connected applications I was building have changed dramatically over the past five years.

- Data (storage, collection, and analysis) has become of paramount importance and great value to organizations and the technical requirements to support machine learning and artificial intelligence (AI) technologies required a change in the technologies and approaches that I was using.

Let's take a step back and look at the broader changes that have motivated the development of DC/OS and similar systems.

## Highly Connected World

We live in a highly connected world,[1] and the expectations that people have of this connectivity are higher than they have ever been: businesses and consumers expect around-the-clock access to high-quality information, analysis, and services.

To meet the expectations of users, organizations must build and operate interconnected, always-on applications that a range of platforms can consume. Connected devices now include not only phones and PCs, but also electricity meters, refrigerators, and shipping containers. Systems are communicating more data, more frequently, and using more platforms than ever before. Accordingly, organizations need their systems to be scalable, highly available, and resilient.

Because consumers have high expectations and multiple ways of accessing services, even a simple consumer or business software product can require multiple connected services that interact with one or more stateful record stores. It is no longer enough for a business to have a good website, they also want the following:

- Device-specific apps that work with the following:
  — Smartphones
  — Smartwatches
  — Virtual Reality (VR)
- Service-specific integrations with entities such as these:
  — Major providers such a Google or Microsoft
  — Personal services such as Facebook and Twitter

1 *http://bit.ly/2oe6NXf*

    — Business software such as SalesForce, Xero, and Sharepoint

- New ways of interacting with users:
    — Virtual assistants like Alexa, Siri, and OK Google

    — Chatbots

    — Augmented Reality (AR)

To improve decision making and develop their competitive advantage, businesses want to collect and analyze information about these frequent and increasingly complex interactions. This requires investment in business processes, technology, and application development. Making the best use of data requires adopting big data, fast data, and machine learning strategies.

Building applications for this highly connected environment requires the ability to rapidly develop new software and update existing applications without introducing bugs or affecting reliability. Software development and operational strategies have emerged to facilitate this, such as Continuous Integration (CI), A/B testing, Site Reliability Engineering (SRE), Service (and microservice)-Oriented Architectures (SOA), and Agile development methods.

From this section, I can list these specific requirements that the MEA must have to be useful in our highly connected world:

- Can scale to support tens of thousands of simultaneous connections
- Can scale to support tens of thousands of transactions/second
- Resilience to expected failures (loss of nodes or a network partition)
- Fast, large volume (terabyte–petabyte scale) data collection and storage
- Fast, arbitrary analytics on live and stored data
- Support for modern software development methodologies
- Support for modern operational practices

From this list, you can see that the requirements I have for the MEA are not just about specific technical details (such as the support for simultaneous connections). It also needs to meet the broader requirements of teams that work with it (such as supporting the software development methodology). In the next sections, we'll investigate some of the different areas that are affected by the MEA.

## Operations

It takes more to run an application in production than installing some software and starting applications. For operators, their job truly begins on day two—maintaining, upgrading, and debugging a running cluster without downtime.[2]

In this report, I am using "operations" as a term to refer to all the tasks that arise to keep applications and services up and running. Traditionally, system administration has involved routine manual intervention to keep systems functioning correctly. These operational approaches have had to evolve to meet the needs of always-on, highly connected modern systems. Advanced operational approaches have been developed coining terms such as *Day 2 Ops*, *DevOps*, and the aforementioned SRE. These approaches use software to define system configuration and automate operational tasks.

SRE is a term that originates from Google, and the SRE approach is set out in an excellent book that is available online for free.[3] The aim of SRE is to deliver an optimal combination of feature velocity and system reliability. The responsibilities of SRE, as defined by Google, are availability, latency, performance, efficiency, change management, monitoring, emergency response, and capacity planning.

That provides a good summary of the typical concerns of an operations team. Operations is highly technical, and the efficiency and effectiveness of the operational team is dependent on many details of the systems that it uses and maintains. It is essential that an MEA addresses operational requirements and supports a range of operational approaches. Here are key operational tools and practices:

- Containerization
- Orchestration
- Dynamic service discovery
- Infrastructure as code

2 *https://dcos.io/blog/2016/join-the-dc-os-day-2-operations-working-group/index.html*

3 *https://landing.google.com/sre/book.html*

- Continuous integration
- Continuous deployment

It is neither effective nor scalable for daily operations task or failure handling to be manual processes. Operational teams need systems that can automatically respond within milliseconds to problems that arise so that they are self-healing and fault tolerant. To provide reliability and meet uptime requirements, the MEA should include not only redundancy but also capacity to correct faults itself. To fully realize the benefits of operational automation, teams need to be able to program systems to work with their in-house applications and to perform tasks according to their specific business requirements. This ability to program and customize operational systems behavior is another requirement I have of the MEA.

## Application Development

Businesses want their software development teams to produce new applications and features with shorter timescales to keep up with technology developments and fast-changing usage patterns. Examples of recent developments that prompt organizations to want to develop new applications are AR and VR and an explosion of smart devices.

To rapidly develop applications, software engineering teams have widely adopted methodologies focused on maintaining a high speed of development. At the same time, it is also necessary that software meet high standards of reliability and scalability. To deliver reliable, scalable applications and develop quickly, software engineers want to make use of reliable high-level abstractions, which they consume as services through SDKs and APIs. Here are some examples of these high-level services:

- Databases
- Message queues
- Object storage
- Machine learning
- Authentication
- Logging and monitoring

- Data processing (map-reduce)

By using high-level abstractions, software engineers can develop new applications more quickly and efficiently. Using well-known and well-tested systems for underlying services can also contribute to the reliability and scalability of the resulting application.

Having access to a wide range of sophisticated abstractions improves both software development and system operation. For example, if software engineers have access to a graph database, a transactional relational database, and a highly concurrent key-value database, they can make use of each database for appropriate tasks. Choosing the right tool for the job makes both development and subsequent operation much more efficient than attempting to force tasks onto an unsuitable service.

To allow fast and versatile application development, the MEA should allow us to easily use a range of high-level service abstractions provided by well-known, reliable, and scalable implementations.

## Hardware and Infrastructure

Any organization deploying an enterprise application needs to consider what computing infrastructure it will use—predominantly, this decision is focused on computing and network hardware but can include many other concerns. Deciding on what infrastructure to use is an extremely significant and difficult decision to make for many businesses, and choices typically have long-lasting consequences.

Before we go further into this topic, it is important to stress that DC/OS can run on a wide range of computing infrastructures, including on-premises datacenters and cloud platforms; it does not require you to use a particular infrastructure.

Cloud computing platforms provide a spectrum of services, from bare-metal servers to high-level abstractions like databases and message queues, as described in the previous section. Examples of companies that provide these services include Amazon Web Services (AWS), Google Cloud, Microsoft Azure, RapidSwitch, and Heroku.

The major cloud providers are widely used; have extremely good Service-Level Agreements (SLAs); provide a range of sophisticated management and configuration tools; and offer myriad pricing options, including pay-as-you-go. Using cloud platforms has many advantages for organizations compared with the alternatives. For the majority of organizations, building and operating all of the necessary infrastructure on-premises is a significant undertaking and often requires making infrastructure, software, or architectural design compromises to use fewer or less-sophisticated devices and tools in order to be feasible.

There are many benefits to using cloud platforms but there are also drawbacks:

- Problems of vendor lock-in
- Difficulty of compatibility or interoperation with existing on-premises systems
- Lack of transparency about how services are implemented
- Information security concerns
- Lack of control over service provision and development
- Regulatory restrictions
- Specialized performance or hardware requirements
- Financial considerations

In some cases, to avoid dependence on a single provider, some organizations set up systems to use multiple platforms or use a combination of on-premises and cloud platforms, which adds complexity.

So, we will add the requirement that the MEA should not force you to use a specific cloud or on-premises infrastructure. It should work equally well on a range of computational infrastructure. Furthermore, it should allow you to use the same configuration and management tools, irrespective of the underlying infrastructure provider so that it is possible to use multiple providers easily.

## Analytics, Machine Learning, and Data Science

Modern, highly connected businesses and software systems have access to huge amounts of information. In recent years, the scope for software systems to collect, analyze, and ultimately generate intelligence from data has increased exponentially.

Effective collection and exploitation of data from software systems is being used by businesses to build significant competitive advantages. To make the most from the opportunities requires systems to have the capacity to collect, store, and analyze large volumes of data. Subsequent to analysis, organizations need to incorporate the results of that analysis into the operation and decision-making process.[4]

Real-time analytics is most commonly associated with advertising, sales, and the financial industries, but it is now finding uses in an entire range of applications; for example, to provide system administrators with Canary metrics[5] or using machine learning and predictive analytics to automatically scale infrastructure and services in datacenters.[6]

An ideal machine learning system automatically analyzes information from live systems and uses the results to make predictions and decisions in real time. To realize the value from data, an MEA must treat data collection, storage, and analytics as principal concerns fully supported by the system architecture and incorporated into software development and system operation.

Many existing application architectures such as the 12-factor app were developed to address the needs of applications that run as services and use localized, transactional data architectures (such as SQL databases) for storing data. In these data architectures, analysis is performed as a separate function, typically one removed from live systems requiring Extract, Transform, and Load (ETL) processes and separate data warehouse infrastructure. These systems are costly, difficult to adapt to changing data models (slowing development), and, most important, take a long time to close the loop between data collection, analysis, and action. A data-driven service architecture still has all of the requirements of an architecture such as the 12-factor app, but it has additional requirements related to the automation of collection and analysis of data.

The requirement that we have for the MEA is that it will support the collections, storage, and analysis of large amounts of data and that it will allow us to easily use the tools and techniques of modern data

4 *https://medium.com/@Zetta/the-intelligence-era-and-the-virtuous-loop-784e9928f51b*

5 *http://techblog.netflix.com/2013/08/deploying-netflix-api.html*

6 *https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/*

science, such as distributed storage and computing systems (Hadoop, Spark, and so on).

## Business Value

*Back when IT was just infrastructure, your tech stack wasn't a competitive business asset. But when you add data into the equation—that changes the game. For example, both Netflix and HBO create original programming and distribute their content. Only Netflix is able to analyze viewer behavior in detail and use that to inform programming and content creation.*

—Edward Hsu, VP product marketing, Mesosphere

Software systems and computing infrastructure have been seen by many organizations as a cost of doing business—a cost similar to office leases or utility bills. But for successful technology companies, software systems and computing infrastructure are valuable business assets. Time and money well invested can provide a valuable return or competitive advantage. The competitive advantage can be realized in many ways including from exploiting data, as illustrated in the quote opening this section, from taking advantage of new technologies or from being able to deliver new and more sophisticated applications faster than competitors.

The easiest benefit for businesses to realize by improving their system architecture is in improvements to the performance of teams that work directly with software and systems in areas such as the following:

*Data collection and analysis*
: Increasing the value extracted from data. Reducing associated infrastructure and support costs.

*Software development*
: Increasing feature velocity. Making more data-driven decisions.

*Operations*
: Improved uptime and reliability. Reduced operational costs. Faster recovery times.

These are the topics that have been discussed in the previous sections of this chapter. Taking a more holistic view, there are other strategic business considerations when making technology choices:

- Avoiding vendor lock-in

- Human resource considerations
- Control and visibility of infrastructure
- Information security and regulatory requirements

The majority of the concerns covered in this section are about managing business risk rather than meeting a specific technical requirement. The weight that you apply to these risks when making architecture choices will depend on your beliefs about risks and your tolerance for accepting risks in different areas.

## Vendor Lock-In

Vendor lock-in occurs when a business is heavily reliant on a product or service that is provided by a supplier (vendor). An example is the reported reliance of Snapchat on Google Cloud, as Snapchat's S-1 filing (part of its IPO documentation) states:

> Any disruption of or interference with our use of the Google Cloud operation would negatively affect our operations and seriously harm our business.

Lock-in like this poses a risk because the supplier might stop providing or change the nature of its services, or the supplier can take advantage of the locked-in customer by increasing the price that it charges. Vendor lock-in usually arises because there are no alternate providers or there are significant technical or financial costs to switch to an alternate provider. With many technology products, numerous small technical differences between similar services mean that there can be significant switching costs, and so vendor lock-in is a common risk when making technology choices. For example, cloud platforms such as AWS, Azure, and Google Cloud Platform provide similar services, but there are differences between the APIs, SDKs, and management tools for those services, which means that moving a system from one to another would require significant software engineering work.

Technology lock-in occurs when a business is heavily reliant on a specific technology; for example, a company can become locked-in to a particular database software because it contains large amounts of critical business data, and moving that data to an alternative database software is too difficult or expensive.

A situation which is less commonly mentioned is when an organization becomes locked-in to using internal services such that it has high switching costs to transition to alternatives. Sometimes, this might be technology lock-in, but it is in many cases more similar to vendor lock-in except that the vendor is a department internal to the company. This is a situation that our architecture should avoid and discourage from occurring—if it facilitates on-premises provision of products and services, it should also allow for easy transition to external products and services. A common example of this is businesses that are locked-in to the use of on-premises IT infrastructure and face significant switching costs to transition to cloud infrastructure despite many potential advantages to doing so. The best way to avoid lock-in is to choose an architecture and systems that keeps switching costs to a minimum.

Lock-in is a situation that businesses want to avoid and so can be a significant concern when making architecture choices. In some cases, organizations put a lot of money and effort into setting up systems so that they can use multiple technology providers to avoid reliance on a single supplier.

Because of this, the MEA should minimize vendor and technology lock-in. Specifically, for a software system, this means that the architecture should allow us to use a range of different software systems to provide services (databases, message queues, logging, and so on) and it should make it easy to switch between different providers.

## Human Resources

Choosing a technology, however technically appropriate, for which there are few competent or experienced engineers and/or administrators available creates risks:

- Will it be possible to hire or subcontract sufficient engineers to make use of the technology?
- Can the organization develop sufficient expertise to maintain the technology after it's in place?

In some cases, making bold and unusual technical choices can have significant benefits, usually when the advantage of technical performance in a specific area is more important than other concerns. In general, however, staffing risks can make a more common technology with a larger or less-expensive talent pool a better choice than an unusual choice, even if it is a better technical fit. Following are some human-resource concerns:

- Skills and experience that exist within the organization
- Cost and availability of skills and experience
- Projection of future cost and future availability of skills and experience

Technology and architecture choices can have dramatic effects on staffing requirements by allowing tasks to be automated or outsourced. In particular, modern software orchestration systems (such as those provided by cloud platforms and DC/OS) automate or facilitate automation of an entire range of tasks, particularly operational tasks. There is also massive scope in making use of improved data architectures and machine learning software to reduce the workload associated with analytics and data science.

The MEA should allow us to automate operational and data tasks, and the technologies used should have good availability of skilled and experienced engineers and operators so that it is easy for the business to find competent staff.

## Control

Regardless of contracts and SLAs, provision of services by third parties exposes businesses to certain risks. In some extreme cases, providers have discontinued services, choosing to break contracts rather than continue unprofitable activities. In other cases, customers have lost access to systems and infrastructure when the business providing them has failed to pay its bills (e.g., for power or network access) or filed for bankruptcy. A more common occurrence is that periodically providers update their services, changing tools and interfaces, which forces users to spend engineering effort to change their applications to use the updated tools/interfaces.

For some businesses in regulated industries, there might be concerns about the ability of third parties to comply with regulatory requirements, particularly regarding privacy and security.[7]

The MEA should work both for businesses that want to exercise a high level of control over their infrastructure/systems, but it should

7 There are people who argue that a specialized infrastructure provider is able to do a better job on security or regulatory compliance than in-house solutions. I am not making the case either way—I'm just explaining that this is a position some businesses take.

not create extra work for those who are more easy-going or who want to outsource infrastructure provision to specialist third parties.

## Regulatory and Statutory Requirements

Information systems and companies that operate them are subject to legal and regulatory requirements. Many countries have privacy or data protection laws, and certain industries or business requirements have more stringent requirements. Here are some examples:

- HIPAA affects personal medical and healthcare-related information in the United States.
- PCI DSS has requirements for systems that handle credit card and other personal banking information
- European Union Data Protection rules apply to Personally Identifiable Data in Europe

Here are some examples of requirements resulting from regulation:

- Localization of data; for example, EU Data Protection Rules place restrictions on the transfer of personal data outside of the EU.
- Logging and audit; for example, PCI DSS requires that systems log access to network and data, and it should be possible to audit those logs.
- Authentication and access control; for example, many information security regulations require that users should be appropriately authenticated to access data.

Our enterprise architecture should not prevent meeting these or other regulatory requirements. It should make typical requirements such as localization, auditing, and authentication straightforward to enforce and manage.

## Chapter Conclusion: MEA Requirements

I have provided some context for the situations in which DC/OS is commonly used and identified a range of requirements for the MEA to meet, from technical requirements, such as the ability to deliver internet-connected applications that can handle high transaction rates, to broader requirements, such as facilitating operational and software development methodologies. To recap, the key requirements from this chapter are that DC/OS should do the following:

- Meet the technical needs of modern, internet-connected applications including transaction volume, horizontal scalability, and durable persistence.
- Deliver state-of-the-art reliability and consistency, within the bounds of CAP theorem (for distributed systems) and limitations of networked applications.
- Facilitate high-volume data collection and storage, fast analysis, and machine learning.
- Enable high productivity in teams that use the system—software developers, data scientists, and system administrators.
- Be compatible with multiple infrastructure options and have low switching costs associated with moving an operational system from one infrastructure to another to avoid vendor lock-in.
- Be compatible with a range of technologies for software development, allowing for concurrent use of different technologies and minimal switching costs to avoid technology lock-in.
- Be realistic and cost effective in terms of computational and human resources to deliver and operate for both small and large organizations.

It should be clear that any architecture that meets these needs will be a distributed system designed to run across multiple individual machines capable of handling a diverse workload. It is my belief that these requirements are not well met by most existing systems, and that by meeting these requirements, DC/OS and the MEA is significantly better for organizations building networked software applications and services than existing solutions.

These are bold claims, and some of the requirements might seem too broad or to demand too much flexibility to be practical. For example, the requirement to work well if infrastructure is running on a cloud platform or in an on-premises datacenter—these are radically different environments, and you might be concerned that any system that works in both gets the benefits of neither. The proposition that a single enterprise architecture can meet so many diverse needs can sound unrealistic—you might think that there is too much variation in different organizations to allow us to come up with a single solution or pattern that will work well for everyone.

These challenges seem daunting, but do not worry! There are many examples of technological developments that solve problems in seemingly very different conditions. Consider technologies that provide powerful abstractions such as TCP/IP Networking, which is used to send control signals to Mars Rovers as part of a network with a handful of endpoints separated by huge distances, with high latency, and low bandwidth. The exact same technology is used to send cat videos from YouTube to my laptop—a relatively short distance as part of a network with billions of endpoints predominantly composed of low-latency, high-bandwidth connections.

In the next chapters, I explain what DC/OS is in detail and how you can use it to meet the requirements set out so far. Analogously to the example of TCP/IP networking, DC/OS is a technology that provides powerful abstractions that can be applied to solving problems in a range of different circumstances and environments. There are software systems, such as those used for controlling the avionics of a fighter jet, for which this architecture would not be appropriate. But for use by businesses and other organizations to run networked software systems, typically providing some services over the internet and maintaining some internal state, the MEA using DC/OS is an excellent choice.

CHAPTER 3

# Understanding DC/OS

In this chapter, I'm going to introduce Data Center Operating System (DC/OS) and explore the high-level abstractions that DC/OS provides. I will also describe some of the services, such as Cassandra, Kafka, and Spark, that you can run on DC/OS.

Many introductions to DC/OS focus on describing what DC/OS *can do* rather than what it *is*. At the very beginning of this report, I defined DC/OS like this:

> DC/OS is a system composed of Linux nodes communicating over a network to provide software-defined services. A DC/OS cluster provides a software-defined platform to which applications can be deployed and can scale to thousands of nodes in a datacenter. DC/OS provides an operational approach and integrated set of software tools to run complex multicomponent software systems and manage the operation of those systems.

Like most other descriptions, that focuses on what DC/OS *does* rather than what it *is*. In this section, I will unpack a bit more what this "system composed of Linux nodes communicating over a network" is:

> DC/OS is a system made up of different software components, written in a range of programming languages, running on multiple Linux nodes in an appropriately configured TCP/IP network. There are many different DC/OS executables (components) running on each of the nodes along with their dependencies. Each of these DC/OS components provides some specific function or service (for example internal load balancing). DC/OS is the system that results

from the combination of these individual services working together.

DC/OS has been built based on lessons learned at some of the most successful tech companies, using the most advanced systems and infrastructures in the world. Among these companies are Google, Twitter, Airbnb, Uber, and Facebook. The approaches used in DC/OS have often been developed by companies to manage phenomenal growth and to operate at global scale. In some cases, the solutions used in DC/OS are radically different to those used outside of leading technology companies. DC/OS allows us all to work in ways similar to these leading companies, but, depending on your background and experience, you might find some of these approaches unusual at first.

Depending on your experience and area of responsibility, you might be concerned with specific aspects of DC/OS. Let's consider this from the two main perspectives of operations and of development:

- From an *operational* point of view, we can describe DC/OS as a system for software-defined configuration and automation of complex, interdependent applications running on clusters of machines that can run on any networked Linux nodes.
- From a *software development* point of view, we can describe DC/OS as a platform that allows us to develop distributed systems composed of applications with access to a selection of core platform services that provide high-level abstractions including persistent storage, message queues, and analytics.

## Getting Started with DC/OS

The best way to begin using DC/OS is to think of it as the one-and-only application that you need to explicitly run on all nodes.

NOTE

There are some tasks that you don't do inside DC/OS: basic Linux configuration (you need Linux to be running correctly before you install and run DC/OS) and most low-level security-related tasks (iptables, restricting accounts, file permissions, Linux software updates/patches, antivirus, and intrusion detection).

Nodes running DC/OS communicate with one another (correctly configured, of course) to create a cluster of computational resources that can execute arbitrary tasks. After a DC/OS cluster is up and running, you should then run and manage all other applications and tasks via DC/OS.

To be clear, DC/OS is not a configuration/orchestration tool similar to Puppet, Chef, Ansible, or Cloud Formation; it is a cluster-scale operating system that allows software to define and to manage complex configuration of large numbers of nodes, among other things

**NOTE** There is still a place for these tools in configuring nodes with DC/OS in the first place, but this is much simpler than using them for entire cluster configurations.

The DC/OS installation will detect automatically the CPU and RAM available to each node when it is installed. However, if you have node instances with other different properties or capabilities that you will need to use to determine application placement (for example, some nodes might be equipped with solid-state drive [SSDs]), you can configure them at setup time either as Mesos attributes or by assigning machine resources to a Mesos ROLE.

The instructions for installing the latest version of DC/OS on various platforms are available online at *https://dcos.io*. Whatever process you use to set up your nodes, after they are up and running, everything related to deploying and managing your applications is handled through DC/OS.

## How DC/OS works

In DC/OS nodes are either masters or agents. DC/OS is made up of a number of different components. Each component is a separate executable application, and all DC/OS components are run as systemd units

**NOTE** systemd is a part of a number of Linux distributions and it is the main dependency of DC/OS.

Depending on the nature of the DC/OS node (master or agent, which you defined when you installed DC/OS on that node), a slightly different combination of DC/OS components will run. All masters run the same set of components and all agents run the same set of components, so there are only two node system configurations in a DC/OS cluster.

Nodes are configured to run DC/OS by copying the DC/OS component application files onto the node and then configuring systemd to run the appropriate components. This is done automatically in the installation scripts provided by Mesosphere.[1]

NOTE

For public clouds such as AWS and Azure, there are deployment templates that you can use.[2]

## Master Nodes

Master nodes act as coordinators for the cluster and durably record the configuration of the cluster. A leader is chosen dynamically from among the available masters using elections carried out on ZooKeeper. The leadership model is used so that changes to the state of the cluster can be synchronized. Changes to the cluster state are carried out by the leading master instance and duplicated to a quorum of the master nodes by using Zookeeper. Having multiple master instances provide redundancy and duplication of the persisted state —if the leading master fails, a new leader will automatically be chosen from the available master nodes. Having multiple masters does not allow for any significant distribution of workload because the leading master does the majority of the work.

NOTE

The *number* of masters has no impact on the scalability or performance of DC/OS or Apache Mesos. *There is only ever one leading master* that governs operations across the cluster. If you have five masters, you are able to tolerate multiple concurrent master failures, and there is little benefit to adding more master nodes.

1 *https://dcos.io/docs/1.8/administration/installing/custom/advanced/*

2 *https://dcos.io/docs/1.8/administration/installing/cloud/*

Masters are responsible for monitoring the state of the cluster and assigning tasks to agent nodes. Masters assign tasks to agents to ensure that the operational state of the cluster matches the desired (configured) state as far as possible.

### Mesos Masters

DC/OS uses Apache Mesos for task scheduling. DC/OS masters are also the masters for the underlying Mesos cluster included in DC/OS. To illustrate the number of masters you might need, Twitter runs 30,000 nodes in a single Mesos cluster with just five Mesos masters.

The usual limiting resource on masters as cluster size increases is memory, because masters build the state of the cluster in memory, so it is most important that masters have sufficient memory to do this or you will see performance problems.

## Agent Nodes

Agent instances notify the DC/OS masters of their available resources. The masters allocate those resources to tasks, which the agent is instructed to execute. DC/OS uses Apache Mesos internally to perform resource allocation and task scheduling. The resources that agents make available via Mesos are CPUs, GPUs, memory (RAM), ports, and disk (storage).[3] You can allocate resources to specific roles, restricting their use to specific applications; otherwise, if no roles are specified, resources are used for any applications.

You can add agent instances to a DC/OS cluster at any time. When a new agent node is provisioned, it will register itself with the leading master. During registration, the agent provides the master with information about its attributes and the resources that it has avail-

3 *http://mesos.apache.org/documentation/latest/attributes-resources/*

able. After it is registered with the master, the agent will begin receiving task assignments.

### Custom Node Attributes

You can give agent nodes custom attributes, which are advertised alongside the available resources and can be used by task scheduling code. A commonly supported attribute is "rack," which you can set to a string value indicating which physical rack a node is located in inside a datacenter. Schedulers can use this attribute to avoid placing instances of the same task in the same rack but to distribute them over multiple racks. This is desirable because an entire rack might fail at once.

## Mesos Tasks and Frameworks

Mesos[4] is the underlying task scheduler that is used internally by DC/OS. Mesos is automatically set up on nodes as part of DC/OS installation. Mesos is responsible for the low-level assignment and execution of tasks on agent nodes. When Mesos runs a task on an instance, it uses cgroups to restrict the CPU and RAM that is available to that task to the amount specified by the scheduler. This allocation prevents tasks from consuming excess resources to the detriment of other applications on the same node.

Tasks are provided to Mesos by frameworks. A Mesos framework is an application that uses the Mesos API to receive resource offers from Mesos and replies to resource offers to instruct Mesos to run tasks if the framework requires tasks to run and the offer has sufficient resources.

To recap: agent nodes provide resources to Mesos masters. Masters coordinate offering unused resources to frameworks. If frameworks want to use resources, they accept resource offers and instruct the masters to run tasks on the agents. Following are the resources that Mesos can manage:

- CPU
- RAM

4 *http://mesos.apache.org/documentation*

- Ports
- Persistent volumes
- GPU

Mesos combines the resources of a cluster of agent nodes into an abstract pool of computing resources. The use of cgroups makes it possible to safely run multiple applications on the same node without the risk that resource contention will cause problems ("noisy neighbor" syndrome).

**NOTE** Mesos does not specify the semantics for handling task failure or loss of an agent;[5] this must be handled by the framework.

Here are the benefits of the task abstraction provided by Mesos:

- Running multiple workloads on a single cluster increases resource utilization[6]
- Having a handful of base system configurations (master, public agent, private agent) makes management of nodes and OS configuration much simpler than having a per-application system configurations
- You can use frameworks to automate complex operations tasks, including failure handling and elastic scaling. Frameworks can each implement their own custom logic with very few constraints.

## Mesos Attributes and Roles

In addition to the resources that Mesos agents make available, Mesos allows agents to describe individual properties in two ways. The properties that nodes can use are *attributes* and *roles*. Attributes are key-value pairs that are passed along with every offer and can be used or ignored by the framework. Roles specify that some resources

5 *http://mesos.apache.org/documentation/latest/high-availability-framework-guide/*

6 *https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf*

on the agent node should be offered only to frameworks that have the same role.

This information is passed on in the offers made to frameworks so that they can decide to make use of machines based on their particular properties.

Attributes can be used, for example, to specify the rack and row in a datacenter where the machine is located. This can be used by frameworks to ensure that their tasks are well distributed across the datacenter so that it will not be vulnerable to failure of a single component such as a switch or power supply. Use of attributes does not disrupt frameworks that are not aware of them because they will ignore them.

Allocating resources (e.g., CPU and RAM) to roles prevents frameworks that do not share the role from accessing those resources. By reserving all resources on a node for a role, tasks that do not belong to the associated frameworks are prevented from running on that node at all. For example, a typical DC/OS setup will have some nodes in a subnet with public IP addresses, whereas the majority of the nodes are placed in a private network DMZ only accessible from within the datacenter. In this setup, the machines with public IP addresses have all of their resources (CPU, RAM) assigned at setup time to the “public agent” role in Mesos. This means that only tasks which are configured with the “public agent” role are executed on these machines. This process is called *static partitioning*.

**NOTE**

Although the example uses networking, setup of machines static partitioning can be done for a range of reasons; for example, reserving all machines with GPUs to a specific role. Static partitioning does not have anything to do with network partitioning.)

## Other Mesos Functionality

In addition to per-agent resources, Mesos has developmental work to support *external resources*[7] that are not tied to specific nodes but can be allocated to specific tasks. The proposal for this suggests that use cases could include network bandwidth, IP addresses, global ser-

7 *https://issues.apache.org/jira/browse/MESOS-2728*

vice ports, distributed file system storage, software licenses, and SAN volumes

NOTE

Some of these proposals are under development at the time of writing. DC/OS is undergoing rapid development so you should check the latest DC/OS and Mesos documentation to understand what additional functionality is available.

Mesos uses health checks[8] to monitor the health of a task. Mesos health checks can be shell commands, HTTP, or TCP checks. The details of the health checks to run on a task are set by the Framework scheduler.

NOTE

If no health checks are set, Mesos monitors tasks as processes and will notice if they stop or crash but will not notice if they are still running but unresponsive.

## DC/OS Abstractions

As a DC/OS user, we do not have to work at the low level of abstraction provided by Mesos. DC/OS provides a selection of ways of running applications for common requirements. DC/OS also provides a core set of components (some of which run on MESOS), which provide complementary functionality to Mesos. There are three main methods for running applications in DC/OS: *apps*, *jobs*, and *packages*. Let's take a closer look at them:

*Apps*
: These are long-running applications run by Marathon. Marathon runs specified number of instances of the ap container and ensures the availability of the app by automatically replacing app instances in case of crashes, loss of a node, and other failures. Marathon also ensures the availability of the app during the deployment of a new version.

8 *https://github.com/apache/mesos/blob/master/docs/health-checks.md*

*Jobs*
Jobs are one-off or scheduled applications run by Metronome. Job instances are executed on the cluster according to the specified schedule and continue to run (and consume resources) until they either shut down or crash. Jobs are not restarted or reassigned in the case of failure, either of the job or the agent node running the job.

*Packages*
Packages are published app definitions for common services packaged for DC/OS. Packages can also include a DC/OS command-line interface (CLI) plug-in, which allows you to use the DC/OS CLI to manage the package

NOTE

It is possible to define packages that do not include an app, just a DC/OS CLI plug-in.

You can use packages to publish software to run on DC/OS. Package definitions can be published to public or private registries.

## Other DC/OS Components

The DC/OS system is made up of many different components (all open source) which together provide a reliable system that allows you to configure a cluster of machines to reliably run applications using powerful abstractions such as the aforementioned apps and jobs.

Here are some important DC/OS components that will be mentioned in this report:

- Zookeeper and Exhibitor
- Admin Router
- Metronome
- Marathon
- Mesos-DNS
- Cosmos

- Minuteman

There are many more components, which will not be mentioned in this report and provide services from utilization logs (history service) to IP network overlay for containers (Navstar). Details of all DC/OS components are available at *https://dcos.io/docs/1.8/overview/components/*. Studying the documentation for all the components is the best way to develop an advanced understanding of DC/OS.

## DC/OS Packages

You can use packages to run single-instance applications such as Jenkins or NGINX and to run Mesos frameworks to manage distributed system such as Cassandra or Kafka on a DC/OS cluster.

Mesosphere provides a public registry of packages[9] called the Mesosphere Universe. You can install and configure packages from the universe using the DC/OS GUI or the DC/OS CLI. As of this writing, there are more than 70 packages in the universe registry, including:

- Cassandra (provides its own Mesos framework)
- HDFS (provides its own Mesos framework)
- Jenkins
- Kafka (provides its own Mesos framework)
- Spark
- Zepplin
- MySQL
- NGINX
- Marathon-LB/HA Proxy

Packages that use their own Mesos frameworks run a scheduler application on Marathon as an app. The app registers with Mesos as a framework and then communicates directly with Mesos masters to schedule tasks independently of Marathon. For example, the Kafka

9 *http://mesosphere.github.io/universe/*

app communicates with the Mesos masters to schedule Kafka brokers as tasks on Mesos.

NOTE

It is possible to configure your DC/OS cluster to use a private package repository (an alternate universe) alongside or in addition to the Mesosphere universe.

Packages typically allow some degree of initial configuration, such as the following:

- Specifying the number of nodes in a Cassandra cluster
- Specifying the default sharding and replication of Kafka topics
- Specifying the number of name, data, and journal nodes in HDFS

Packages can provide an application-specific API and a DC/OS CLI integration. Typically, the CLI integration includes methods for checking on the health of the package and methods for altering the configuration of the package. Packages can also have persistent internal state (for example, using ZooKeeper to store custom configuration).

NOTE

Uninstalling a package might require manually removing persisted state from ZooKeeper and manually removing the framework from Mesos.

Packages that run their own Mesos framework take on direct responsibility for scheduling child tasks on Mesos. These packages must implement handling for all the error scenarios that might occur, from crashing tasks, to failure of an agent instance or a network partition. The advantage of this is that each package can tailor its behavior to the requirements of the application that it is managing. For example, a simple stateless application can start more tasks if an agent fails, whereas stateful applications such as Cassandra or

HDFS have more constraints and need to trade off consistency and availability in their failure handling.[10]

Published packages range from production-ready to highly experimental. Packages each have their own licensing terms, source code availability, and maintainers. For all of these reasons, I recommend that you investigate these properties for each package before using it.

Using packages on DC/OS allows you to easily add distributed services to your systems. Do you need a message broker? Install the Kafka package. Need a distributed NoSQL database? Install the Cassandra package. The benefits of this should not be underestimated: setting up a Cassandra cluster on bare metal could take months of work for some organizations.

In addition to providing a convenient and reliable way to install services, packages also provide automation and tooling that makes ongoing operation and maintenance of packages much easier. Using complex distributed systems without advanced operational tools such as DC/OS has usually required separate dedicated resources for each distributed/clustered service, and each system adds a significant amount to the operational team's workload.

By provisioning a DC/OS cluster with packages that provide high-level platform abstractions such as distributed key-value data stores (Cassandra), distributed computation (Spark), distributed pub-sub queues (Kafka), and distributed file systems (HDFS), software architects and engineers can easily and effectively build modern, high-performance systems. An important benefit of DC/OS is that you do not need to specify all of these services up front. You can be confident that, provided your cluster has sufficient capacity, it is easy to add services at any point.

Table 3-1 compares the capabilities provided by some packages against those provided publicly by AWS and built internally by Google:

10 *https://en.wikipedia.org/wiki/CAP_theorem*

*Table 3-1. Comparison of services available on different platforms*

| Service | DC/OS Package | Google | AWS |
|---|---|---|---|
| NoSQL data store | Cassandra | BigTable | DynamoDB |
| Distributed compute | Spark | Dremel | Elastic MapReduce |
| Distributed file system | Ceph, HDFS | GFS | EMRFS |

While comparing different systems, it is worth mentioning that the functionality of DC/OS is very similar to that provided by Google's Borg cluster management service.

## Package Examples

Packages in DC/OS provide a range of valuable services. As of this writing the available packages are changing rapidly, and the most mature and well-developed packages are Cassandra, Kafka, and Spark, so I will use each of these as examples. These are sophisticated distributed systems in their own right, so, because space is limited in this report, it will be a very simplified look at what is provided.

### Cassandra

Cassandra is an open source NoSQL database and Apache Software Foundation project. Cassandra is in production use at a number of major companies such as Apple and Netflix. This is the description of Cassandra from its Apache project page:

> The Apache Cassandra database is the right choice when you need scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Cassandra's support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages.[11]

The DC/OS Cassandra service[12] is an open source DC/OS package provided by Mesosphere. The Cassandra package includes a DC/OS CLI plug-in and HTTP API providing commands for common Cas-

11 *http://cassandra.apache.org*

12 *https://github.com/mesosphere/dcos-cassandra-service/tree/master/docs*

sandra administration tasks via the Cassandra Mesos framework scheduler. Here are key features of the DC/OS Cassandra service:

- Uses persistent storage volumes
- You can apply configuration changes and software updates at runtime
- Health checks and metrics for monitoring
- HTTP API and DC/OS CLI commands for the following:
  - Backup and restore of all data in a cluster
  - Automated cleanup and repair
  - Automated replacement of permanently failed nodes
- You can install multiple Cassandra clusters on a single DC/OS cluster
- You can configure the Cassandra service to span multiple DC/OS clusters (e.g., in different datacenters)

Installing and maintaining Cassandra on a DC/OS cluster is straightforward. After you install it, it's easy to begin using Cassandra as the storage layer for applications. DataStax provide high-quality, open source client libraries for Cassandra in a range of common programming languages.[13]

To begin reading and writing to Cassandra, grab the relevant client library and begin writing queries for your application. The example here is based on the Mesosphere Tweeter tutorial and shows a simple Tweet class that uses Cassandra for storing and searching tweet data:

```
require 'cassandra'

CASSANDRA_OPTIONS = {
    hosts: ['node-0.cassandra.mesos'],
    timeout: 300,
    consistency: :quorum,
}

# Tweet class that talks to Cassandra
class Tweet
  include ActiveModel::Serialization
```

13 *http://docs.datastax.com/en/developer/driver-matrix/doc/common/driverMatrix.html*

```
  @@cluster = Cassandra.cluster(CASSANDRA_OPTIONS)
  @@keyspace = 'tweeter'
  @@session  = @@cluster.connect(@@keyspace)
  @@generator = Cassandra::Uuid::Generator.new
  @@paging_state = nil

  attr_accessor :id, :content, :created_at, :handle

  def destroy
    @@session.execute(
        'DELETE from tweets WHERE id = ?',
        arguments: [@id])
  end

  def self.create(params)
    c = Tweet.new
    c.id = SecureRandom.urlsafe_base64
    c.content = params[:content]
    cassandra_time = @@generator.now
    c.created_at = cassandra_time.to_time.utc.iso8601
    c.handle = params[:handle].downcase
    @@session.execute(
        'INSERT INTO tweets (kind, id, content, created_at, han
dle) ' \
      'VALUES (?, ?, ?, ?, ?)',
        arguments: ['tweet', c.id, c.content, cassandra_time,
c.handle])
    c
  end

  def self.find(id)
    tweet = @@session.execute(
        'SELECT id, content, created_at, handle FROM tweets
WHERE id = ?',
        arguments: [id]).first
    c = Tweet.new
    c.id = tweet['id']
    c.content = tweet['content']
    c.created_at = tweet['created_at'].to_time.utc.iso8601
    c.handle = tweet['handle']
    c
  end
end
```

Example Ruby class that uses Cassandra on DC/OS:

- *https://github.com/mesosphere/tweeter/blob/master/app/models/tweet.rb*

## Kafka

Kafka is an open source distributed streaming platform and Apache Software Foundation project. Kafka is in production use at a number of major companies such as LinkedIn and IBM. This is the description of Kafka from its project page:

> Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies.

I personally cannot recommend Kafka highly enough. Using keyed messages on sharded queues and having multiple consumers for a single topic within a single consumer group is a very powerful and scalable technique. I have used Kafka to handle very large transaction rates and implement complex data-processing pipelines. Kafka is very robust, queues are persisted to disk and thus survive broker restart, and Kafka allows replication of queues across multiple brokers, providing redundancy so that data is not lost in case of an individual node failure.

Kafka can be used for a range of tasks:

- Messaging
- Activity tracking
- Metrics
- Log aggregation
- Stream processing
- Event sourcing
- Commit log

The DC/OS Kafka service[14] is an open source DC/OS package provided by Mesosphere. The package includes a DC/OS CLI plug-in and HTTP API providing commands for common Kafka administration tasks via the Kafka Mesos framework scheduler. Following are key features of the DC/OS Kafka service:

- You can install multiple Kafka clusters on a single DC/OS cluster

14 *https://github.com/mesosphere/dcos-kafka-service*

- Elastic scaling of brokers
- Single-command installation
- High availability runtime configuration and software updates
- Uses persistent volumes
- Support for logging and performance monitoring

Installing and maintaining Kafka on a DC/OS cluster is straightforward. After you install it, it's easy for software engineers to begin using Kafka from their applications. Kafka has an extremely broad support, with clients in more than 15 programming languages and integrations to capture events from a range of other programs. Kafka also has a Spark integration, which means that you can use it as a data source for Spark Streaming.

This example in Ruby shows how easy it is to set up a Kafka Producer[15] to record events:

```
KAFKA_TOPIC = 'page_visits'

KAFKA_OPTIONS = {
    seed_brokers: 'broker.kafka.l4lb.thisdcos.directory:9092'
}

kafka = Kafka.new(KAFKA_OPTIONS)

producer = kafka.producer

producer.produce(page_view_data.to_json, topic: KAFKA_TOPIC)
producer.deliver_messages
```

## Spark

Spark is a fast and general engine for large-scale data processing. Spark is an Apache Software Foundation project. You can use it to execute a range of computational tasks on data held in memory across a cluster of agents. Unlike Kafka or Cassandra, Spark is not a long-running cluster process. If no computation is being run, Spark does not need to use any resources on the cluster or run any tasks, so there is very little installation or maintenance associated with it. The main features provided by the DC/OS Spark package are log-

15 A basic example of a Kafka producer. This example is using a VIP to discover the seed broker. (*https://github.com/mesosphere/tweeter*)

ging of Spark jobs, a DC/OS CLI plug-in, and DC/OS GUI integration.

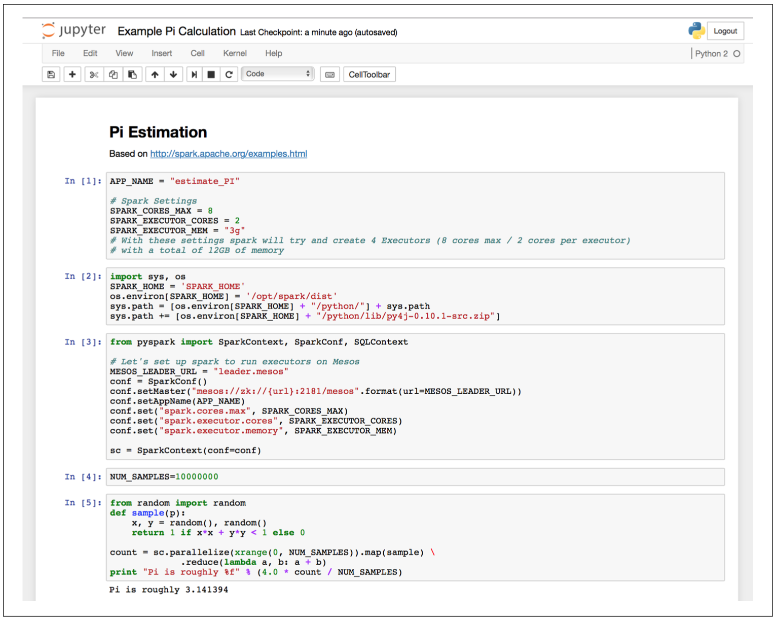
Spark can load data from a range of sources including HDFS, AWS S3, and Cassandra. Spark also has a streaming mode that you can use to process data from Kafka and other data streams.

You can write Spark jobs in Scala, R, or Python (although the R and Python interfaces do not have all the features that are available in Scala). To begin performing analytics on data stored in Cassandra, HDFS, S3, or a Kafka data stream, the best approach is to use one of the analytics notebook packages available in the DC/OS universe, such as Zepplin[16] or a Spark Jupyter Notebook.[17]

Notebooks (Figure 3-1) allow software engineers or data scientists to write and execute Spark jobs via a web browser. It is possible to begin running interactive data analysis in minutes, although system administrators should ensure that notebooks are appropriately configured for their cluster and security requirements. It is possible for Spark to take all available CPU or RAM in a cluster to execute analytics, and it is also possible for Spark to saturate network connections and overload data storage systems during read and write operations; however, configuring Spark appropriately will prevent these problems.

16 *https://github.com/mesosphere/dcos-zeppelin*

17 *https://github.com/andypetrella/spark-notebook/*



Pi Estimation

Based on http://spark.apache.org/examples.html

```
In [1]: APP_NAME = "estimate_PI"

        # Spark Settings
        SPARK_CORES_MAX = 8
        SPARK_EXECUTOR_CORES = 2
        SPARK_EXECUTOR_MEM = "3g"
        # With these settings spark will try and create 4 Executors (8 cores max / 2 cores per executor)
        # with a total of 12GB of memory

In [2]: import sys, os
        SPARK_HOME = 'SPARK_HOME'
        os.environ[SPARK_HOME] = '/opt/spark/dist'
        sys.path = [os.environ[SPARK_HOME] + "/python/"] + sys.path
        sys.path += [os.environ[SPARK_HOME] + "/python/lib/py4j-0.10.1-src.zip"]

In [3]: from pyspark import SparkContext, SparkConf, SQLContext

        # Let's set up spark to run executors on Mesos
        MESOS_LEADER_URL = "leader.mesos"
        conf = SparkConf()
        conf.setMaster("mesos://zk://{url}:2181/mesos".format(url=MESOS_LEADER_URL))
        conf.setAppName(APP_NAME)
        conf.set("spark.cores.max", SPARK_CORES_MAX)
        conf.set("spark.executor.cores", SPARK_EXECUTOR_CORES)
        conf.set("spark.executor.memory", SPARK_EXECUTOR_MEM)

        sc = SparkContext(conf=conf)

In [4]: NUM_SAMPLES=10000000

In [5]: from random import random
        def sample(p):
            x, y = random(), random()
            return 1 if x*x + y*y < 1 else 0

        count = sc.parallelize(xrange(0, NUM_SAMPLES)).map(sample) \
                    .reduce(lambda a, b: a + b)
        print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)

        Pi is roughly 3.141394
```

*Figure 3-1. An example interactive notebook running a Spark Job. In this case Spark is using Mesos directly to run executors and the DC/OS Spark package is not required.*

More examples are available at *https://github.com/dcos/examples/tree/master/1.8/spark/*.

There are many other packages available in the DC/OS Universe, and their state and sophistication are changing rapidly. You should ensure that you understand what assumptions packages you use make and what guarantees they provide. Here are a few other notable packages (as of this writing):

- Confluent-Kafka (Confluent's enterprise version of Kafka)
- DSE (Datastax' enterprise version of Cassandra)
- ElasticSearch
- HDFS
- Ceph
- Minio
- Jenkins
- Nginx

- Marathon-LB (HA Proxy)

You can view the available packages via the DC/OS GUI or at the mesosphere universe GitHub repository.

## DC/OS CLI

The DC/OS CLI is an application that you can install on any PC and which you can use to execute commands to control your DC/OS cluster. You can use the CLI interactively or you can script it to automate tasks.

The DC/OS CLI has a core set of functionality for managing nodes, installing or removing packages, and inspecting the state of the cluster, as demonstrated in Example 3-1.[18]

*Example 3-1. An example of core dcos functionality provided by the CLI*

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

# To install a package
# dcos package install [--options=<config-file-name>.json] <serv-
icename>

dcos package install --options=cassandra_configuration.json cassan-
dra
```

The DC/OS CLI functionality can be extended by packages. A package can include a CLI plug-in, which, when installed, adds package-specific commands to the CLI. For example, the Spark CLI includes commands such as those shown in Example 3-2 for submitting Spark jobs to the cluster:

*Example 3-2. Two examples of CLI plug-in behaviors provided by the Cassandra and Kafka packages*

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

# To backup a Cassandra cluster to S3
dcos cassandra --name=cassandraCluster1 backup start \
   --backup_name=latestCassandraCluster1 \
   --external_location=s3://my-s3-bucket \
```

18 *https://dcos.io/docs/1.8/usage/cli/command-reference/*

```
    --s3_access_key=S3_ACCESS_KEY \
    --s3_secret_key=S3_SECRET_KEY \


# To list kafka topics
dcos kafka --name=kafka topic list
# [
#   "topic1",
#   "topic0"
# ]
```

In this chapter, I have described how you can use DC/OS to create an environment for your applications that contains high-level services such as databases and message queues. You should now be familiar with the abstractions provided by DC/OS and have a basic understanding of the relationship between DC/OS and Mesos. You should begin to see that it is easy and fast to write powerful applications that make use of the provided services and that those applications could run in DC/OS.

I have introduced some of the packages that are available in the DC/OS universe, which you can use to provide services to your applications. At this point, you might be wondering how you begin running your own software on DC/OS. In the next chapter, we look at the options for deploying and managing your own applications and other important details related to running your own applications in DC/OS.

CHAPTER 4

# Running Applications in DC/OS

In this chapter, I explain the options that you have for running software on DC/OS, the services that DC/OS provides to your applications, and the details that you need know to build and run your own software on DC/OS. Figure 4-1 shows the Modern Enterprise Architecture (MEA) recommended by Mesosphere.

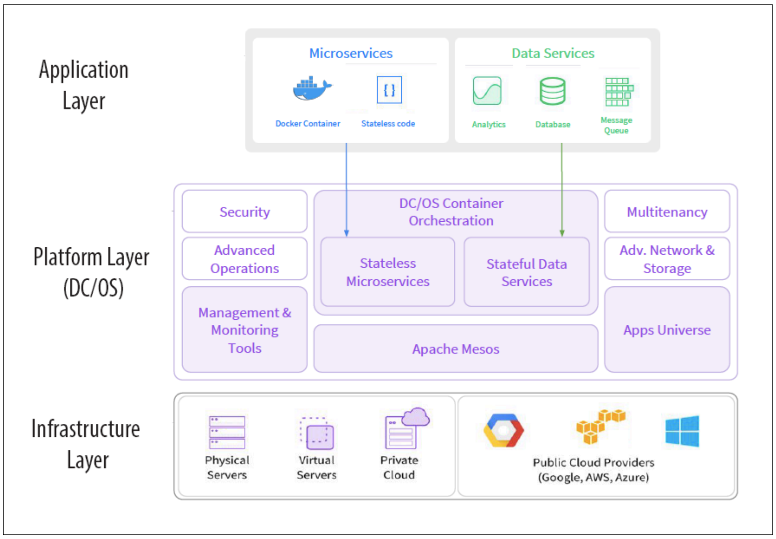


*Figure 4-1. The DC/OS Modern Enterprise Architecture (Source: Mesosphere)*

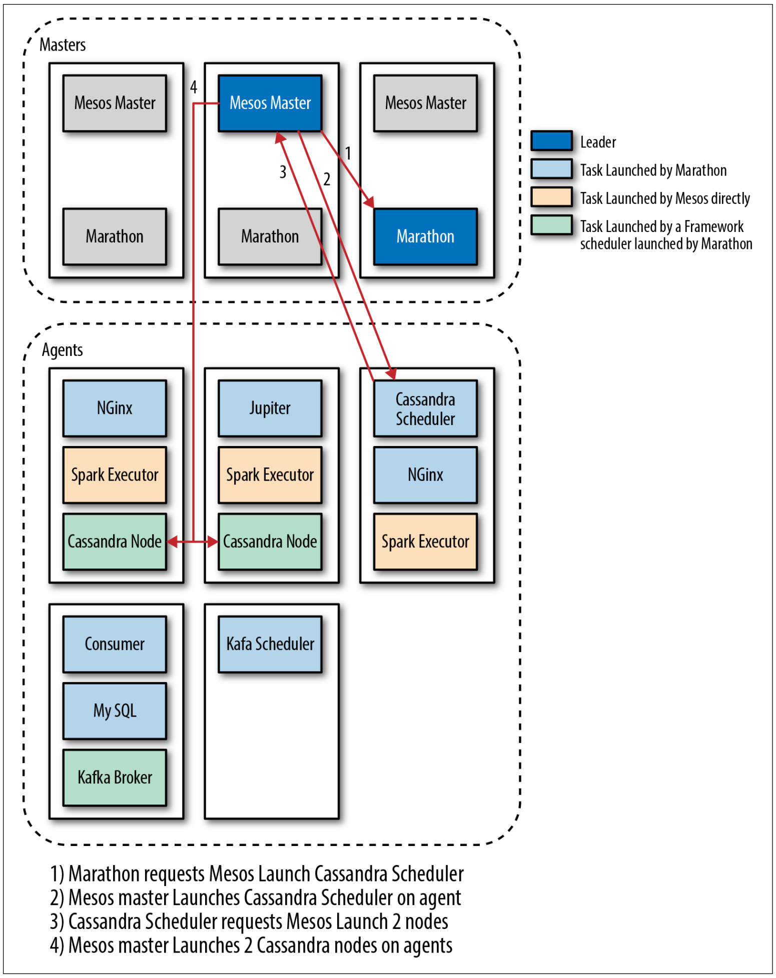
According to the principles of this architecture, we should write our services as stateless executables and use packages such as Cassandra to store application state. Programs written in this way are easily managed as Marathon apps.

## Marathon (for apps) and Metronome (for jobs)

The most common way of running software in DC/OS is as a Marathon *app* or a Metronome *job*. *Apps* are for services intended to be always running and available, such as web servers, CRUD services, and so on. *Jobs* are tasks that are run according to some time schedule (or one-off) and run to completion. Other than that distinction these two approaches are virtually identical.

Marathon (runs apps) and Metronome (runs jobs) are both DC/OS components that act as Mesos frameworks to place tasks on Mesos. Both add Docker support, health checks, failure handling, and change management appropriate to their use case. For example, if a Marathon app fails or the node that it is running on is lost, Marathon will automatically deploy a replacement task to Mesos. It is important to understand that the failure-handling behavior is not determined by Mesos but must be provided by the Mesos framework, in this case, Marathon.

Figure 4-2 shows how Marathon runs on masters in DC/OS and how Marathon runs apps on Mesos. In the figure, I use the Cassandra scheduler as an example of how a Marathon app can itself be a Mesos framework scheduler.



Masters
Mesos Master
Mesos Master
Mesos Master
Marathon
Marathon
Marathon
4
1
2
3
Leader
Task Launched by Marathon
Task Launched by Mesos directly
Task Launched by a Framework scheduler launched by Marathon
Agents
NGinx
Spark Executor
Cassandra Node
Jupiter
Spark Executor
Cassandra Node
Cassandra Scheduler
NGinx
Spark Executor
Consumer
My SQL
Kafka Broker
Kafa Scheduler
1) Marathon requests Mesos Launch Cassandra Scheduler
2) Mesos master Launches Cassandra Scheduler on agent
3) Cassandra Scheduler requests Mesos Launch 2 nodes
4) Mesos master Launches 2 Cassandra nodes on agents

*Figure 4-2. Some complex interactions between Marathon, apps and Mesos.*

## Containerization

Apps and jobs can execute code in a *Mesos sandbox* on an agent node or they can run a Docker container. Docker images allow developers to fully define the environment in which an application is executed and build an immutable container image that can run anywhere. In addition to the portability and reproducibility benefits of using Docker images, there are two important operational advantages:

- Containerization means that nodes do not need to have any application dependencies installed (e.g., a specific version of openssl).
- Container isolation means that one node can run multiple applications that have incompatible dependencies (e.g., two applications that depend on incompatible versions of openssl)

After jobs finish, their sandboxes and container environments remain on the node (which can be very useful for debugging a failing app) and are eventually cleaned up by a garbage collection process.

## Alternative Platform Layers

Marathon and Metronome are both Mesos frameworks, which provide a Platform as a Service (PaaS) layer[1] onto which containerized applications can be deployed. There are alternatives that provide similar PaaS functionality that can be run on DC/OS. For example:

- Kubernetes[2]
- Docker Swarm[3]

These alternatives are not core DC/OS components, and because Marathon is the most mature and well-supported mechanism for running containerized applications on DC/OS, that is what we will cover in this report.

## Marathon Pods

A pod in Marathon links multiple apps together into a group of tasks that are executed together on a single agent. Pods allow interdependent tasks to be deployed together and to share certain resources. Tasks within a pod share a network interface (either the host network interface or a virtual network interface, depending on con-

1 *https://mesosphere.com/blog/2015/07/10/why-your-private-cloud-could-look-a-lot-like-paas/*

2 *https://kubernetes.io/docs/getting-started-guides/dcos/*

3 *https://mesosphere.com/blog/2015/05/20/hyperscaling-docker-swarm-with-mesos-mesosphere-hackweek/*

figuration) and can communicate with one another on the localhost interface.

**NOTE** Pods and groups are easily confused. In Marathon, a group is a set of services (apps and/or pods) within a hierarchical directory path structure—Marathon groups exist only for namespacing and organization; they have no affect on where or how tasks are run.

You can find more information on pods (which are still experimental as of this writing) at *https://dcos.io/docs/1.9/usage/pods/*.

Example 4-1 shows an example pod with three containers.

*Example 4-1. An example pod with three containers (based on https://dcos.io/docs/1.9/usage/pods/examples/)*

```
{
  "id": "/pod-with-multiple-containers",
  "labels": {},
  "version": "2017-01-03T18:21:19.31Z",
  "environment": {},
  "containers": [
    {
      "name": "sleep1",
      "exec": {
         "command": {
         "shell": "sleep 1000"
        }
    },
    "resources": {
       "cpus": 0.01,
       "mem": 32,
       "disk": 0,
       "gpus": 0
      }
    },
    {
      "name": "sleep2",
      "exec": {
        "command": {
           "shell": "sleep 1000"
         }
    },
    "resources": {
       "cpus": 0.01,
       "mem": 32
      }
    },
    {
    "name": "sleep3",
    "exec": {
       "command": {
       "shell": "sleep 1000"
       }
    },
    "resources": {
       "cpus": 0.01,
       "mem": 32
    }
  }
  ],
  "secrets": {},
  "volumes": [],
  "networks": [
    {
    "mode": "host",
    "labels": {}
    }
  ],
  "scaling": {
    "kind": "fixed",
    "instances": 10,
    "maxInstances": null
  },
  "scheduling": {
    "backoff": {
    "backoff": 1,
    "backoffFactor": 1.15,
    "maxLaunchDelay": 3600
    },
    "upgrade": {
    "minimumHealthCapacity": 1,
    "maximumOverCapacity": 1
    },
    "placement": {
    "constraints": [],
    "acceptedResourceRoles": []
    },
    "killSelection": "YoungestFirst",
    "unreachableStrategy": {
    "inactiveAfterSeconds": 900,
    "expungeAfterSeconds": 604800
    }
  },
  "executorResources": {
    "cpus": 0.1,
    "mem": 32,
    "disk": 10
  }
}
```

## Failure Handling in Marathon

With Marathon, you can configure health checks for all apps. If no health checks are specified, Marathon will use the Mesos state of the task as the health check. Marathon health checks are versatile: you can use a number of protocols or execute commands in the task sandbox/container. Here are allowed health checks:

- HTTP
- HTTPS
- TCP
- COMMAND
- MESOS_HTTP
- MESOS_HTTPS

The first two methods (HTTP and HTTPS) make calls from the Marathon leader; the last two (MESOS_HTTP and MESOS_HTTPS) are executed on the Mesos task host. COMMAND is used to execute commands in the task sandbox/container (as appropriate). You can find details of health check options in the Marathon documentation.

**NOTE**

COMMAND and MESOS_ checks are pushed down to Mesos and are implemented as Mesos health checks. Why is that important? Because Marathon health checks are not visible to some components that use the Mesos API. For example, the Minuteman load balancer is able to use only Mesos health checks (and is not aware of Marathon health checks) to determine healthy task instances to which to route requests.

If a task fails health checks, Marathon will terminate the failing task and replace it with a new task. Marathon aims to keep the configured number of task instances in a healthy state. To achieve this, it will automatically start tasks to replace tasks that quit or crash until the desired number of healthy tasks is achieved. This can lead to *crashlooping*, wherein tasks are stuck in a loop constantly failing and being replaced. A slow crashloop (e.g., one in which a task always fails 10 minutes after it is started) is not obvious to the administrator or users, because a lot of the time Marathon will show that the desired number of instances are running and healthy.

If Marathon is unable to communicate with a node, it will decide that it has been lost. Lost means that the node is not communicating with Marathon. In this situation, Marathon cannot determine if the node has been shut down temporarily, shut down permanently, or if it is still running but there is a communication problem (e.g., with the network). To avoid overwhelming nodes by reallocating large numbers of tasks in case of a communications failure (such as a network partition) or temporary shutdown (such as a rolling restart of nodes), Marathon limits the rate at which tasks can be rescheduled for lost nodes.

I know of a number of people who have run into problems because they were not aware of this behavior and they deliberately shut down multiple active nodes (e.g., for maintenance) expecting Marathon to reassign the running tasks automatically, only to find that this happens relatively slowly. It is important to be aware that in case of a large-scale failure it will take Marathon some time (potentially hours) to recover the desired cluster state. To deliver reliability in face of potential multiple-agent failures, it is necessary to have sufficient instances running that you are not relying on Marathon to reassign tasks in a multiagent failure situation.

If a task run by Marathon is unable to communicate with the Marathon scheduler, it will continue to run normally. This is the default behavior of Mesos tasks, unless they are specifically written to terminate when they lose connection to their scheduler, which would not be recommended.

## High Availability and Change Management in Marathon

Marathon uses the information that it gets from Mesos along with its own health checks to provide change handling for apps. Marathon aims to ensure that *at least* the configured number of app instances are healthy at all times. To achieve that, when an app configuration change is made, Marathon will keep all existing instances of the app running and start new instances of the app (with the new configuration). Marathon begins removing the old instances only after new instances have started up and passed health checks so that the total number of healthy instances is maintained. Provided that services and clients are well written, this means that configuration changes, including software version changes, can be carried out without application downtime just by changing the Marathon configuration (for more complex configuration changes, see the section "Deployment" in Chapter 6).

## Other Marathon Features

Marathon is a very sophisticated application and has many advanced features and configuration options that cannot be covered in the space available in this report. Nonetheless, here are some Marathon features that you might find of interest:

- Metrics
- Events API
- Constraints
- IP per task
- Dependency handling

If you want to find out about these features or any other information about Marathon, the Marathon GitHub repository is a great resource as well as the Mesosphere documentation for Marathon.

CHAPTER 5

# Writing Applications to Run on DC/OS

In this chapter, we look at important things that you need to know before you begin to design and write applications to run on DC/OS. In particular, I will address two questions:

- How do your applications communicate with one another and with services provided by packages? (Service Discovery)
- Where and how is data stored in DC/OS? (Persistence)

The answers to these two questions allow you to create applications from interdependent services running on DC/OS. In the final part of this chapter, I describe the structure of an example system consisting of multiple applications and services running on DC/OS.

## Service Discovery in DC/OS

Service discovery is a mechanism that applications can use to dynamically discover the location and address of services that they want to communicate with at runtime. Service discovery replaces static configuration, which is a typical part of many multicomponent systems. In this situation, we are principally concerned with service discovery for applications running within DC/OS.

## What Is Service Discovery?

An example of a service discovery mechanism that most people are familiar with is DNS (specifically DNS A Records). DNS is a mechanism for looking up the specific IP address of a machine that provides a particular service for accomplishing the following:

- Identifying the specific address of the server, because the service might change over time.
- It is useful to reference the desired service (i.e., the website) by a common name that's independent of address of the service; for example, a URL like *www.mywebsite.com*.

These same requirements apply to applications that provide a service in a DC/OS cluster. Generally, it's not possible to know in advance which node a task in DC/OS will run on. The specific node determines the IP address (and potentially also the port) at which a service can be found. The location of tasks for a service can also change over time in DC/OS. If another application wants to reliably communicate with a service running in DC/OS it cannot use a statically configured IP address. So, there must be a dynamic way to determine the address of each service in DC/OS if applications are to be able to reliably communicate!

Clearly, service discovery is a necessity in DC/OS but it provides developers and system administrators with great benefits compared with methods that rely on static configuration:

- There is no need to maintain and distribute static configurations.
- Systems with dynamic service discovery are more capable of self-healing in failure scenarios.
- The software development process is simpler because software developers do not need to consider or write configuration handling code.

There are two principal methods of service discovery in DC/OS:

- VIPs—provided by Minuteman Mesos DNS
- Mesos DNS docs

I will describe each of these in the next two sections.

## Service Discovery Using VIPs

Virtual IPs (VIPs) can be used to map the IP addresses and ports of a service to a single Virtual IP. DC/OS VIPs are name-based rather than numerical like regular IP addresses.[1]

A named VIP contains three components:

- Private virtual IP address
- Port (a port on which the service is available)
- Service name

Marathon app VIPs have the following form:

```
<service-name>.marathon.l4lb.thisdcos.directory:<port>
```

An example marathon VIP for an app with ID "my-service" using port 5555 would look like this:

```
my-service.marathon.l4lb.thisdcos.directory:5555
```

For Kafka brokers, the VIP looks like this:

```
broker.kafka.l4lb.thisdcos.directory:9092
```

VIPs are resolved by Minuteman, a distributed Layer 4 load balancer (the `l4lb` in the preceding VIP example is an acronym for "Layer 4 load balancer"). Minuteman is a core component of DC/OS. Minuteman runs on every DC/OS node (as a systemd unit) and constantly updates the node's IP route tables to match the current state of the cluster.

As Example 5-1 demonstrates, Marathon supports setting VIPs for apps in the app definition, but if this is not set, VIPs are not automatically generated (unlike Mesos DNS records).

**NOTE** DC/OS packages also can register VIPs (e.g., Kafka).

1 *https://docs.mesosphere.com/1.8/usage/service-discovery/load-balancing-vips/virtual-ip-addresses/*

*Example 5-1. An example Marathon APP definition including a named VIP*

```
{
  "id": "/my-service",
  "cmd": "sleep 10",
  "cpus": 1,
  "portDefinitions": [
    {
    "protocol": "tcp",
    "port": 5555,
    "labels": {
          "VIP_0": "/my-service:5555"
    },
    "name": "my-vip"
    }
  ]
}
```

A Minuteman VIP has the following properties:

- Provides load balancing (Layer 4, so *TCP only*)
- Includes mesos health checks support (DC/OS 1.9)
- Still in preview (in DC/OS 1.8)
- Allows drop-in replacement for addresses in legacy systems
- Does not check health or readiness at application level (if your app retries on failure this is probably tolerable)
- The expected time to detect a task status change in Mesos and act upon it can be as much as 10 seconds. This means that requests could continue being routed to a failed node during that time.
- Will not work for applications running outside the DC/OS cluster. VIPs only work on nodes where Minuteman is already running

Minuteman VIPs are a great scalable solution for service discovery within a cluster. However, Minuteman's knowledge of task state is based on Mesos status information, which has a slow resolution time, and Minuteman does not provide any Layer 7 (connection-aware) load balancing or routing. If you need zero downtime or connection-aware behavior, you aren't going to be satisfied by Minuteman VIPs, and you should investigate an alternative approach such as using Marathon-LB.

For most internal services the operational benefits of built-in load balancing are significant and the limitations are not a problem for well-written applications. Minuteman provides built-in load balancing for applications and services in a DC/OS cluster that is reliable, fault tolerant, and self-healing. The configuration required is minimal and VIPs are easy to use in existing code. Following are common operational tasks that VIPs provide automatically:

- Upgrade of an existing dependent service (with compatible API). In general, services implemented as Marathon apps can be upgraded with no downtime if VIPs are used.
- Failover of requests to an alternate instance if an instance fails.

For a basic example of how you can use a VIP in application code, see the Kafka application example code in the section "Package Examples" in Chapter 3.

## Service Discovery with Mesos DNS

Every DC/OS master runs Mesos DNS,[2] which is a Mesos-aware DNS Server implementation. All Mesos agents are automatically configured to use the Mesos masters for DNS. Figure 5-1 shows the Mesos DNS lookup process.

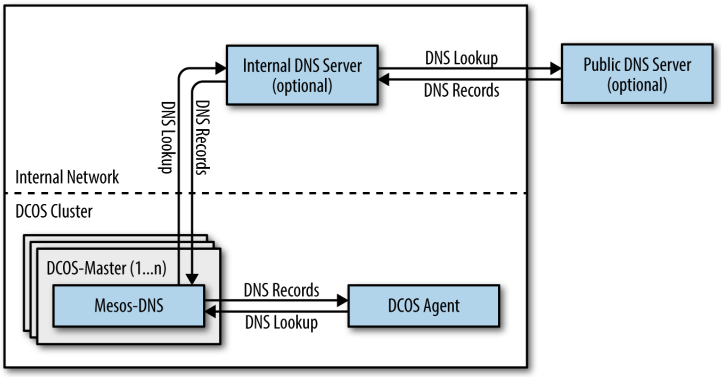


*Figure 5-1. Mesos DNS diagram*

2 *https://dcos.io/docs/1.8/usage/service-discovery/mesos-dns/*

Mesos DNS populates its records by polling the masters for the Mesos state every 30 seconds, and based on this, every task running in Mesos gets predictable DNS record entries. You can use this, for example, to address the current leader using `leader.mesos`. Every Marathon app gets the following DNS record entries in Mesos DNS, which can be used for service discovery:

- A record entry at `<`*`app id`*`>.marathon.mesos`, which returns the IP addresses of all app instances
- SRV record entry at `<`*`app id`*`>.<tcp|udp>.marathon.mesos`, which returns the IP addresses and ports of all running app instances

Beyond basic round-robin, Mesos DNS does not provide any load-balancing functionality. Mesos DNS also suffers from the following limitations:

- Using SRV records requires extra Mesos DNS–specific application logic to make SRV requests, handle the results, decide which of the advertised instances to use, and convert it to a regular address for a client (Example 5-2).
- Using A records only allows IP lookup and requires that the service ports are fixed and set either by hard-coding or by configuration.

*Example 5-2. Client code for generating a service URL from a DNS SRV request*

```
function fetchMarathonServiceUrl(serviceName){
    var dnsUrl = ['_' + serviceName, '_tcp.mara
thon.mesos'].join('.');
    return resolveSrv(dnsUrl).then(function(addresses){
         return 'http://' + addresses[0].name + ':' +
addresses[0].port;
    })
}
```

SRV records are generally the best for service discovery with Mesos DNS because they provide the ports as well as the IP addresses of running instances. This example shows the records returned when running two instances of `my-app` in Marathon:

```
$ dig SRV _my-app._tcp.marathon.mesos
; <<>> DiG 9.10.2-P2 <<>> SRV _my-app._tcp.marathon.mesos
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 46949
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDI-
TIONAL: 2
;; QUESTION SECTION:
;_my-app._tcp.marathon.mesos. IN SRV
;; ANSWER SECTION:
_my-app._tcp.marathon.mesos. 60 IN SRV  0 0 14266 my-app-hycpw-
s0.marathon.mesos.
_my-app._tcp.marathon.mesos. 60 IN SRV  0 0 18865 my-app-mwk7y-
s17.marathon.mesos.
;; ADDITIONAL SECTION:
my-app-mwk7y-s17.marathon.mesos. 60 IN  A 10.0.2.49
my-app-hycpw-s0.marathon.mesos. 60 IN A 10.0.2.37
```

This example shows that there are two instances of `my-app` running at 10.0.2.37:14266 and 10.0.2.49:18865. If you are using SRV records for service discovery, it is up to your client implementation to determine which of the available instances will be used for requests (Mesos DNS randomly orders the responses).

Because Mesos DNS is also using the Mesos status for tasks, it suffers from some of the same limitations as VIPs:

- Not aware of Marathon health or readiness checks (unless COMMAND, MESOS_TCP, or MESOS_HTTP checks are used).
- 30-second poll times on Mesos status mean responses to changes in cluster state are slow.

MESOS DNS has some additional limitations compared with VIPs:

- Mesos DNS runs only on master instances, thus it does not scale as well as Minuteman because clients have to make frequent requests to masters to get the latest DNS records.
- Some client libraries cache DNS A Record lookups or do not failover to an alternate IP when multiple A Records are provided but one is failing. Either of those can affect a client's ability to recover when tasks fail.

One difference between Mesos DNS and Minuteman VIPs is that Mesos DNS allows you to discover the addresses of all running instances of a service. Knowing all addresses is useful for systems that implement mesh or overlay networks or have their own load-balancing behavior.

As an alternative to using DNS queries to discover information stored in Mesos DNS, it also is possible to query the Mesos DNS server with HTTP requests. In addition to service lookup, which matches the functionality already described for Mesos DNS, the Mesos DNS HTTP API supports enumerating all the tasks and services available in the cluster and querying the Mesos DNS configuration. You can find more information on this in the Mesos DNS documentation on GitHub.

## Other Service Discovery Options

As well as the previously defined service discovery techniques, there are a range of different ways of performing service discovery, and each method has advantages and limitations. The following list provides a good selection of service discovery methods:

- Mesos API
- Marathon-LB
- AdminRouter
- Package/Framework APIs
- DC/OS CLI (uses admin router and/or package APIs)
- Third-party options (provided by packages)
  — Linkerd
  — Vamp

To deploy software applications on DC/OS requires reliable service discovery. It is not necessarily sufficient for DC/OS applications to look up the addresses of their dependencies on initialization. DC/OS applications (or the routing solution they are using) must be tolerant to changes in the location of required services during runtime. Whatever service discovery strategy is being employed the implications will require careful consideration by software engineers. If your application has very high performance or reliability requirements, extensive testing might be required. For best performance, it will probably be necessary to put in place and test DC/OS–specific code for service discovery and interservice communication in your applications.

## Managing Persistent State in DC/OS

Providing durable, available storage and persistent state is a significant challenge in distributed systems. Many design and architecture patterns advocate the use of stateless services; however, there are virtually no business applications that can be delivered by a totally stateless system. In modern systems, information can be stored on a variety of media including RAM, magnetic and solid-state drives, tape storage, and so on (paper printouts, microfilm, and so forth also can be considered as storage media for some applications).

There are trivial and fundamental challenges posed when delivering persistent storage in a distributed software system. Here are some of the trivial challenges:

*Discovery*
: Because instances of my program might be running on different nodes from those where data is stored, how does my application find a particular piece of persisted information? (That is, looking for data on the local file system is unlikely to work.)

*Locality*
: If data is not present on the node executing a program, I/O performance might be worse than that associated with local disk storage (if we are performing lots of non-local I/O, the network capacity of the infrastructure to support our application must be considered)

Here are the fundamental challenges:

- How to deliver long-term *durability*?
- How to provide short-term *reliability*?

Reliability can be defined as the probability that information written to a storage system will be able to be retrieved in a timely fashion after a given period of time has elapsed: "99.99 percent probability that data written now will be able to be retrieved within 1 week of a request up to 7 years from now"

Durability is concerned with ensuring that data will not be irrevocably lost in a range of failure or disaster scenarios, such as the following:

- Loss/failure of a disk drive (e.g., mechanical failure)

- Loss of all data in a datacenter (e.g., datacenter fire)
- Loss of data stored in a particular software system (e.g., resulting from a computer virus or hack)

Long-term durability is typically provided by ensuring that data is copied to multiple physical media in distinct, geographically separate locations. Durability concerns are heavily driven by regulatory and business-specific requirements. Meeting long-term durability requirements will typically require specific infrastructure or application-level solutions, including the following:

- Synchronization of data between multiple DC/OS clusters
- Application-specific backup, potentially to offsite, cloud, or other alternate storage (e.g., backup of Cassandra files to AWS S3; backup of SQL files to tape storage)
- Disk-level backup, potentially to offsite, cloud, or other alternate storage
- Storage-Area Network (SAN)–based backup or duplication of data

Short-term reliability is concerned with ensuring that applications will be able to consistently read and write data to persistent storage under expected operating conditions. Unreliable behaviors can be either of the following:

- A request using out-of-date (not the latest written) information (consistency failure)
- A request cannot be satisfied (availability failure)

Technical architects should be aware that 100 percent reliability is almost always the wrong reliability target. In the case of distributed systems, CAP Theorem tells us that there must be trade-offs between consistency, availability, and fault tolerance. Different systems, and system configurations allow designers to make trade-offs between these properties to suit particular applications.

Storage is presented to application developers as one of two abstractions—filesystem or database (I'm considering object storage such as AWS S3 to fall under the filesystem category)—the general distinction between these is that a database-style abstraction allows the application developer to query a data store to read and write data according to some application-specific schema. A filesystem-style abstraction allows the application developer to read and write binary data at a specific address or location.

By default, Mesos tasks lose internal state (such as the sandbox/container filesystem) when they terminate, which does not provide reliable persistence. To provide reliable persistent state, it is necessary to store state somewhere outside of Mesos tasks.

When using DC/OS, persistence can be provided by Mesos Persistent Volumes, by user-configured external storage, by a package that provides storage, or by an independent service. These solutions are independent, can provide both database and filesystem abstractions, and, in some cases, can be combined. In the next sections, I will cover in more detail the methods by which we can persist state in DC/OS.

## Local Persistent Volumes

Local persistent volumes are a built-in feature of DC/OS[3] that allocates requested amount of local disk storage on an agent node when a task is first launched. The persistent volume is maintained on the node and tracked by Mesos, even if the task that it was originally created for finishes or if the node is restarted. Framework can then require that subsequently, when launching tasks, the persistent volume that they need must be available and it will be mounted on any subsequent task that requires it.

After a persistent volume has been created on a node, all subsequent tasks that require it can run only on the node with the persistent volume, which reduces the flexibility of the cluster to handle failures.

Persistent volumes are easy to use with Marathon apps, as shown in Example 5-3.

*Example 5-3. Marathon app definition to run POSTGRES with a persistent volume (taken from https:// dcos.io/docs/1.9/usage/storage/persistent-volume/)*

```
{
  "id": "/postgres",
```

3 *https://dcos.io/docs/1.9/usage/storage/persistent-volume/*

```
  "cpus": 1,
  "instances": 1,
  "mem": 512,
  "container": {
    "type": "DOCKER",
    "volumes": [
    {
          "containerPath": "pgdata",
          "mode": "RW",
          "persistent": {
          "size": 100
          }
    }
    ],
    "docker": {
    "image": "postgres:latest",
    "network": "BRIDGE",
    "portMappings": [
          {
          "containerPort": 5432,
          "hostPort": 0,
          "protocol": "tcp",
          "name": "postgres"
          }
    ]
    }
  },
  "env": {
    "POSTGRES_PASSWORD": "password",
    "PGDATA": "/mnt/mesos/sandbox/pgdata"
  },
  "residency": {
    "taskLostBehavior": "WAIT_FOREVER"
  },
  "upgradeStrategy": {
    "maximumOverCapacity": 0,
    "minimumHealthCapacity": 0
  }
}
```

If the node where a local persistent volume resides fails or does not have enough other resources for a task that requires the persistent volume (e.g., all its CPU is occupied by other tasks), it will not be possible for a task requiring the persistent volume to start. Marathon supports only running a single instance of apps using persistent volumes; it does not support high-availability configuration changes for apps using persistent volumes. There are a number of other limitations, which you can see detailed in the DC/OS documentation. Because of these limitations local persistent volumes alone cannot provide highly available, reliable storage for DC/OS applications.

## External Persistent Volumes

External persistent volumes are a built-in feature of DC/OS.[4] However, they are currently experimental. External persistent volumes enable apps to be more fault-tolerant than local persistent volumes because they are not pinned to a specific node. If a node fails, tasks can be rescheduled on another node and will automatically have access to the associated data on the external persistent volume.

Support for external persistent volumes in DC/OS is provided by REX-Ray, which can be used to mount external block storage into Docker or Mesos containers at runtime. REX-Ray supports Amazon Elastic Block Store, OpenStack Cinder, EMC Isilon, EMC ScaleIO, EMC XtremIO, EMC VMAX, and Google Compute Engine storage, so it is suitable for public cloud and on-premises deployments. You can get more information on REX-Ray from Mesosphere and EMC.

NOTE

Built-in support for REX-Ray in DC/OS must be enabled when DC/OS is installed. As of this writing, this support is not enabled by default.

Currently, Marathon APPs using external volumes support only a single instance and do not support high-availability configuration changes. There are a number of other limitations detailed in the DC/OS documentation.

External persistent volumes can provide highly available, reliable storage for DC/OS applications if the underlying storage is highly available and reliable. To deliver high availability with external persistent volumes, you must write your application as a Mesos framework because it is not possible to run a Marathon app using external persistent volumes in high-availability mode. Example 5-4 shows a Marathon app configuration that uses external persistent volumes.

4 *https://dcos.io/docs/1.9/usage/storage/external-storage/*

*Example 5-4. Marathon app definition using external persistent volumes*

```
{
  "id": "/test-external-volumes",
  "instances": 1,
  "cpus": 1,
  "mem": 1024,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "my-repo/my-image"
    },
    "volumes": [
    {
       "containerPath": "/data/my-persisted-data",
       "external": {
       "name": "my-external-vol",
       "provider": "dvdi",
       "options": { "dvdi/driver": "rexray" }
     },
       "mode": "RW"
    }
    ]
  },
  "upgradeStrategy": {
    "minimumHealthCapacity": 0,
    "maximumOverCapacity": 0
  }
}
```

## Other External Storage

External storage can be provided by mounting nonlocal storage (if available) as a filesystem at an OS level so that storage can be used via the usual OS filesystem behaviors. Here are two nonlocal storage solutions:

- Network Attached Storage (NAS) (providing file-level storage over a general-purpose TCP/IP network)
- Storage Area Network (SAN) (providing block-level storage over a dedicated storage network)

Using external storage in this way without built-in Mesos or DC/OS support requires nodes to have appropriate configuration at the OS level independent of any DCOS setup or for applications to bootstrap the OS with the storage at runtime.

The key requirement of these solutions is that they should allow data to be consistently accessible from all nodes in the cluster with similar performance so that storage applications can access their persisted data while remaining agnostic as to which node they are running on.

General external storage solutions face coordination problems. For example, what happens if two tasks concurrently change the same data. Handling coordination might require extra work from application developers or in setup and configuration of storage. External storage can potentially provide highly available, reliable storage for DC/OS applications but can require significant work to implement correctly.

## Storage Packages

Packages that provide reliable storage on DC/OS duplicate data across multiple nodes to provide redundancy. With sufficient redundancy, you can expect that storage will be reliable even if there are some individual task restarts, failures, or losses. Storage packages expose their redundant storage to applications as a service, typically using a filesystem or database abstraction, as described earlier in this chapter.

In principle, a service with a lot of redundancy could provide reliable storage in memory alone if the DC/OS cluster could be expected to always have sufficient available capacity (you can try this with Spark); however, it is more common for storage packages to make use of multiple persistent volumes (external or local) and provide a mechanism for administrators to notify it when a persistent volume is irretrievably lost so that it can be replaced with a new one.

In an advanced setup, some storage packages (such as Cassandra) can be configured to store redundant copies of data in separate, distinct racks or even datacenters to minimize the risk of a single failure removing multiple replicas simultaneously. This can be handled automatically if the rack and dc attributes of a node are set as Mesos attributes.

The following are some examples of DC/OS packages that provide reliable storage:

- Cassandra/DSE (database)
- HDFS (filesystem)

- Kafka/Confluent-Kafka

NOTE

Kafka stores data reliably if correctly configured but only for a time window—typically, Kafka queues roll off after 1 to 10 days, depending on configuration.

- ArrangoDB (database)

Storage packages should provide a mechanism for replacing a lost persistent volume to ensure that desired replication is maintained. Replacing a lost node requires transferring data from the other nodes that have copies of the data that was on the lost node to a replacement node. Because it's an expensive operation to transfer large amounts of data between nodes, it is not normally done automatically by storage packages, because they cannot determine if a node is only down temporarily (e.g., for maintenance) or lost permanently. What this means in practice is that the mechanism for replacing a lost node usually involves human intervention to instruct the package that a node is not going to come back. This behavior means that storage packages are not self-healing; thus, users should ensure that they have appropriate alerting mechanisms and other procedures in place to handle failure scenarios that might affect storage packages.

NOTE

The operational reliability and behavior of storage packages that use persistent volumes will vary if they are using external or local persistent volumes.

As of this writing, there are a number of packages that provide storage but do not have built-in redundancy and so cannot tolerate node failure. This includes the MySQL and PostgreSQL packages, which run only a single database instance on a DC/OS node without any backup or duplication.

## Publishing Applications and Services

Most applications and services will be of little use to an organization if they are accessible only from within a DC/OS cluster. To be useful, it is necessary to publish some applications and services so that they can be accessed over the public internet or a private intranet. In this section, I will explain how you can access services from outside of a DC/OS cluster.

## High Availability and Load-Balancing External Services

Published services are generally exposed by using URLs, which are resolved by DNS to one or more IP addresses. If your URL resolves to a single IP address, this is likely a Single Point of Failure (SPoF). This can be partially addressed by having multiple IP addresses associated with a single URL (DNS load balancing). However, because DNS records are persisted on both clients and DNS servers beyond your control (which might not respect short TTLs), it is desirable that the servers attached to public IP addresses are extremely reliable.

In DC/OS, tasks move unpredictably between nodes with different IP addresses, so it is not advisable to try to configure DNS records to use IP addresses of nodes managed by DC/OS. In this respect, DC/OS is not particularly unusual. The recommendation for high-availability URLs is to have in place dedicated load balancers/reverse proxies. These reverse proxies should be set up appropriately (often using specialized hardware) to provide high availability and DNS load balancing.

The typical DC/OS setup is to use dedicated reverse proxies to direct requests to multiple Marathon-LB instances running on a public subnet outside of the network DMZ. Marathon-LB can then automatically route requests into the DC/OS DMZ for any Marathon application. Marathon-LB has support for a range of Layer 7 load-balancing features and can also be set up to perform autoscaling of Marathon applications. Figure 5-2 illustrates the recommended arrangement for load balancing public services with DC/OS and Marathon-LB.

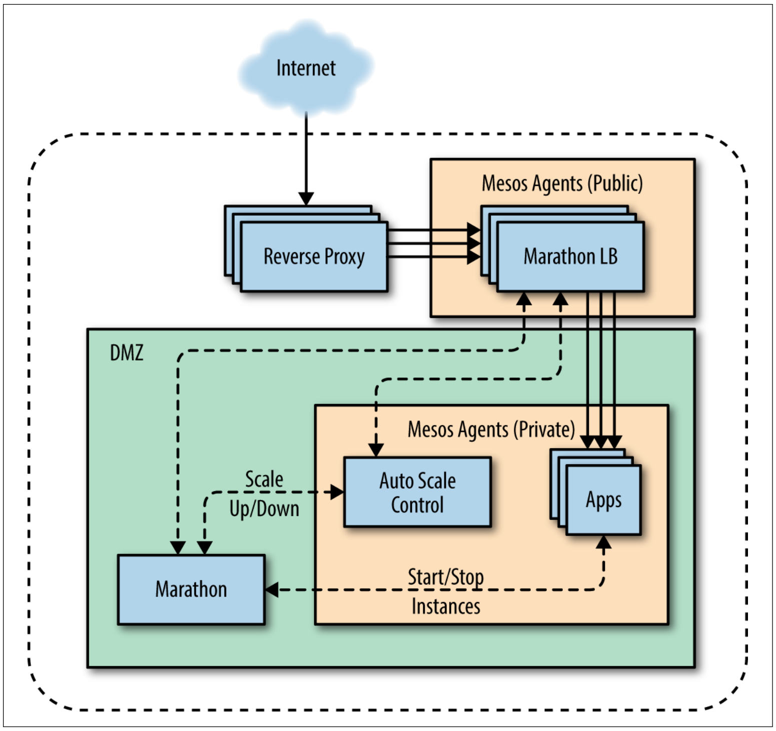


*Figure 5-2. Load balancing public services with DC/OS and Marathon-LB*

## Section Conclusion: Example Applications on DC/OS

Using all of the topics covered so far in this report, it is possible to describe how we could implement an example system on DC/OS. The demonstration application has the following requirements:

- Provide a public REST API for an existing mobile app and website
- Host HTML, CSS, and JavaScript for the website that uses the REST API
- Save the business state in a database
- Ensure that outbound notifications are sent in response to certain actions

- Allow analytics of the stored business state

We can use DC/OS packages for the business database, external load balancer, and to provide an event queue for the actions that will trigger outbound notifications. For example:

- Business database
  - Cassandra package
- Load balancer
  - External Marathon-LB running on statically-partitioned agents[5]
- Event queue
  - Kafka package

The remainder of the system can be implemented as a selection of Marathon apps. These will be a public website server, a public REST API server, an internal microservice that will implement our business logic for performing CRUD-type operations on the business state, a Kafka consumer responsible for sending outbound notifications in response to actions, and a Jupyter notebook that can be used to execute Spark jobs against the business data stored in Cassandra. With the exception of the Jupyter notebook (which will be used internally), all of the services are stateless, and we can run as many instances as we require to meet load and reliability requirements. Each of these apps would be implemented as follows:

- Website server
  - NGinX running in a Docker container and hosting static content.
  - Configured as vhost for site.example.com
  - DNS A Records directing our site.example.com to the load-balancer IPs
- Public REST API Server
  - Scala REST API server running in a Docker container
  - Configured as vhost for rest.example.com

5 *https://mesosphere.com/blog/2015/12/04/dcos-marathon-lb/*

    - Communicates with internal CRUD microservice
    - Event messages are sent to Kafka

- Internal microservice (RPC)
    - Go application running in a Docker container as a marathon app
    - Configured with:

      ```
      VIP: business-crud.marathon.l4lb.thisdcos.directory:
      9000
      ```

    - Exposes Remote Procedure Call (RPC) interface and applies business logic to requests
    - Result of CRUD requests are persisted to Cassandra

- Analytics interface
    - Python data analytics can be performed on a Jupyter (or Zepplin[6]) notebook running in a Docker container.
    - Executes Spark jobs that consume data stored in Cassandra

- Internal Kafka consumer
    - Node.js application running in a Docker container as a Marathon app
    - Does not need to be configured with a VIP
    - Probably does not need multiple instances (if an instance dies it will be replaced and this is async queue consumer so redundancy is probably not required)
    - Read data and perform updates using the internal RPC microservice

Figure 5-3 shows the interaction between these components.

6 *https://dcos.io/docs/1.9/usage/tutorials/spark/*

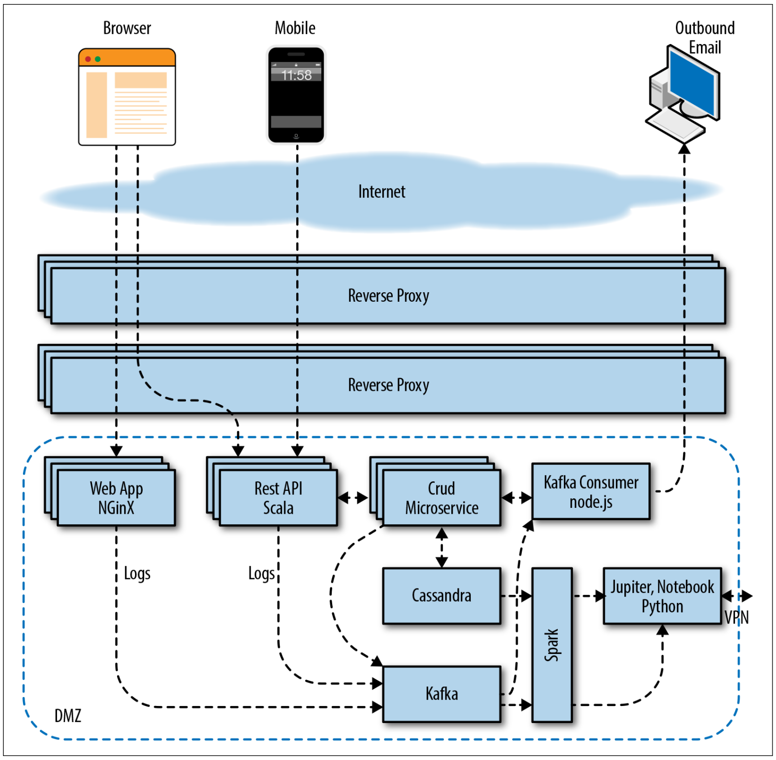


*Figure 5-3. Example system: application layer*

This example is very brief, but, hopefully, it illustrates how important components of a typical enterprise system could be implemented and deployed to work together on DC/OS.

In this chapter, we have looked at the things that you need to know in order to design and build applications to run on DC/OS. After applications are up and running, it is necessary to maintain them, monitor them, scale them and deal with any problems that arise. These tasks and more are the responsibility of the operations team, and I will discuss them in the next chapter.

CHAPTER 6

# Operating DC/OS in Production

In this chapter, I outline the things you need to know to run DC/OS and applications on DC/OS in production. I also highlight strategic topics that you should consider before you begin using DC/OS as a live/production system.

## Scaling

It is rare, even in large enterprises, for applications to be delivered fully formed to a large user base. Typically, applications begin with a small user base, either because it is first used by early adopters and takes some time to grow in popularity, or because the provider begins by providing a limited "alpha" or "beta" version to a restricted audience.

Demand planning and forecasting can be extremely difficult, particularly in the fast-moving world of consumer applications, so it is important that our architecture allows us to increase the capacity of our applications quickly and easily.

Increasing application capacity in DC/OS is straightforward for well-designed applications. Packages and Marathon apps can scale horizontally with simple configuration changes to increase the number of desired instances. However, this scaling is ultimately restricted by the total resources available in the DC/OS cluster. It is important that the DC/OS cluster allows us to scale the resources available to it.

## Scaling a DC/OS Cluster

Adding more agent nodes to DC/OS is an easy task. You can add new instances at any time, following the same procedure as initial cluster setup. After they have been set up with DC/OS services and configured to communicate with the existing masters, new agent's resources are available to DC/OS within minutes. In cloud environments such as Amazon Web Services (AWS) or Microsoft Azure, you can easily automate this process.

## Scaling Masters

Masters in DC/OS cannot be scaled horizontally because work is not distributed across them but concentrated on the leading master. Changing the number of master nodes in DC/OS is *not* straightforward, and it will not improve performance or operational capacity (multiple masters exist to provide redundancy). Generally, if there is a problem with the performance of master nodes, it is necessary to increasing the resources (RAM and CPU) of the master instances. You can do this by means of a rolling replacement of the master nodes—this is when masters are replaced by higher capacity instances one-by-one. When the leading master is removed for replacement, the cluster automatically fails-over the leadership to one of the other masters.

## Capacity Planning

Given a DC/OS cluster of a fixed workload, there are a range of situations in which application scaling might still occur. For these cases, it is advisable to ensure that a DC/OS cluster has sufficient spare capacity to meet operational needs. For a given workload, determining the necessary spare capacity is an important responsibility of system administrators. Following are a couple of situations in which application scaling commonly occurs:

*Deploying a version or configuration change*
: Most packages (Marathon included) will typically start new instances before terminating running instances following a configuration change. Packages will wait for new instances to be ready before they begin terminating old instances. Marathon uses this strategy, starting new tasks and only removing old ones when the new ones are healthy, until all instances have been replaced. This means that during a configuration change,

the cluster requires sufficient resources to run extra instances compared with normal operating conditions.

*Loss of nodes*
: There are a number of unavoidable situations that can cause loss of a node; for example, a hardware failure. Based on the size and set up of your cluster you should determine the number of simultaneous failures that the system needs to be able to support. Because DC/OS will reschedule tasks onto available nodes when some nodes fail, it is important that the remaining nodes have sufficient capacity to run applications from failed nodes.

## Dynamic Workloads

Adjusting the workload configuration in DC/OS is not difficult, and you can do it via the DC/OS GUI, CLI, or via package APIs (including the Marathon API). You can adjust resource allocation manually or automatically. DC/OS does not have any "autoscaling" abstractions or tools built in, but autoscaling behaviors are simple to implement using Marathon (or other package) APIs or by scripting the DC/OS command-line interface (CLI).

You can take advantage of dynamic configuration changes to achieve high utilization of computing resources; however, you need to be careful to ensure that applications are not starved of resources because of "noisy neighbors" on the cluster consuming all available resources. This approach is best used to shift resources between nonurgent background processing tasks and urgent service delivery tasks. For example, during periods of high demand, background tasks would be scaled down and service delivery tasks would scale up.

### Batch Processing

You can increase the speed of large batch-processing operations such as MapReduce using Hadoop or Spark by allocating more resources. In some situations, it might be desirable to allocate significant resources to a specific processing task; for example, a daily Extract, Transform, and Load (ETL) process or computationally intensive task like training a neural network.

## Multidatacenter DC/OS Configuration

Some organizations operate systems spread across multiple datacenter facilities to provide redundancy, geographical distribution, or simply because they cannot fit in a single datacenter. The recommendation from Mesosphere is that each datacenter should operate a single DC/OS cluster, and any cross-datacenter systems or services will require specific configuration. Some packages, notably Cassandra, have support for multi-DC/OS-cluster configuration.[1]

If datacenters have very good interconnection, it is possible to run a DC/OS cluster that spans multiple datacenters. This can be done in AWS with a cluster split across multiple Availability Zones (AZs) in a single region. In this situation, it is strongly recommended that the DC/OS masters are all in a single AZ and only the agents are spread across different AZs. This situation provides greater redundancy; however, it does increase the risk of a failure that can cause the following:

- Significant number of nodes no longer be able to communicate with the leading master
- Significant number of tasks no longer able to communicate with their framework scheduler

It is not recommended to attempt making changes to cluster configuration if there are agent nodes unable to communicate with the leading master, nor is it recommended to attempt making changes to a package configuration if there are tasks that are unable to communicate with their framework scheduler.

## Deployment

Change management is one of the key operational responsibilities. Typically, this means deploying new applications or making configuration changes to existing applications (including deploying new versions or rolling back an application version). The principal requirement is that systems should remain available and functioning during any change, and that if there are problems with the changes,

1 *https://docs.mesosphere.com/1.8/usage/service-guides/cassandra/multi-dc/*

the system should be quickly reverted to a previous stable and functioning state.

NOTE

In DC/OS, changes are made by updating the configuration of DC/OS to the desired configuration. DC/OS (or packages running on DC/OS) take responsibility for making changes to the state of the cluster to produce the desired state based on the configuration.

DC/OS keeps a durable record of the current configuration of the system but does not track previous configurations. Marathon does keep a record of current and previous app configurations and allows rollback to previous configurations.

You can easily deploy packages, Marathon apps, and metronome jobs on DC/OS through the DC/OS GUI; however, you should use this approach carefully. A manually applied configuration is difficult to manage in DC/OS—as it is with any other system. Users are recommended to make use of a Configuration as Code strategy to simultaneously automate and document DC/OS configuration. There are tools available in DC/OS that you can use to automate configuration. You can store both package and app configurations as JSON documents and you can deploy that configuration by using the DC/OS CLI or HTTP APIs. Moreover, if you use these methods, you can store the configuration and deployment scripts under version control. This provides for much better change management, you can use it to manage multiple environments (e.g., for development and testing) and as part of a business continuity strategy. These approaches are compatible with using an Infrastructure as Code approach and tooling to automate deployment and configuration of DC/OS.

There are a number of advanced operational techniques that you can automate on DC/OS. Here are some interesting examples:

- Canary analysis
- Shakedown
- DRAX ("Chaos Monkey" for DC/OS)

## Deploying a Marathon Application

You can deploy or update a Marathon application by sending the desired app definition to Marathon. Updates will be gradually rolled out using health checks, as described earlier in the section "High Availability and Change Management in Marathon" on page 50. You can roll back deployment of a totally new Marathon app by deleting the app by ID from the GUI or CLI. You can roll back an update by reverting to the previous configuration, which is possible via the marathon GUI or DC/OS CLI.

Marathon applications get their runtime setup from resource URIs and/or a docker image. These methods behave slightly differently.

### Resource URIs

Marathon uses the Mesos fetcher to fetch resources from URIs and place them in the APP sandbox. The Mesos fetcher can fetch resources using HTTP(S), HDFS, S3, (S)FTP, and the local filesystem, and it will automatically unpack common compressed file formats such as *.tgz* and *.zip* files. The Mesos fetcher uses a local cache, so you should take care if you're updating resources under the same address (in general, if you have updated resources, you should place them at a new address and change the URIs to point at the new address). This facility is used for a variety of purposes. Its principal use is to deploy application files and dependencies in to the sandbox ready for execution.

**NOTE**

As of Mesos v0.22, the fetcher code does not make downloaded files executable by default so you might need to use a command of the form `chmod u+x <executable-file> && ./<executable-file>`.

### Docker images

If a Docker image is specified, Marathon will, by default, fetch it from DockerHub. By default, Marathon will use the local Docker cache for images. You can configure applications to fetch images from private repositories on DockerHub by including a file containing dockerhub credentials in the resource URIs. If no command is specified, Marathon will run the Docker container's default command. If a command is specified, Marathon will run the specified command inside the container.

NOTE

To deploy Docker images from private Docker repositories (on DockerHub or elsewhere) requires providing credentials, which you can read about at *http://bit.ly/2mAoe8l*.

## Application Configuration

With Marathon, you can store application-specific configurations as a dictionary of values that will be passed to the application as environment variables at runtime (this is true whether or not Docker is being used).

NOTE

Changing the env value on the app is a configuration change and will trigger a rolling deployment of new tasks (with the new env values). An example of how this can be used is to set the VIP address of a service that a Marathon app requires.

```
{
 "id": "/my-app",
 "cmd": "...",
 "cpus": 1,
 "env": {
    "MY_SERVICE_VIP": "my-service-
v1.marathon.l4lb.thisdcos.directory:5555"
    }
}
```

By setting VIP configuration as part of the Marathon app definition, you can manage all VIP configuration in a single place and make coordinated changes to the VIP being used by clients and servers. This is better than the easy-to-occur situation in which the VIP is hard-coded in the client or in a resource file. You can use this setup to manage a switchover of a client app from using one version of a service to another.

## Deploying an Application Update with a Breaking Interface Change

Marathon has a built-in facility for deploying updated app versions without downtime and with health and readiness checks on app instances. Sometimes, when there are interdependencies between services, it is desirable to have a more controlled switchover. You can do this in a similar manner as a blue-green deployment pattern.[2] To achieve this, two instances of the service (old version and new version) can both run in Marathon with different IDs and VIPs, as demonstrated here:

```
{
  "id": "/my-service-v1",
  "cmd": "...",
  "cpus": 1,
  "portDefinitions": [
    {
    "protocol": "tcp",
    "port": 5555,
    "labels": {
         "VIP_0": "/my-service-v1:5555"
    },
    "name": "my-service-v1-vip"
    }
  ]
}
{
  "id": "/my-service-v2",
  "cmd": "...",
  "cpus": 1,
  "portDefinitions": [
    {
    "protocol": "tcp",
    "port": 5555,
    "labels": {
         "VIP_0": "/my-service-v2:5555"
    },
    "name": "my-service-v2-vip"
    }
  ]
}
```

After both apps are up and running, apps that depend on them can be switched over by updating the dependent app env configuration to change the VIP they use to reach the updated service:

```
{
 "id": "/my-app",
 "cmd": "...",
 "cpus": 1,
 "env": {
    "MY_SERVICE_VIP": "my-service-
v2.marathon.l4lb.thisdcos.directory:5555"
```

2 *https://mesosphere.github.io/marathon/docs/blue-green-deploy.html*

```
        }
    }
```

When all apps are moved over to the new service version, you can remove the old service version.

NOTE

If you want to perform a deployment like this for a service that is being load-balanced by Marathon LB, you should consult the Marathon LB documentation at *http://bit.ly/2nl4Xqh*.

For even more fine-grained control (e.g., as part of a 1 percent or 10 percent test), you can deploy a separate instance of the same Marathon app under a different app ID using the updated service while the main instance of the Marathon app continues to use the old version of the service. You can find the details for how to configure Marathon apps and Marathon-LB how to perform traffic splitting at *http://bit.ly/2mZqIs9*.

These procedures can be completely managed by the operations team with no need for input from software developers or changes to application software, provided that it has been written to retrieve VIP locations from environment variables.

## Deploying a DC/OS Package

DC/OS packages can each have package-specific configuration parameters, which are set when the package is installed. Sometimes, you can change these after installation, but not always. Packages are installed from the Universe repository.

### The DC/OS Package Universe

The DC/OS Universe is based on a public GitHub repository. The DC/OS Cosmos component is responsible for looking up information from the DC/OS Universe public repository. You can add a private repository to Cosmos in addition to the DC/OS Universe, or to replace it. Details of how to do this are in the DC/OS documentation.

## Package Configuration

The configuration options for each package are defined as a json-schema[3] document in the *config.json* of each package's universe entry. Every package's configuration template and default settings are included in the registry as JSON files. You can view the *config.json* for a package via the DC/OS CLI using the command `dcos package describe --config <package name>`, as shown in Example 6-1, or by looking at the latest entry for the package in the package registry on GitHub.

*Example 6-1. An example package config json-schema*

```
$ dcos package describe --config spark
# NOTE - the example output here has been deliberately abridged
and does not show all spark configuration options
{
  "properties": {
    "hdfs": {
    "description": "Spark-HDFS configuration properties",
    "properties": {
         "config-url": {
         "description": "Base URL that serves HDFS config files
(hdfs-site.xml, core-site.xml).  If not set, DCOS Spark will use
its default configuration and read from DCOS HDFS.",
         "type": "string"
         }
    },
    "type": "object"
    },
    "history-server": {
    "description": "History Server configuration properties",
    "properties": {
         "cleaner-enabled": {
         "default": false,
         "description": "Specifies whether the History Server
should periodically clean up event logs from storage.",
         "type": "boolean"
         },
         "cleaner-interval": {
         "default": "1d",
         "description": "How often the job history cleaner checks
for files to delete. Files are only deleted if they are older than
spark.history-server.cleaner-maxage.",
         "type": "string"
```

3 *http://json-schema.org/*

```
          }
     },
     "type": "object"
     },
     "service": {
     "description": "DCOS Spark configuration properties",
     "properties": {
          "cpus": {
          "default": 1,
          "description": "CPU shares",
          "minimum": 0.0,
          "type": "number"
          },
          "docker-image": {
          "default": "mesosphere/spark:1.0.7-2.1.0-hadoop-2.6",
          "description": "The docker image used to run the dis-
patcher, drivers, and executors.  If, for example, you need a
Spark built with a specific Hadoop version, set this variable to
one of the images here: https://hub.docker.com/r/mesosphere/spark/
tags/",
          "type": "string"
          },
          "log-level": {
          "default": "INFO",
          "description": "log4j log level for The Spark Dis-
patcher.  May be set to any valid log4j log level: https://
logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Level.html",
          "type": "string"
          },
          "mem": {
          "default": 1024.0,
          "description": "Memory (MB)",
          "minimum": 1024.0,
          "type": "number"
          },
          "name": {
          "default": "spark",
          "description": "The Spark Dispatcher will register with
Mesos with this as a framework name.  This service will be avail-
able at http://<dcos_url>/service/<name>/",
          "type": "string"
          },
          "spark-dist-uri": {
          "default": "https://downloads.mesosphere.com/spark/assets/
spark-2.1.0-bin-2.6.tgz",
          "description": "Spark distribution URI.  This is only
used to run spark-submit on your local machine.  The DCOS Spark
CLI will fetch this Spark distribution.  This variable has no
effect on the Spark distribution running the dispatcher, drivers,
nor executors.  If you want to customize the Spark build used in
the jobs themselves, configure service.docker-image.",
          "type": "string"
          }
      },
      "type": "object"
      }
    },
    "type": "object"
}
```

Installation configuration is provided as a JSON file using the DC/OS CLI the command is:

```
    dcos package install  --options=<configuration-file.json>
    <package-name>
```

Or if you are using the DC/OS GUI to install a package, you can upload your configuration file or you can enter configuration properties using a form. If you do not provide configuration options, default values will be used.

Following are valid configuration options for the preceding example Spark configuration schema:

```
    {
     "hdfs": {
      "cleaner-enabled": true
        },
        "service": {
             "log-level": "DEBUG"
             "name": "spark-test"
     }
    }
```

**NOTE** It is not necessary to set all configuration parameters; for those you do not set, defaults will be used.

When you install a package, the configuration options you provide are combined with the default values from the configuration schema, the values set in the package's *resources.json* file and the package's marathon app template file (the *marathon.json.mustache* file in the package entry) to create a Marathon app definition, which is deployed to DC/OS. If you want to view the Marathon app definition before deploying it, you can do so via the CLI by passing the `--render` option.

## Updating a Package

Changes to the configuration of an installed package are not necessarily possible. Packages that do support changes to the installation configuration do so in different ways, so it is necessary to check the appropriate method for each package individually.

The usual methods for changing an installed package configuration are either via a specific API on the package itself (which can be exposed in the DC/OS CLI) or by changing the package app definition in Marathon. When making changes to an installed package, it is recommended that you read the package documentation carefully and thoroughly tests the identical operation in a sandbox environment, if possible. It is up to package authors to implement configuration change handling, and DC/OS does not make any guarantees for the effect on a running package if you change its configuration after installation.

Updating a package to use a new version is something that must be handled on a case-by-case basis. Updating a package will require an understanding of the specific software/system run by the package. If a package does not store any state, it is recommended to follow the procedure for blue/green deployments; that is, deploy the package with the desired configuration change under a new ID and switch dependents to use that before removing the original. If a package stores state and it cannot be upgraded online, it might be necessary to perform a backup, remove the package, install the latest version of the package, and then restore the data from the backup.

# Security in DC/OS

Information security is an important and very technical topic. Fundamentally, DC/OS is a system of Linux nodes communicating over a network to provide software-defined services. This is not different from many large information systems, and the security considerations and controls for DC/OS are the same as those for any such system.

Much of the work of securing systems is done on the underlying Linux systems on which DC/OS is being run. It's not possible to go into this topic in detail in this report; however, there are some aspects of DC/OS configuration that are important to security, and in this section I will describe some of them.

DC/OS takes responsibility for controlling access to its management interfaces. DC/OS also enforces authentication on connections to the DC/OS cluster from outside the cluster. Authentication of users and access control is enforced by the admin router. Communication that does not go through the admin router is not subject to authentication or access control in DC/OS. The security features available in Mesosphere Enterprise DC/OS use the same mechanism but are more sophisticated.

Authentication and permissions management here refers to authentication required to access the DC/OS management interfaces or to access any DC/OS services from outside the DC/OS cluster.

**NOTE** Neither DC/OS nor Mesosphere Enterprise DC/OS provide for authentication or permissions management that you can use in software that you write to run on DC/OS.

## Secure Networking

DC/OS does not provide any tools for managing network security; however, it has a recommended network configuration[4] and works well with a zone-based network security design. In this design, nodes are each assigned a Mesos role corresponding to a specific "security zone." Figure 6-1 illustrates this design.

**NOTE** The AWS and Azure standard templates for DC/OS implement this network design.

4 *https://dcos.io/docs/1.8/administration/securing-your-cluster/*

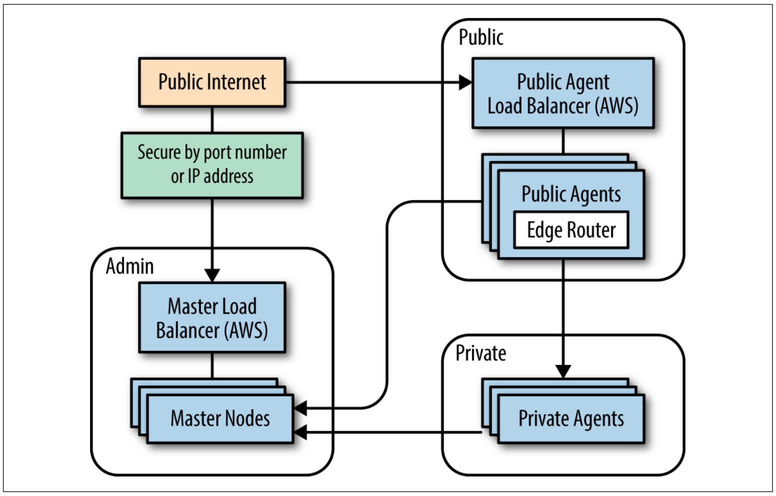

Public
Public Internet
Public Agent Load Balancer (AWS)
Secure by port number or IP address
Public Agents
Edge Router
Admin
Master Load Balancer (AWS)
Private
Master Nodes
Private Agents

*Figure 6-1. Standard DC/OS network configuration*

Using a Mesos role (canonically `slave_public`) to identify nodes outside of the network DMZ ensures that only tasks specifically configured with this role will be executed outside of the DMZ. The recommended system setup is to have a public zone and a DMZ consisting of a private and admin regions. Instances in the DMZ should not be addressable from the WAN/public internet. The public zone acts as a bridge between the DMZ and the public internet. The private zone is a subnet and typically contains the majority of nodes. The admin zone should be a separate subnet in the DMZ that contains only the master instances.

To perform DC/OS management, it is necessary to communicate with the master instances from DC/OS users' computers—depending on the setup and infrastructure being used it might be necessary for the master nodes to have public IP addresses so that they can be accessed for management tasks; however, this is not recommended, and you should try to avoid it.

Access to the master nodes should be as secure and as restricted as possible. If you are accessing the master nodes via the internet, you should enforce HTTPS on connections to the admin router (requires setting up certificates on master instances, as detailed in the DC/OS Documentation). One solution to access the admin zone via the public internet is to put in place a secure VPN and ensure that admin zone can be accessed only from outside the cluster via that VPN.

Other recommendations for securing master nodes are to restrict the ports that allow incoming connections and restrict access to the admin zone to connections originating from specific IP addresses. Further security can be obtained by placing admin zone behind a reverse proxy.[5]

## Administrative Access and Authentication

The default installation of Open Source DC/OS uses OAuth to authenticate users before allowing them access to the administer the DC/OS cluster (both via the CLI or the GUI). You can add administrators by email address. To gain access, administrators must authenticate themselves by using OAuth, provided they have an account at the same email address with one of Microsoft, Google, or GitHub. DC/OS Enterprise provides alternative options for authentication, including SSO.[6]

You can configure DC/OS to use alternative OAuth providers for administrator authentication,[7] or you can disable OAuth, in which case DC/OS will switch to using username/password basic authentication.

OAuth authentication for the DC/OS CLI is done using temporary tokens. If you try to use the DC/OS CLI without a valid token, you will be prompted to visit a website and authenticate to gain an OAuth token that can be entered in the DC/OS CLI. These tokens last only a couple of days before you need to replace them. Token expiration can cause problems for administrators who are trying to automate tasks using the DC/OS CLI because it requires regular token updates.

Open source DC/OS does not have fine-grained permissions for administration: all administrators have the same status and can add and remove other administrators. If finer control is required, you should use Mesosphere Enterprise DC/OS.

5 *https://dcos.io/docs/1.8/administration/haproxy-adminrouter/*

6 *https://docs.mesosphere.com/1.8/administration/id-and-access-mgt/sso/*

7 *https://dcos.io/docs/1.8/administration/id-and-access-mgt/configure-your-security/*

## Monitoring and Intrusion Detection

Monitoring and intrusion detection are important parts of a complete information security management system. DC/OS provides a wide range of metrics, monitoring, and logging outputs that you can capture and monitor in various ways. It is highly recommended that system monitoring and audit logging tools are not run on DC/OS, because there are a number of problems with that arrangement for both system monitoring and monitoring for security purposes.

DC/OS core components are regular Linux processes run as systemd units. You can use standard Linux monitoring tools to monitor DC/OS core components. Depending on the tools that you use, this might require some DC/OS–specific configuration for best results.

If you are monitoring or auditing DC/OS, you might also want to investigate these options:

- Docker monitoring tools (e.g., cAdvisor)
- Marathon metrics and alerts

## Adding SSL

Securing public web traffic with DC/OS is straightforward. It is recommended that the SSL connections of HTTPS are terminated with Marathon-LB. This reduces the number of applications with access to certificates (compared with terminating SSL at every application) and the amount of configuration is less. You also can use Marathon-LB to automatically redirect nonsecure connections to use HTTPS. For further security it is recommended to register your site for HSTS.[8]

Core DC/OS does not offer any key/certificate management facilities. There is an example DC/OS package that uses letsencrypt to generate certificates and automatically propagate them to Marathon-LB; this project is a good starting point, but it is not recommended for a production environment, because it automatically restarts Marathon-LB every 24 hours to propagate certificates, which might not be desirable.

8 *https://hstspreload.appspot.com/*

## Mesosphere Enterprise DC/OS

Mesosphere Enterprise DC/OS allows DC/OS users to have fine-grained access permissions. In Open Source DC/OS, all users have full access to all services. With DC/OS Enterprise, you can restrict the access of individual users and groups of users to specified services. This is not restricted to DC/OS core services: you can control access to user-defined services such as apps, jobs, or packages. The level at which you can specify permissions is very fine. To learn more, consult the enterprise documentation at *https://docs.mesosphere.com/1.8/administration/id-and-access-mgt/*.

In Mesosphere Enterprise DC/OS, you also can configure applications to run using service accounts with restricted permissions. For example, you can use this to ensure that a particular service can communicate only with other services that it specifically requires. This can reduce the security risk if an individual service is compromised; for example, by a remote code execution vulnerability.

Another security feature in Mesosphere Enterprise DC/OS is the ability to enforce access control by the admin router. The admin router performs authentication and access control permission checks on all requests

**NOTE**

In Open Source DC/OS, the admin router performs authentication (unless this has been disabled) but does not perform permissions checks because there are no fine-grained permissions in Open Source DC/OS.

You can configure Mesosphere Enterprise DC/OS to use strict mode, whereby all communication between running tasks is routed via the admin router

**NOTE**

If you are not using strict mode then it is possible that communication internal to the cluster is not using the admin router and so is not being checked.

This allows all communication to have security rules applied to it by the admin router and provides a single place to log all traffic. However, this could cause performance problems when scaling if the admin router becomes a bottleneck.

## Disaster Planning and Business Continuity

Business continuity plans that define actions to be taken in the case of disasters are a requirement of many businesses. What disasters might occur, and the effect that they can have on operations will vary for different systems. I will briefly describe some of the options available in DC/OS for business continuity planning.

### Durability of Data in DC/OS

Most disaster recovery processes require that stored business data can be retrieved and restored following a disaster. Typically, durable persistence is achieved by writing information to media such as a hard disk, solid-state disk (SSD), or tape storage. In a modern business scenario, persistence of data to a single disk does not provide sufficient durability guarantees (if failure of a single disk results in permanent loss of data, we will probably have problems).

The probability and proportion of data that can acceptably be lost is the durability requirement for stored data. Here, I am defining durability as the probability that data, after it is saved, cannot be feasibly retrieved after a given period of time. In practice, a business might have multiple requirements corresponding to different business cases; for example, a business could require the following:

- Regulatory requirements for 100 percent durability of stored transaction data for 7 years.
- Analytics requirement for 99.9 percent durability of traffic data for 1 year.

Durability might include requirements for offline and/or offsite backups of data to protect data from loss in case of extreme events such as a hacker or rogue employee deleting data from within a software system, or total destruction of a datacenter in a natural disaster.

There are a number of strategies to deliver durability, including the following:

- Block-level duplication (e.g., RAID, SAN).
- Application-level backup. This could include cloud backup (e.g., AWS offers write-only storage and versioned storage on S3).

- Data replication (e.g., Cassandra, HDFS). This can include software systems that replicate data across multiple datacenters

DC/OS packages for persistence—in particular Cassandra—include tools to simplify taking backups. You can configure Cassandra for multidatacenter redundancy. When using distributed systems such as Cassandra and HDFS, it is important to consider how they will react in case of network partitions (wherein a chunk of instances are unable to communicate with other instances in the cluster).

The approaches just outlined are all valid, and what is best for you will depend on your situation and resources.

## Recovering DC/OS After a Disaster

If all existing systems are lost as a result of a disaster such as total loss of a datacenter, a DC/OS cluster can be replicated on to alternate infrastructure if available and if DC/OS configuration has been well maintained.

Having a good orchestration system in place to start up a replacement DC/OS cluster at another location the best approach to recovering from a major disaster. After a replacement cluster is started, you should install package and app configurations, ideally using scripts for the DC/OS CLI. Once that is complete the following tasks remain: restoring persistent data from backups, configuring hardware load balancers/reverse proxies, and updating public DNS records.

CHAPTER 7

# Implications of Using DC/OS

Moving to Mesos was a huge operational win. We codified our configurations and can deploy slowly to preserve hit-rate and avoid causing pain for persistent stores as well as grow and scale this tier with higher confidence.[1]

In this report, I have explained how DC/OS works and how you can use it to run software applications and services. In this chapter, I take a step back and explain at a high level what DC/OS does, which I think is most exciting. I will then get back into the detail and explore the specific benefits of using DC/OS from different perspectives and show that it meets the requirements set out in the first chapter.

Successful cloud platforms such as Amazon Web Services (AWS), Microsoft Azure, and a Google Cloud Platform enabled companies like Netflix and Snapchat to build scalable businesses. Each individual platform provides a range of high-level services that can be configured and managed with a single set of tools. Major tech companies such as Facebook and Google have built similar platforms internally. DC/OS is exciting to me because it can provide a comparable platform and suite of high-level services to any company, whether they want to run in the cloud or in private datacenters.

1 *https://blog.twitter.com/2017/the-infrastructure-behind-twitter-scale*

By building on top of Mesos and Docker, DC/OS abstracts away concerns about hardware, operating systems, and the locations of running services. What this provides is an abstraction for applications that is analogous to the abstraction for machines provided by hardware virtualization technologies. This application-level abstraction is powerful and exciting because it allows operational teams to radically rethink their approaches to delivering reliability and performance.

## How DC/OS Addresses Enterprise Application Architecture Requirements

In Chapter 2, I set out technical and business requirements that DC/OS and the Modern Enterprise Architecture (MEA) should meet to be useful for organizations. In this section, I will explore the benefits of DC/OS from the different perspectives of software development and operations, and I will recall the requirements set out in Chapter 2 and summarize how they are met.

### Operational Benefits of DC/OS

Here are key benefits that using DC/OS brings to operations teams:

- DC/OS provides multiple strategies to deliver high availability (HA) and reliability. Deploying applications and services as Marathon apps is simple, and provides health checks and automatic handling of applications upgrades and configuration changes without downtime. Redundancy is easily configured with automatic failover via load balancing with Marathon-LB or Minuteman.
- For more advanced operational automation, there is the option to write a custom Mesos framework, which gives complete control to implement custom HA strategies and operational automation. DC/OS frameworks can make use of DC/OS core components such as Zookeeper, AdminRouter, and Minuteman for common tasks.
- Complex, interdependent systems can run fault tolerant on DC/OS by using Mesos task scheduling combined with load balancing and service discovery.

- Configuration and deployment of DC/OS can be easily written as code and automated so that even the worst disaster situations can be handled reliably.
- Segregation of resources means that we can run diverse workloads with better resource utilization on a single cluster. Contingency capacity can be allocated easily to any application or service providing greater resiliency than siloed infrastructure.
- By having just two distinct node types (Masters and Agents) basic security and provisioning is easily managed. A handful of base OS configurations are easier to harden and test than many different node setups and complex setup scripts.

## Development Benefits of DC/OS

Here are key benefits that DC/OS brings to software engineering teams:

- First-class support for containers allows software developers to specify their application runtime environment, which fosters consistency between development, testing, and deployment.
- Straightforward access to powerful high-level abstractions provided by packages allows software developers to concentrate on delivery of business features rather than tackling distributed system or operational challenges.
- Provision of a good set of package services allows us to write software that is stateless (relying on services for state) and so much simpler to develop and manage.
- Built-in service discovery and load balancing allow developers to use microservice patterns safely and without creating operational risks and challenges. The use of these approaches allows developers to make smaller, lower-risk updates to applications with higher frequency which generally results in improved reliability.
- DC/OS can be run on a range of infrastructure, both on-premises and in public cloud environments. Whatever the infrastructure, deployment and configuration remain the same, so applications, workloads, and systems can be easily reproduced in different infrastructure environments. For developers, this means that development environments can easily use different

infrastructure from production environments; for example, develop on cloud and deploy to a private datacenter if both are running DC/OS.

## MEA Requirements

In Chapter 2, I set out the requirements that I had for the MEA to be a great solution to a wide range of uses. Here, I will recall those and explain how DC/OS can meet them.

***Requirement:*** *Meet the technical needs of highly connected systems, including high transaction rates and volume, and durable persistence.*

DC/OS clusters can scale to thousands of nodes, and DC/OS applications can be can be horizontally scaled to handle demand and make use of dedicated load balancers. Internally, Minuteman VIPs provide resilient service discovery and straightforward scaling. Durable persistence can be provided by packages such as Cassandra or HDFS, which also scale horizontally and handle high transaction rates.

***Requirement:*** *Deliver state of the art reliability and consistency, within the bounds of CAP theorem (for distributed systems) and limitations of networked applications.*

DC/OS is a remarkably resilient system. The ability of the system to automatically handle and recover from individual failures (such as the loss of a node) without operational intervention allows systems running on DC/OS to be very reliable. DC/OS packages allow users to make use of some sophisticated and reliable, fault-tolerant distributed systems such as Cassandra and Kafka. You can handle arbitrary failure handling by writing your own Mesos framework.

***Requirement:*** *Facilitate high-volume data collection and storage, fast analysis, and machine learning.*

DC/OS has first-class support for analytics and allows intensive computational workloads to be run on the same cluster as other operational workloads, eliminating the need for Extract, Transform and Load (ETL) and data warehousing. The Spark package is an easy-to-run and exceptionally fast analytics platform that can read and write data from a variety of sources including Cassandra, HDFS, and AWS S3.

***Requirement:*** *Enable high productivity and low toil in teams that use the system, principally software developers and system administrators.*

DC/OS facilitates a number of modern software development practices and has minimal technology tie-in. Packages provide developers with high-level services, load-balanced service discovery makes using microservices patterns straightforward, and automated change management facilitates continuous deployment and Agile development.

DC/OS can revolutionize system administration in many organizations. Operationally, it is very similar to tools used by leading companies such as Google, Facebook, and Twitter. By abstracting away hardware concerns, allowing containerization, and enabling automation of many tasks, DC/OS can dramatically reduce the operational workload associated with complex and distributed systems.

***Requirement:*** *Be compatible with multiple infrastructure options and have low switching costs associated with moving an operational system from one infrastructure to another to avoid vendor lock-in.*

DC/OS systems can be run effectively on cloud or local infrastructure. If good practices have been followed (e.g., Infrastructure as Code), replicating an entire DC/OS cluster in another location or on different infrastructure is straightforward and switching costs are dramatically reduced compared with using the proprietary services of specific cloud platforms.

***Requirement:*** *Be compatible with a range of technologies and allow for concurrent use of different technologies for minimal switching costs to avoid technology lock-in.*

You can write DC/OS applications in almost any programming language and using almost any tools or frameworks (provided they are Linux compatible). Although DC/OS comes with some specific technologies such as Marathon, there is little lock-in. Alternatives to Marathon can easily be run on Mesos. All the core DC/OS components are open source, so it's easy to begin writing your own alternatives, if necessary. Most packages are not specific to DC/OS and thus can be run without it allowing the potential to easily switch away from DC/OS.

***Requirement:*** *Makes good use of computational and human resources.*

> The Mesos scheduler is able to dramatically improve computer resource utilization by allowing safe, effective scheduling of diverse tasks on shared hardware. DC/OS uses common tools that are familiar to many software engineers and system administrators—even if they have not used DC/OS before—such as Docker, Mesos, and Zookeeper. These tools are not DC/OS-specific and are widely used. Similarly, packages are not specific to DC/OS and are widely used. Businesses should have no problem finding competent engineers and administrators. DC/OS and most packages are open source, so it is possible for developers to see for themselves how it works. DC/OS is available free of charge, so adoption (and therefore training and experience) is not limited to large enterprises and people who have worked in them.

## Conclusion

This report has set out the requirements for an MEA to meet the needs of businesses that want to build and run reliable, internet-connected applications and services at scale, with development and operational efficiency. I have shown that DC/OS and the MEA meets those needs.

Until recently, systems like DC/OS have only been accessible to the largest and most advanced companies, such as Google, Amazon, Twitter, and Facebook, based on their investments into tooling, infrastructure, and technology. DC/OS is free, open source software and allows any company to use it without needing to sacrifice independence from specific suppliers, control over infrastructure, or flexibility in other software and technology choices.

In this report, I have explained how DC/OS works to provide an application platform, I have offered some software development and operational best practices for DC/OS users, and have highlighted limitations of DC/OS. These things should help you to design systems to run on DC/OS effectively. Hopefully, I have explained in enough detail how DC/OS functions "under the hood" that you can understand DC/OS behavior, and you can begin to troubleshoot DC/OS applications.

If you have read this report cover to cover, you should now feel confident to begin using DC/OS, and you should now know more about how DC/OS works and why you might choose to use it than you did before.

## About the Author

**Andrew Jefferson** is VP of Engineering at Tractable, a VC backed Artificial Intelligence company based in London. Andrew has worked on large scale systems at Apple and for startups in San Francisco and London. Andrew holds a degree from Cambridge University