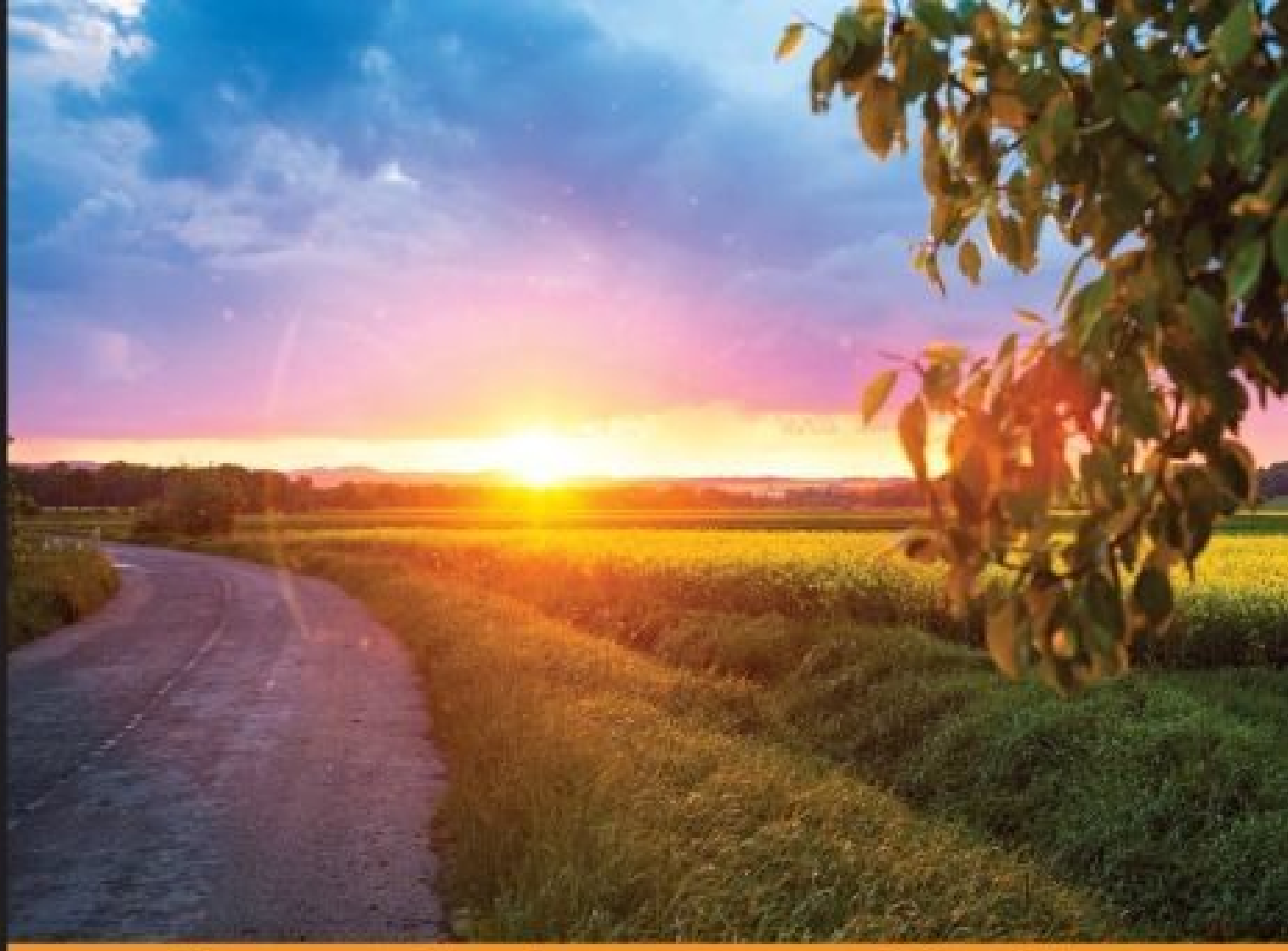


Community Experience Distilled

Apache Mesos Essentials

Build and execute robust and scalable applications using Apache Mesos

Dharmesh Kakadia

[PACKT] PUBLISHING open source*
community experience distilled

Apache Mesos Essentials

Table of Contents

Apache Mesos Essentials
Credits
About the Author
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers, and more
        Why subscribe?
        Free access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Downloading the color images of this book
        Errata
        Piracy
        Questions
1. Running Mesos
    Modern data centers
    Cluster computing frameworks
    Introducing Mesos
        The master
        Slaves
        Frameworks
    Why Mesos?
    Single-node Mesos clusters
        Mac OS
        Fedora
        Installing prerequisites
            CentOS
            Ubuntu
        Build Mesos

        Start Mesos
    Running test frameworks
    Mesos Web UI
    Multi-node Mesos clusters
    Mesos cluster on Amazon EC2
    Running Mesos using Vagrant
    The Mesos community
        Case studies
            Twitter
            HubSpot
            Airbnb
        Mailing lists
    Summary
2. Running Hadoop on Mesos
    An introduction to Hadoop
    Hadoop on Mesos
    Installing Hadoop on Mesos
    An example Hadoop job
    Advanced configuration for Hadoop on Mesos
        Task resource allocation
        Metrics reporting
            CSV
            Graphite
            Cassandra
        Authentication
        Container isolation
        Additional configuration parameters
    Summary
3. Running Spark on Mesos
    Introducing Spark
    Spark job scheduling
    Spark Standalone mode
    Spark on Mesos
    Tuning Spark on Mesos
    Summary
4. Complex Data Analysis on Mesos
    Complex data and the rise of the Lambda architecture
    Storm

- Storm on Mesos
- Storm-Mesos configuration
- Spark Streaming
  - Running Spark Streaming on Mesos
  - Tuning Spark Streaming
    - Selecting the batch size
    - Garbage collection
    - Concurrency
    - Handling failures
    - Task overheads
- NoSQL on Mesos
  - Cassandra on Mesos
  - Summary

5. Running Services on Mesos
- Introduction to services
- Marathon
  - The Marathon API
  - Running Marathon
  - Marathon example
  - Constraints
  - Event bus
  - The artifact store
  - Application groups
  - Application health checks
- Chronos
  - The Chronos REST API
  - Running Chronos
  - A Chronos example
- Aurora
  - Job life cycle
  - Running Aurora
  - Aurora cluster configuration
  - Aurora job configuration
  - An Aurora client
  - An Aurora example
  - Aurora cron jobs
- Service discovery
  - Mesos-DNS

Installing Mesos-DNS
Mesos-DNS configuration
Running Mesos-DNS
Packaging
Summary
6. Understanding Mesos Internals
The Mesos architecture
Mesos slave
Mesos master
Frameworks
Communication
Auxiliary services
Resource allocation
The Mesos scheduler
Weighted DRF
Reservation
Static reservation
Dynamic reservation
Resource isolation
Mesos containerizer
Docker containerizer
External containerizer
Fault tolerance
ZooKeeper
Failure detection and handling
Registry
Extending Mesos
Mesos modules
Module naming
Module compatibility
Allocation module
Mesos hooks and decorators
Task labels
Summary
7. Developing Frameworks on Mesos
The Mesos API
Mesos messages
The scheduler API

The SchedulerDriver API
The executor API
The ExecutorDriver API
Developing a Mesos framework
Setting up the development environment
Adding the framework scheduler
Adding the framework launcher
Deploying our framework
Building our framework
Adding an executor to our framework
Updating our framework scheduler
Running multiple executors
Advanced topics
Reconciliation
Stateful applications
Developer resources
Framework design patterns
Framework testing
RENDLER
Akka-mesos
Summary
8. Administering Mesos
Deployment
Upgrade
Monitoring
Container network monitoring
Multitenancy
Authorization and authentication
API rate limiting
High availability
Master high availability
Slave removal rate limiting
Slave recovery
Maintenance
Mesos interfaces
The Mesos REST interface
The Mesos CLI
Configuration

The Mesos master
The Mesos slave
Mesos build options
Summary
Index

Apache Mesos Essentials

Apache Mesos Essentials

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2015

Production reference: 1240615

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78328-876-2

www.packtpub.com

Credits

Author

Dharmesh Kakadia

Reviewers

Tomas Barton

Andrea Mostosi

Sai Warang

Acquisition Editor

Sonali Vernekar

Content Development Editor

Nikhil Potdukhe

Technical Editor

Mitali Somaiya

Copy Editor

Rashmi Sawant

Project Coordinator

Sanchita Mandal

Proofreaders

Stephen Copestake

Safis Editing

Indexer

Hemangini Bari

Graphics

Sheetal Aute

Production Coordinator

Komal Ramchandani

Cover Work

Komal Ramchandani

About the Author

Dharmesh Kakadia is a research fellow at Microsoft Research, who develops the next-generation cluster management systems. Before coming to MSR, he completed his MS in research from the International Institute of Information Technology, Hyderabad, where he worked on improving scheduling in cloud and big data systems. He likes to work at the intersection of systems and data and has published research in resource management at various venues. He is passionate about open source technologies and plays an active role in various open source communities. You can learn more about him at `@DharmeshKakadia` on Twitter.

I would like to thank my family members, friends, and colleagues for always being there for me. I would also like to thank the reviewers and the entire Packt Publishing staff for putting in the hard work to make sure that the quality of the book was up to the mark. Without help from all these people, this book would never have made it here.

About the Reviewers

Tomas Barton is a PhD candidate at Czech Technical University in Prague, who focuses on distributed computing, data mining, and machine learning. He has been experimenting with Mesos since its early releases. He has contributed to Debian packaging and maintains a Puppet module for automated Mesos installation management.

Andrea Mostosi is a technology enthusiast. He is an innovation lover from childhood. He started his professional career in 2003 and has worked on several projects, playing almost every role in the computer science environment. He is currently the CTO at The Fool, a company that tries to make sense of the Web and social data. During his free time, he likes to travel, run, cook, ride a bike, and write code.

I would like to thank my geek friends, Simone M, Daniele V, Luca T, Luigi P, Michele N, Luca O, Luca B, Diego C, and Fabio B. They are the smartest people I know and comparing myself with them has always pushed me to do better.

Sai Warang is a software developer working at a Canadian start-up called Shopify. He is currently working on making real-time tools to protect the hundreds of thousands of online merchants from fraud. In the past, he has studied computer science at the University of Waterloo and worked at Tagged and Zynga in San Francisco on various data analytics projects. He occasionally dabbles in creative writing.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Preface

Mesos makes it easier to develop and manage fault-tolerant and scalable distributed applications. Mesos provides primitives that allow you to program for the aggregated resource pool, without worrying about managing resources on individual machines. With Mesos, all your favorite frameworks, ranging from data processing to long-running services to data storage to Web serving, can share resources from the same cluster. The unification of infrastructure combined with the resilience built into Mesos also simplifies the operational aspects of large deployments. When running on Mesos, failures will not affect the continuous operations of applications.

With Mesos, everyone can develop distributed applications and scale it to millions of nodes.

What this book covers

Chapter 1, *Running Mesos*, explains the need for a data center operating system in the modern infrastructure and why Mesos is a great choice for it. It also covers how to set up singlenode and multimode Mesos installations in various environments.

Chapter 2, *Running Hadoop on Mesos*, discusses batch data processing using Hadoop on Mesos.

Chapter 3, *Running Spark on Mesos*, covers how to run Spark on Mesos. It also covers tuning considerations for Spark while running on Mesos.

Chapter 4, *Complex Data Analysis on Mesos*, demonstrates the various options for deploying lambda architecture on Mesos. It covers Storm, Spark Streaming, and Cassandra setups on Mesos in detail.

Chapter 5, *Running Services on Mesos*, introduces services and walks you through the different aspects of service architecture on Mesos. It covers the Marathon, Chronos, and Aurora frameworks in detail and helps you understand how services are deployed on Mesos.

Chapter 6, *Understanding Mesos Internals*, dives deep into Mesos fundamentals. It walks you through the implementation details of resource allocation, isolation, and fault tolerance in Mesos.

Chapter 7, *Developing Frameworks on Mesos*, covers specifics of framework development on Mesos. It helps you learn about the Mesos API by building a Mesos framework.

Chapter 8, *Administering Mesos*, talks about the operational aspects of Mesos. It covers topics related to monitoring, multitenancy, availability, and maintenance along with REST API and configuration details.

What you need for this book

To get the most of this book, you need to be familiar with Linux and have a basic knowledge of programming. Also, having access to more than one machine or a cloud service will enhance the experience due to the distributed nature of Mesos.

Who this book is for

This book is for anyone who wants to develop and manage data center scale applications using Mesos.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, folder names, filenames, pathnames, and configuration parameters are shown as follows: "The vagrant files and the `README` file included in the repository will provide you with more details."

A block of code is set as follows:

```
<property>
<name>mapred.mesos.framework.secretfile</name>
<value>/location/secretfile</value>
</property>
```

Any command-line input or output is written as follows:

```
ubuntu@local:~/mesos/ec2 $ ./mesos-ec2 destroy ec2-test
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "On the web UI, click on **+ New Job**, and it will pop up a panel with details of the job."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: http://www.packtpub.com/sites/default/files/downloads/1234OT_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. Running Mesos

This chapter will give you a brief overview of Apache Mesos and cluster computing frameworks. We will walk you through the steps for setting up Mesos on a single-node and multi-node setup. We will also see how to set up a Mesos cluster using Vagrant and on Amazon EC2. Throughout this book, we will refer to Apache Mesos and Mesos interchangeably. We will cover the following topics in this chapter:

- Modern data centers
- Cluster computing frameworks
- Introducing Mesos
- Why Mesos?
- A single-node Mesos cluster
- A multi-node Mesos cluster
- A Mesos cluster on Amazon EC2
- Running Mesos using Vagrant
- The Mesos community

Modern data centers

Modern applications are highly dependent on data. The manifold increase in the data generated and processed by organizations is continually changing the way we store and process it. When planning modern infrastructure for storing and processing the data, we can no longer hope to simply buy hardware with more capacity to solve the problem. Different frameworks for batch processing, stream processing, user-facing services, graph processing, and ad hoc analysis are every bit as important as the hardware they run on. These frameworks are the applications that power the data center world.

The size and variety of big data means traditional scale-up strategies are no longer adequate for modern workloads. Thus, large organizations have moved to distributed processing, where a large number of computers act as a single giant computer. The cluster is shared by many applications with varying resource requirements, and the efficient sharing of resources at this scale among multiple frameworks is the key to achieving high utilization. There is a need to consider all these machines as a single warehouse scale computer. Mesos is designed to be the kernel of such computers.

Traditionally, frameworks run in silos and resources are statically partitioned among them, which leads to an inefficient use of resources. The need to consider a large number of commodity machines as a single computer, and the ability to share resources in an elastic manner by all the frameworks requires a cluster computing framework. Mesos is inspired by the idea of sharing resources in a cluster between multiple frameworks while providing resource isolation.

Cluster computing frameworks

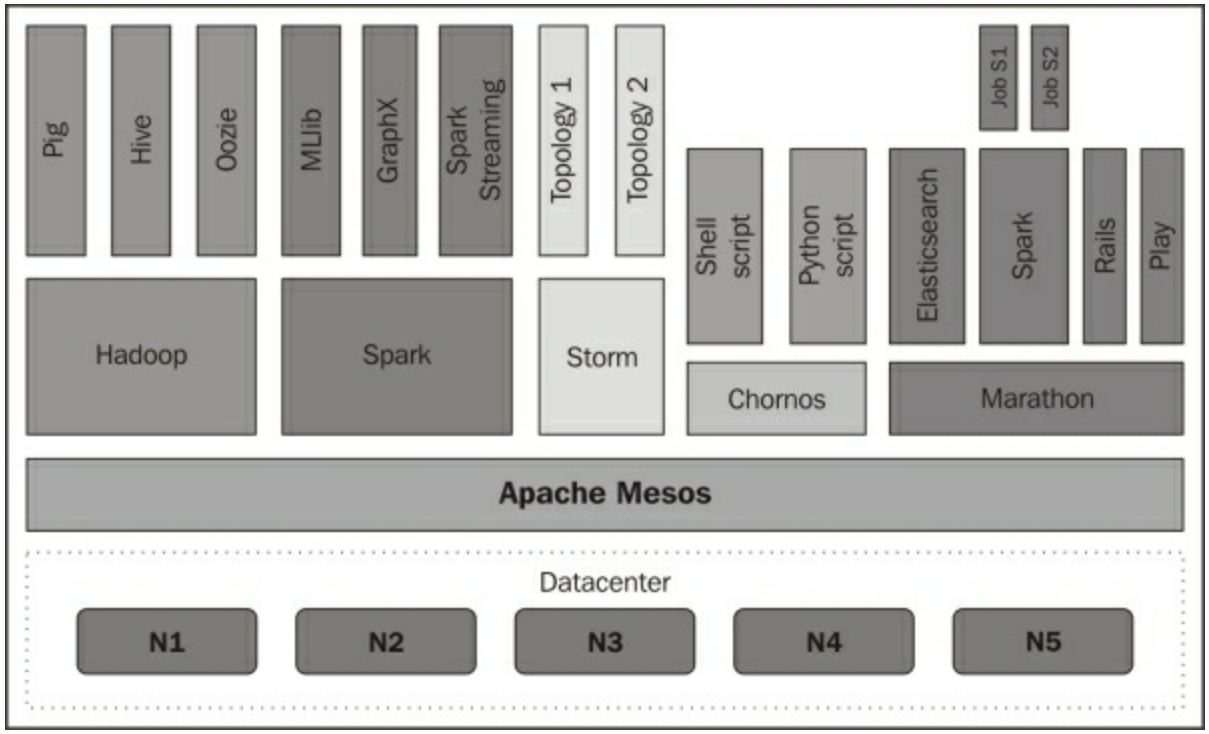
In modern clusters, the computing requirements of different frameworks are radically different, and organizations need to run multiple frameworks and share data and resources between them. Resource managers face challenging and competing goals:

- Efficiency: Efficiently sharing resources is the prime goal of cluster management software.
- Isolation: When multiple tasks are sharing resources, one of the most important considerations is to ensure resource isolation. Isolation combined with proper scheduling is the foundation of guaranteeing **service level agreements** (**SLAs**).
- Scalability: The continuous growth of modern infrastructure requires cluster managers to scale linearly. One important scalability metric is the delay experienced in decision-making by the framework.
- Robustness: Cluster management is a central component, and robust behavior is required for continuous business operations. There are many aspects contributing to robustness, from well-tested code to fault-tolerant design.
- Extensible: Cluster management software is a huge development in any organization and has been used for decades. During an operation, the changes in the organization policy and/or the hardware invariably require change in how the cluster resources are managed. Thus, maintainability becomes an important consideration for large organizations. It should be configurable considering constraints (for example location, hardware) and support for multiple frameworks.

Introducing Mesos

Mesos is a cluster manager aiming for improved resource utilization by dynamically sharing resources among multiple frameworks. It was started at the University of California, Berkeley in 2009 and is in production use in many companies, including Twitter and Airbnb. It became an Apache top-level project in July 2013 after nearly two years in incubation.

Mesos shares the available capacity of machines (or nodes) among jobs of different natures, as shown in the following figure. Mesos can be thought of as a kernel for the data center that provides a unified view of resources on all nodes and seamless access to these resources in a manner similar to what an operating system kernel does for a single computer. Mesos provides a core for building data center applications and its main component is a scalable two-phased scheduler. The Mesos API allows you to express a wide range of applications without bringing the domain-specific information into the Mesos core. By remaining focused on core, Mesos avoids problems that are seen with monolithic schedulers.



Mesos as a data center kernel

The following components are important for understanding the overall Mesos architecture. We will briefly describe them here and will discuss the overall architecture in more detail in Chapter 6, *Understanding Mesos Internals*.

The master

The master is responsible for mediating between the slave resources and frameworks. At any point, Mesos has only one active master, which is elected using ZooKeeper via distributed consensus. If Mesos is configured to run in a fault-tolerant mode, one master is elected through the distributed leader election protocol, and the rest of them stay in standby mode. By design, Mesos' master is not meant to do any heavy lifting tasks itself, which simplifies the master design. It offers slave resources to frameworks in the form of resource offers and launches tasks on slaves for accepted offers. It also is responsible for all the communication between the tasks and frameworks.

Slaves

Slaves are the actual workhorses of the Mesos cluster. They manage resources on individual nodes and are configured with a resource policy to reflect the business priorities. Slaves manage various resources, such as CPU, memory, ports, and so on, and execute tasks submitted by frameworks.

Frameworks

Frameworks are applications that run on Mesos and solve a specific use case. Each framework consists of a scheduler and executor. A scheduler is responsible for deciding whether to accept or reject the resource offers. Executors are resource consumers and run on slaves and are responsible for running tasks.

Why Mesos?

Mesos offers huge benefits to both developers and operators. The ability of Mesos to consolidate various frameworks on a common infrastructure not only saves on infrastructure costs, but also provides operational benefits to the Ops teams and simplifies developers' view of the infrastructure, ultimately leading to business success. Here are some of the reasons for organizations to embrace Mesos:

- Mesos supports a wide variety of workloads, ranging from batch processing (Hadoop), interactive analysis (Spark), real-time processing (Storm, Samza), graph processing (Hama), high-performance computing (MPI), data storage (HDFS, Tachyon, and Cassandra), web applications (play), continuous integration (Jenkins, GitLab), and a number of other frameworks. Moreover, meta-scheduling frameworks, such as Marathon and Aurora can run most of the existing applications on Mesos without any modification. Mesos is an ideal choice for running containers at scale. This flexibility makes Mesos very easy to adopt.
- Mesos improves utilization through elastic resource sharing between various frameworks. Without a common data center operating system, different frameworks have to run on siloed hardware. Such static partitioning of resources leads to resource fragmentation, limiting the utilization and throughput. Dynamic resource sharing through Mesos drives higher utilization and throughput.
- Mesos is an open source project with a vibrant community. The Mesos pluggable architecture makes it easy to customize it for the organization's needs. Combined with the fact that Mesos runs on a wide range of operating systems and hardware choices, it provides the widest range of options and guards against vendor lock-in. Thus, developing against the Mesos API provides many choices of infrastructure for running them. It also means that the Mesos applications will be portable across bare metal, virtualized infrastructure, and cloud providers.
- Probably, the most important benefit of Mesos is empowering developers to build modern applications with increased productivity. As developers move from developing applications for a single computer to a program against data centers, they need an API that allows them to focus on their logic and not on the nitty-gritty details of the distributed infrastructure. With Mesos, the developers do not have to worry about the distributed aspects and can focus on the domain-specific logic of the application. Mesos provides a rich API to develop scalable and fault-tolerant distributed applications, as we will see in Chapter 7, *Developing Frameworks on Mesos*.
- Operating a large infrastructure is challenging. Mesos simplifies infrastructure

management by providing a unified view of resources. It brings a lot of agility and deploying new services takes a shorter time with Mesos since there is no separate cluster to be allocated. Mesos is extremely Ops-friendly and treats infrastructure resources like cattle and not pets. What this means is that Mesos is resilient in the face of failures and can automatically ensure high availability, without requiring manual intervention. Mesos supports multitenant deployment with strong isolation, which is essential for operating at scale. Mesos provides full-featured REST, web, and command-line interfaces and integrates well with the existing tools, as we will see in Chapter 8, *Administering Mesos*.
- Mesos is battle-tested at Twitter, Airbnb, HubSpot, eBay, Netflix, Conviva, Groupon, and a number of other organizations. Mesos catering to the needs of a wide variety of use cases across different companies is proof of Mesos's versatility as a data center kernel.

Mesos also offers significant benefits over traditional virtualization-based infrastructure:

- Most of the applications do not require strong isolation provided by virtual machines and can run on container-based isolation in Mesos. Since containers have much lower overheads than to VMs, this not only leads to higher consolidation but also has other benefits, such as fast start-up time and so on.
- Mesos reduces infrastructure complexity drastically compared to VMs.
- Achieving fault tolerance and high availability using VMs is very costly and hard. With Mesos, hardware failures are transparent to applications, and the Mesos API helps developers in embracing failures.

Now that we have seen the benefits of running Mesos, let's create a single-node Mesos cluster and start exploring Mesos.

Single-node Mesos clusters

Mesos runs on Linux and Mac OS X. A single machine Mesos setup is the simplest way of trying out Mesos, so we'll go through it first. Currently, Mesos does not provide binary packages for different operating systems, and we need to compile it from the source. There are binary packages available by community.

Mac OS

Homebrew is a Linux-style package manager for Mac. Homebrew provides a formula for Mesos and compiles it locally. We need to perform the following steps to install Mesos on Mac:

1. Install Homebrew from http://brew.sh/.
2. Homebrew requires Java to be installed. Mac has Java installation by default, so we just have to make sure that `JAVA_HOME` is set correctly.
3. Install Mesos using Homebrew with the following command:

```
mac@master:~ $ brew install mesos
```

Tip

Although Homebrew provides a way to try out Mesos on Mac, the production setup should run on Linux.

Fedora

Starting from Fedora 21, the Fedora repository contains the Mesos packages. There are `mesos-master` and `mesos-slave` packages to be installed on the master and slave respectively. Also, there is a `mesos` package, which contains both the master and slave packages. To install the `mesos` package on Fedora version >= 21, use the following command:

```
fedora@master:~ $ sudo yum install -y mesos
```

Now we can continue with the `Start Mesos` section to run Mesos. For Fedora Version <= 21, we have to install the dependencies and Mesos from the source, similar to CentOS as explained in the following section.

Installing prerequisites

Mesos requires the following prerequisites to be installed:

- g++ (>=4.1)
- Python 2.6 developer packages
- Java Development Kit (>=1.6) and Maven
- The cURL library
- The SVN development library
- **Apache Portable Runtime Library** (**APRL**)
- **Simple Authentication and Security Layer** (**SASL**) library

Additionally, we will need autoconf (Version 1.12) and libtool if we want to build Mesos from the git repository. The installation of this software differs for various operating systems. We will show you the steps to install Mesos on Ubuntu 14.10 and CentOS 6.5. The steps for other operating systems are also fairly similar.

CentOS

Use the following commands to install all the required dependencies on CentOS:

1. Currently, the CentOS default repository does not provide a SVN library >= 1.8. So, we need to add a repository, which provides it. Create a new `wandisco-svn.repo` file in `/etc/yum.repos.d/` and add the following lines:

   ```
   centos@master:~ $ sudo vim /etc/yum.repos.d/wandisco-svn.repo
    [WandiscoSVN]
     name=Wandisco SVN Repo
     baseurl=http://opensource.wandisco.com/centos/6/svn-
   1.8/RPMS/$basearch/
     enabled=1
     gpgcheck=0
   ```

   Now, we can install `libsvn` using the following command:

   ```
   centos@master:~ $ sudo yum groupinstall -y "Development Tools"
   ```

2. We need to install Maven by downloading it, extracting it, and putting it in `PATH`. The following commands extract it to `/opt` after we download it and link `mvn` to `/usr/bin`:

   ```
   centos@master:~ $ wget
   http://mirror.nexcess.net/apache/maven/maven-
   3/3.0.5/binaries/apache-maven-3.0.5-bin.tar.gz
   ```

```
centos@master:~ $ sudo tar -zxf apache-maven-3.0.5-bin.tar.gz -C
/opt/
centos@master:~ $ sudo ln -s /opt/apache-maven-3.0.5/bin/mvn
/usr/bin/mvn
```

3. Install the other dependencies using the following command:

```
centos@master:~ $ sudo yum install -y python-devel java-1.7.0-
openjdk-devel zlib-devel libcurl-devel openssl-devel cyrus-sasl-
devel cyrus-sasl-md5 apr-devel subversion-devel
```

Ubuntu

Use the following command to install all the required dependencies on Ubuntu:

```
ubuntu@master:~ $ sudo apt-get -y install build-essential openjdk-6-
jdk python-dev python-boto libcurl4-nss-dev libsasl2-dev libapr1-dev
libsvn-dev maven
```

Build Mesos

Once we have installed all the required software, we can follow these steps to build Mesos:

1. Download the latest stable release from http://mesos.apache.org/downloads/. At the time of writing, the latest release is 0.21.0. Save the `mesos-0.21.0.tar.gz` file in some location. Open the terminal and go to the directory, where we have saved the file or you can directly run the following command on the terminal to download Mesos:

   ```
   ubuntu@master:~$ wget
   http://www.apache.org/dist/mesos/0.21.0/mesos-0.21.0.tar.gz
   ```

2. Extract Mesos with the following command and enter the extracted directory. Note that the second command will remove the downloaded `.tar` file, and rename the version name from the extracted folder:

   ```
   ubuntu@master:~ $ tar -xzf mesos-*.tar.gz
   ubuntu@master:~ $ rm mesos-*.tar.gz ; mv mesos-* mesos
   ubuntu@master:~ $ cd mesos
   ```

3. Create a `build` directory. This will contain the compiled Mesos binaries. This step is optional, but it is recommended. The build can be distributed to slaves instead of recompiling on every slave:

   ```
   ubuntu@master:~/mesos $ mkdir build
   ubuntu@master:~/mesos $ cd build
   ```

4. Configure the installation by running the `configure` script:

   ```
   ubuntu@master:~/mesos/build $ ../configure
   ```

   The `configure` script supports tuning the build environment, which can be listed by running `configure --help`. If there are any dependencies missing, then the configure script will report, and we can go back and install the missing packages. Once the configuration is successful, we can continue with the next step.

5. Compile it using `make`. This might take a while. The second step is `make check`:

   ```
   ubuntu@master:~/mesos/build $ make
   ubuntu@master:~/mesos/build $ make check
   ```

   The `make check` step builds the example framework, and we can now run Mesos from the build folder directly without installing it.

6. Install Mesos using the following command:

```
ubuntu@master:~/mesos/build $ make install
```

The list of commands that Mesos provides is as follows:

Command	Use
`mesos-local.sh`	This command launches an in-memory cluster within a single process.
`mesos-tests.sh`	This command runs the Mesos test case suite.
`mesos.sh`	This is a wrapper script used to launch the Mesos commands. Running without any arguments shows all the available commands.
`gdb-mesos-*`	This command launches the corresponding processes in debugging mode using gdb.
`lldb-mesos-*`	This command launches the corresponding processes in debugging mode using lldb.
`valgrind-mesos-*`	This command launches the corresponding Valgrind instrumentation framework.
`mesos-daemon.sh`	This command starts/stops a Mesos daemon process.
`mesos-start-cluster.shmesos-stop-cluster.sh`	This command starts and stops the Mesos cluster on nodes in the [install-prefix]/var/mesos/deploy/masters and [install-prefix]/var/mesos/deploy/slaves files.
`mesos-start-masters.shmesos-stop-masters.sh`	This command starts and stops mesos masters on nodes listed in the masters file.
`mesos-start-slaves.shmesos-stop-slaves.sh`	This command starts and stops the mesos slaves on nodes listed in the slaves file.

We can now start the local Mesos cluster using the `mesos-local` command, which will start both the master and slave in a single process and provide a quick way to check the Mesos installation.

Start Mesos

Now we are ready to start the Mesos process. First, we need to create a directory for the Mesos replicated logs with read-write permissions:

```
ubuntu@master:~ $ sudo mkdir -p /var/lib/mesos
ubuntu@master:~ $ sudo chown `whoami` /var/lib/mesos
```

Now, we can start the master with the following command, specifying the directory we created:

```
ubuntu@master:~ $ mesos-master --work_dir=/var/lib/mesos
I1228 07:29:16.367847  2900 main.cpp:167] Build: 2014-12-26 06:31:26
by ubuntu
I1228 07:29:16.368180  2900 main.cpp:169] Version: 0.21.0
I1228 07:29:16.387505  2900 leveldb.cpp:176] Opened db in 19.050311ms
I1228 07:29:16.390425  2900 leveldb.cpp:183] Compacted db in
2.731972ms
...
I1228 07:29:16.474812  2900 main.cpp:292] Starting Mesos master
...
I1228 07:29:16.488203  2904 master.cpp:318] Master 20141228-072916-
251789322-5050-2900 (ubuntu-master) started on master:5050
...
I1228 07:29:16.510967  2903 master.cpp:1263] The newly elected leader
is master@master:5050 with id 20141228-072916-2
51789322-5050-2900
I1228 07:29:16.511157  2903 master.cpp:1276] Elected as the leading
master!
...
```

The output here lists the build version, various configurations that the master has used, and the master ID of the cluster. The slave process should be able to connect to the master. The slave process can specify the IP address or the hostname of the master by the `--master` option. In the rest of the book, we will assume that the machine on which the master is running has the hostname `master` and should be replaced with an appropriate hostname or IP address.

```
ubuntu@master:~ $ mesos-slave --master=master:5050
I1228 07:33:32.415714  4654 main.cpp:142] Build: 2014-12-26 06:31:26
by vagrant
I1228 07:33:32.415992  4654 main.cpp:144] Version: 0.21.0
I1228 07:33:32.416199  4654 containerizer.cpp:100] Using isolation:
posix/cpu,posix/mem
```

```
I1228 07:33:32.443282  4654 main.cpp:165] Starting Mesos slave
I1228 07:33:32.447244  4654 slave.cpp:169] Slave started on
1)@master:5051
I1228 07:33:32.448254  4654 slave.cpp:289] Slave resources:
cpus(*):2; mem(*):1961; disk(*):35164; ports(*):[31000-32000
]
I1228 07:33:32.448619  4654 slave.cpp:318] Slave hostname: master
I1228 07:33:32.462025  4655 slave.cpp:602] New master detected at
master@master5050
...
```

The output confirms the connection to the master and lists the slave resources. Now, the cluster is running with one slave ready to run the frameworks.

Running test frameworks

Mesos includes various example test frameworks written in C++, Java, and Python. They can be used to verify that the cluster is configured properly. The following test framework is written in C++, and it runs five sample applications. We will run it using the following command:

```
ubuntu@master:~/mesos/build/src $ ./test-framework --
master=master:5050
I1228 08:53:13.303910  6044 sched.cpp:137] Version: 0.21.0
I1228 08:53:13.312556  6065 sched.cpp:234] New master detected at
master@master
:5050
I1228 08:53:13.313287  6065 sched.cpp:242] No credentials provided.
Attempting to register without authentication
I1228 08:53:13.316956  6061 sched.cpp:408] Framework registered with
20141228-085231-251789322-5050-5407-0001
Registered!
Received offer 20141228-085231-251789322-5050-5407-O3 with
mem(*):1961; disk(*):35164; ports(*):[31000-32000]; cpus(*):2
Launching task 0 using offer 20141228-085231-251789322-5050-5407-O3
Launching task 1 using offer 20141228-085231-251789322-5050-5407-O3
Task 0 is in state TASK_RUNNING
Task 0 is in state TASK_FINISHED
Task 1 is in state TASK_RUNNING
Task 1 is in state TASK_FINISHED
Received offer 20141228-085231-251789322-5050-5407-O4 with
mem(*):1961; disk(*):35164; ports(*):[31000-32000]; cpus(*):2
Launching task 2 using offer 20141228-085231-251789322-5050-5407-O4
Launching task 3 using offer 20141228-085231-251789322-5050-5407-O4
Task 2 is in state TASK_RUNNING
Task 2 is in state TASK_FINISHED
Task 3 is in state TASK_RUNNING
Task 3 is in state TASK_FINISHED
Received offer 20141228-085231-251789322-5050-5407-O5 with
mem(*):1961; disk(*):35164; ports(*):[31000-32000]; cpus(*):2
Launching task 4 using offer 20141228-085231-251789322-5050-5407-O5
Task 4 is in state TASK_RUNNING
Task 4 is in state TASK_FINISHED
I1228 08:53:15.337805  6059 sched.cpp:1286] Asked to stop the driver
I1228 08:53:15.338147  6059 sched.cpp:752] Stopping framework
'20141228-085231-251789322-5050-5407-0001'
I1228 08:53:15.338543  6044 sched.cpp:1286] Asked to stop the driver
```

Here the output shows the framework connected to the master and receives the resource offers from the master. It also shows the various states of the tasks it has launched. The Java example framework is included in the `src/example/java` folder:

```
ubuntu@master:~/mesos/build/src/examples/java $ ./test-framework
master:5050
I1228 08:54:39.290570  7224 sched.cpp:137] Version: 0.21.0
I1228 08:54:39.302083  7250 sched.cpp:234] New master detected at
master@master:5050
I1228 08:54:39.302613  7250 sched.cpp:242] No credentials provided.
Attempting to register without authentication
I1228 08:54:39.307786  7250 sched.cpp:408] Framework registered with
20141228-085231-251789322-5050-5407-0002
Registered! ID = 20141228-085231-251789322-5050-5407-0002
Received offer 20141228-085231-251789322-5050-5407-O6 with cpus: 2.0
and mem: 1961.0
Launching task 0 using offer 20141228-085231-251789322-5050-5407-O6
Launching task 1 using offer 20141228-085231-251789322-5050-5407-O6
Status update: task 1 is in state TASK_RUNNING
Status update: task 0 is in state TASK_RUNNING
Status update: task 1 is in state TASK_FINISHED
Finished tasks: 1
Status update: task 0 is in state TASK_FINISHED
Finished tasks: 2
Received offer 20141228-085231-251789322-5050-5407-O7 with cpus: 2.0
and mem: 1961.0
Launching task 2 using offer 20141228-085231-251789322-5050-5407-O7
Launching task 3 using offer 20141228-085231-251789322-5050-5407-O7
Status update: task 2 is in state TASK_RUNNING
Status update: task 2 is in state TASK_FINISHED
Finished tasks: 3
Status update: task 3 is in state TASK_RUNNING
Status update: task 3 is in state TASK_FINISHED
Finished tasks: 4
Received offer 20141228-085231-251789322-5050-5407-O8 with cpus: 2.0
and mem: 1961.0
Launching task 4 using offer 20141228-085231-251789322-5050-5407-O8
Status update: task 4 is in state TASK_RUNNING
Status update: task 4 is in state TASK_FINISHED
Finished tasks: 5
I1228 08:54:41.788455  7248 sched.cpp:1286] Asked to stop the driver
I1228 08:54:41.788652  7248 sched.cpp:752] Stopping framework
'20141228-085231-251789322-5050-5407-0002'
I1228 08:54:41.789008  7224 sched.cpp:1286] Asked to stop the driver
```

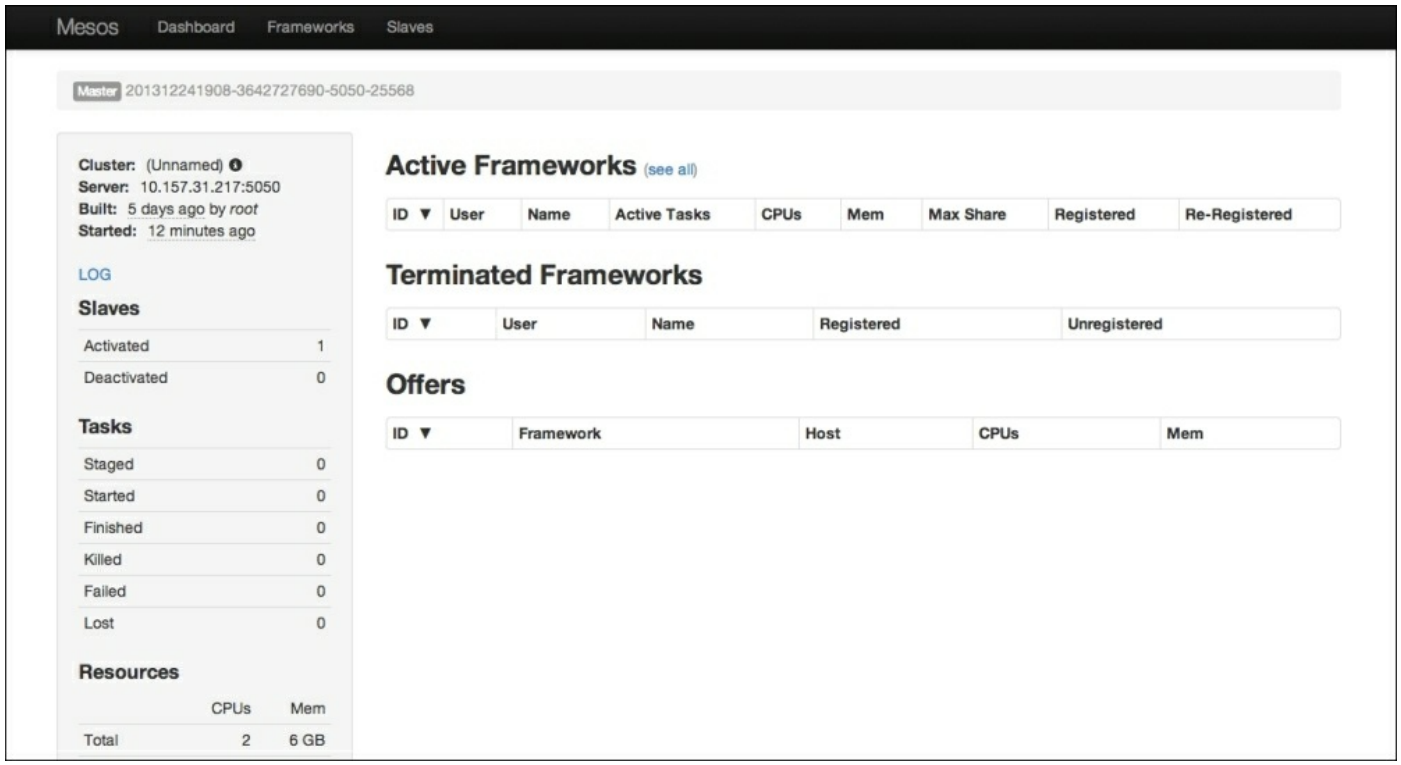
Similarly, the Python example framework is included in the `src/example/python`

folder and shows `frameworkId` and the various tasks states:

```
ubuntu@master:~/mesos/build/src/examples/python $./test-framework
master:5050
I1228 08:55:52.389428  8516 sched.cpp:137] Version: 0.21.0
I1228 08:55:52.422859  8562 sched.cpp:234] New master detected at
master@master:5050
I1228 08:55:52.424178  8562 sched.cpp:242] No credentials provided.
Attempting to register without authentication
I1228 08:55:52.428395  8562 sched.cpp:408] Framework registered with
20141228-085231-251789322-5050-5407-0003
Registered with framework ID 20141228-085231-251789322-5050-5407-0003
Received offer 20141228-085231-251789322-5050-5407-O9 with cpus: 2.0
and mem: 1961.0
Launching task 0 using offer 20141228-085231-251789322-5050-5407-O9
Launching task 1 using offer 20141228-085231-251789322-5050-5407-O9
Task 0 is in state TASK_RUNNING
Task 1 is in state TASK_RUNNING
Task 0 is in state TASK_FINISHED
Received message: 'data with a \x00 byte'
Task 1 is in state TASK_FINISHED
Received message: 'data with a \x00 byte'
Received offer 20141228-085231-251789322-5050-5407-O10 with cpus: 2.0
and mem: 1961.0
Launching task 2 using offer 20141228-085231-251789322-5050-5407-O10
Launching task 3 using offer 20141228-085231-251789322-5050-5407-O10
Task 2 is in state TASK_RUNNING
Task 2 is in state TASK_FINISHED
Task 3 is in state TASK_RUNNING
Task 3 is in state TASK_FINISHED
Received message: 'data with a \x00 byte'
Received message: 'data with a \x00 byte'
Received offer 20141228-085231-251789322-5050-5407-O11 with cpus: 2.0
and mem: 1961.0
Launching task 4 using offer 20141228-085231-251789322-5050-5407-O11
Task 4 is in state TASK_RUNNING
Task 4 is in state TASK_FINISHED
All tasks done, waiting for final framework message
Received message: 'data with a \x00 byte'
All tasks done, and all messages received, exiting
I1228 08:55:54.136085  8561 sched.cpp:1286] Asked to stop the driver
I1228 08:55:54.136147  8561 sched.cpp:752] Stopping framework
'20141228-085231-251789322-5050-5407-0003'
I1228 08:55:54.136261  8516 sched.cpp:1286] Asked to stop the driver
```

Mesos Web UI

Mesos provides a web UI for reporting information about the Mesos cluster. It can be accessed from `<master-host>:<port>`; in our case, this will be `http://master:5050`. This includes the slaves, aggregated resources, frameworks, and so on. Here is the screenshot of the web interface:



Mesos web interface

Multi-node Mesos clusters

We can repeat the previous procedure to manually start `mesos-slave` on each of the slave nodes to set up the cluster, but this is labor-intensive and error-prone for large clusters. Mesos includes a set of scripts in the deploy folder that can be used to deploy Mesos on a cluster. These scripts rely on SSH to perform the deployment. We need to set up a password less SSH. We will set up a cluster with two slave nodes (`slave1`, `slave2`) and a master node (`master`).

Let's configure our cluster to make sure that they have connectivity between them after installing all the prerequisites on all the nodes. The following commands will generate a `ssh` key and will copy them to both the slaves:

```
ubuntu@master:~ $ ssh-keygen -f ~/.ssh/id_rsa -P ""
ubuntu@master:~ $ ssh-copy-id -i ~/.ssh/id_rsa.pub ubuntu@slave1
ubuntu@master:~ $ ssh-copy-id -i ~/.ssh/id_rsa.pub ubuntu@slave2
```

We need to copy the compiled Mesos to both the nodes at the same location, as in the master:

```
ubuntu@master:~ $ scp –R build slave1:[install-prefix]
ubuntu@master:~ $ scp –R build slave2:[install-prefix]
```

Create a masters file in the `[install-prefix]/var/mesos/deploy/masters` directory with an editor of your own choice to list the masters one per line, which in our case will be only one:

```
ubuntu@master:~ $ cat [install-prefix]/var/mesos/deploy/masters
master
```

Similarly, the `slaves` file will list all the nodes that we want to be Mesos slaves:

```
ubuntu@master:~ $ cat [install-prefix]/var/mesos/deploy/slaves
slave1
slave2
```

Now, we can start the cluster with the `mesos-start-cluster` script and use `mesos-stop-cluster` to stop it:

```
ubuntu@master:~ $ mesos-start-cluster.sh
```

This, in turn, calls `mesos-start-masters` and `mesos-start-slaves` that will start the

appropriate processes on the master and slave nodes. The script looks for any environment configurations in `[install-prefix]/var/mesos/deploy/mesos-deploy-env.sh`. Also, for better configuration management, the master and slave configuration options can be specified in separate files in `[install-prefix]/var/mesos/deploy/mesos-master-env.sh` and `[install-prefix]/var/mesos/deploy/mesos-slave-env.sh`.

Mesos cluster on Amazon EC2

The Amazon **Elastic Compute Cloud** (**EC2**) provides access to compute the capacity in a pay-as-you-go model through virtual machines and is an excellent way of trying out Mesos. Mesos provides scripts to create Mesos clusters of various configurations on EC2. The `mesos-ec2` script located in the `ec2` directory allows launching, running jobs, and tearing down the Mesos clusters. Note that we can use this script even without building Mesos, but you will need Python (>=2.6). We can manage multiple clusters using different names.

We will need an AWS keypair to use the `ec2` script, and our access and secret key. We have to make our keys available via an environment variable. Create and download a keypair via the AWS Management Console (https://console.aws.amazon.com/console/home) and give them 600 permissions:

```
ubuntu@local:~ $ chmod 600 my-aws-key.pem
ubuntu@local:~ $ export AWS_ACCESS_KEY_ID=<your-access-key>
ubuntu@local:~ $ export AWS_SECRET_ACCESS_KEY=<your-secret-key>
```

Now we can use the EC2 scripts provided with Mesos to launch a new cluster using the following command:

```
ubuntu@local:~/mesos/ec2 $ ./mesos-ec2 -k <your-key-pair> -i <your-identity-file> -s 3 launch ec2-test
```

This will launch a cluster named `ec2-test` with three slaves. Once the scripts are done, it will also print the Mesos web UI link, in the form of `<master-hostname>:8080`. We can confirm that the cluster is up by going to the web interface. The script provides a number of options, a few of which are listed in the following table. We can list all the available options of the script by running `mesos-ec2 --help`:

Command	Use
`--slave or -s`	This is the number of slaves in the cluster
`--key-pair or -k`	This is the SSH keypair for authentication
`--identity-file or -i`	This is the SSH identity file used for logging into the instances

`--instance-type or -t`	This is a slave instance type, must be 64-bit
`--ebs-vol-size`	This is the size of an EBS volume used to store the persistent HDFS data.
`--master-instance-type or -m`	This is a master instance type, must be 64-bit
`--zone or -z`	This is the Amazon availability zone for launching instances
`--resume`	This flag resumes the installation from the previous run

We can use the login action to log in to the launched cluster by providing a cluster name, as follows:

```
ubuntu@local:~/mesos/ec2 $ ./mesos-ec2 -k <your-key-pair> -i <your-identity-file> login ec2-test
```

The script also sets up a HDFS instance that can be used via commands in the `/root/ephemeral-hdfs/` directory.

Finally, we can terminate a cluster using the following command. Be sure to copy any important data before terminating the cluster:

```
ubuntu@local:~/mesos/ec2 $ ./mesos-ec2 destroy ec2-test
```

The script also supports advance functionalities, such as pausing and restarting clusters with EBS-backed instances. The Mesos documentation is a great source of information for any clarification. It is worth mentioning that Mesosphere (http://mesosphere.com) also provides you with an easy way of creating an elastic Mesos cluster on Amazon EC2, Google Cloud, and other platforms and provides commercial support for Mesos.

Running Mesos using Vagrant

Vagrant provides an excellent way of creating portable virtual environments and thus provides an easy way to try Mesos running in a virtual machine. We will see how to create a single-node and multi-node Mesos cluster on virtual machines using Vagrant:

1. Download and install Vagrant from https://www.vagrantup.com/downloads.html. Vagrant works on all the major operating systems.
2. This Vagrant setup uses additional Vagrant plugins. Install them using the following command:

```
ubuntu@local:~ $ vagrant plugin install vagrant-omnibus
vagrant-berkshelf vagrant-hosts vagrant-cachier vagrant-aws
```

3. Download Vagrant configuration from https://github.com/everpeace/vagrant-mesos/ or clone them using `git` and `cd` to the directory:

```
ubuntu@local:~ $ git clone https://github.com/everpeace/vagrant-
mesos.git ; cd vagrant-mesos
```

4. For a single-node cluster setup, `cd` to the `standalone` directory and run the `vagrant up` command. This will create one virtual machine that will run the Mesos master, slave, and ZooKeeper instances. The Mesos UI will be available at `http://192.168.33.10:5050`:

```
ubuntu@local:~ $ cd standalone ; vagrant up
```

5. For a multi-node setup, `cd` to the `mutlinode` directory. We can configure how many virtual machines can be created for the Mesos masters, slaves, and ZooKeeper instances in the `cluster.yml` file. By default, it will create five virtual machines that run as one ZooKeeper, two Mesos masters, and two Mesos slave instances. The Mesos web UI in multi-node setup will be available at `http://172.31.1.11:5050`:

```
ubuntu@local:~ $ cd multinode ; vagrant up
```

6. The Mesos cluster should be up and running. We can log in to these machines via `ssh` using `vagrant ssh`. A single-node setup assigns them the `master` and `slave` as hostnames, while a multi-node setup names the hosts as `master1`, `slave1`, and so on:

```
ubuntu@local:~ $ vagrant ssh master # to login to master
ubuntu@local:~ $ vagrant ssh slave # to login to slave
```

7. We can bring down the virtual machines using the `halt` command. This allows the virtual machines to be booted again with everything set up using the `up` command. Finally, the `destroy` command will destroy all the virtual machines created by Vagrant. Note that we have to execute the vagrant `destroy` commands from the `standalone` or `multinode` directory accordingly:

```
ubuntu@local:~ $ vagrant halt
ubuntu@local:~ $ vagrant destroy
```

This Vagrant setup also allows many different configurations and also supports you to launch the Mesos cluster on Amazon EC2. The vagrant files and the `README` file included in the repository will provide you with more details.

The Mesos community

Despite being a relatively young project, Mesos has a great community (http://mesos.apache.org/community/). There are a number of success stories of using Mesos by both small and large companies (http://mesos.apache.org/documentation/latest/powered-by-mesos/). Companies use Mesos for use cases, ranging from data analytics to web serving to data storing frameworks.

Case studies

Mesos is used by a number of companies in production to simplify infrastructure management. Here, we will see how some of the companies leverage Mesos.

Twitter

Twitter was the first adopter of Mesos and helped to mature the project during the Apache incubation. Twitter is a real-time conversation social platform. Twitter solved the famous fail whale problem, thanks to the reliability of the infrastructure. Twitter considers Mesos as its base for the entire infrastructure and runs a variety of jobs on the Mesos platform, including analytics, ad platform, typeahead service, and messaging infrastructure. All the new services built at Twitter use Mesos, and more importantly, it has changed the way developers think about resources in distributed environments. Developers now can think in terms of a shared pool of resources instead of thinking about individual machines. Twitter also built the Aurora scheduler framework to manage the long-running services on Mesos.

HubSpot

HubSpot makes inbound marketing products. HubSpot runs Mesos on Amazon EC2 to support more than 150 different types of services. Mesos improved resource utilization and ensured high availability without running multiple copies of services, leading to lower infrastructure costs. HubSpot noted that with Mesos, developers are able to launch new services much faster and scaling services have become much more reliable and easier to scale. HubSpot created the Singularity framework on Mesos and built **Platform-as-a-Service** (**PaaS**) to facilitate standardized deployment of services.

Airbnb

Airbnb is a community-driven rental company and was one of the early adopters of Mesos. Airbnb uses Mesos for running data analysis using Hadoop, Spark, Kafka as well as services, such as Cassandra and Rails. Airbnb also created the Chronos scheduler framework for Mesos. We will learn in detail about Aurora and Chronos in Chapter 5, *Running Services on Mesos*.

Twitter's stack was built on Ruby on Rails and JBoss-esque frameworks, which are mostly service-based in nature, while Airbnb, on the other hand, used Mesos more for data processing and is ETL in nature. Twitter runs Mesos on bare metal using Solaris Zones in a private infrastructure, while Airbnb runs it on top of virtual machines using

VMware and Xen hypervisor on AWS. These validate that Mesos provides general and easy to use API as a kernel of modern distributed infrastructure that can run on a wide range of hardware choices and serves a variety of frameworks on top.

Mailing lists

Mesos maintains very accessible documentation at http://mesos.apache.org/documentation/latest/, detailing most parts of Mesos. When the documentation is not sufficient, the Mesos mailing lists provide an excellent medium to interact with other members and are an essential part of the Mesos community. The user mailing list (<user@mesos.apache.org>) and developer mailing list (<dev@mesos.apache.org>) actively discuss the development and usage of Mesos.

Summary

In this chapter, we gave an overview of the requirements of a modern cluster management framework and demonstrated how to set up Mesos clusters. We are ready to run various frameworks on Mesos, which is where we will turn to in the chapters to follow. We will start with Hadoop framework on Mesos in the next chapter.

Chapter 2. Running Hadoop on Mesos

Apache Hadoop is the most influential distributed data processing framework. Hadoop has demonstrated how to do large-scale data processing over commodity hardware. In this chapter, we will walk you through the steps to run Hadoop clusters on Mesos. We will cover the following topic in this chapter:

- Introduction to Hadoop
- Hadoop on Mesos
- Installing Hadoop on Mesos
- An example Hadoop job
- Advanced configuration for Hadoop on Mesos

An introduction to Hadoop

Apache Hadoop (https://hadoop.apache.org) started out of a large-scale search engine project, Nutch. Hadoop is an implementation of MapReduce paradigm popularized by Google's MapReduce paper. As the success of the project indicates, the MapReduce model of computation is applicable to many real-world scenarios.

The Hadoop project mainly has two parts: Hadoop MapReduce and **Hadoop Distributed File System** (**HDFS**). HDFS provides a scalable, fault-tolerant distributed filesystem on commodity hardware. HDFS consists of one or more Namenodes and multiple Datanodes. A Hadoop Namenode represents the master of a **Distributed File System** (**DFS**) and is responsible for storing all the metadata of the filesystem. A Hadoop Datanode represents the slave of a DFS and stores the actual data. Hadoop uses replication for fault tolerance and throughput. HDFS can also be used independently and by many organizations as a DFS with Mesos.

Hadoop MapReduce is the distributed execution engine. A Hadoop job refers to the user program and is divided into many tasks. Each task can be a Map or a Reduce task and executes on a slice of data. Hadoop MapReduce includes a JobTracker and multiple TaskTrackers. A JobTracker is a component that is responsible for coordinating jobs in the Hadoop framework. It acts as a master and schedules tasks on TaskTrackers. Hadoop TaskTrackers are slaves that execute tasks on behalf of applications.

We gave a short overview of Hadoop. For detailed information on Hadoop, you can refer to The Definitive Guide by Tom White or the official Hadoop documentation at http://hadoop.apache.org.

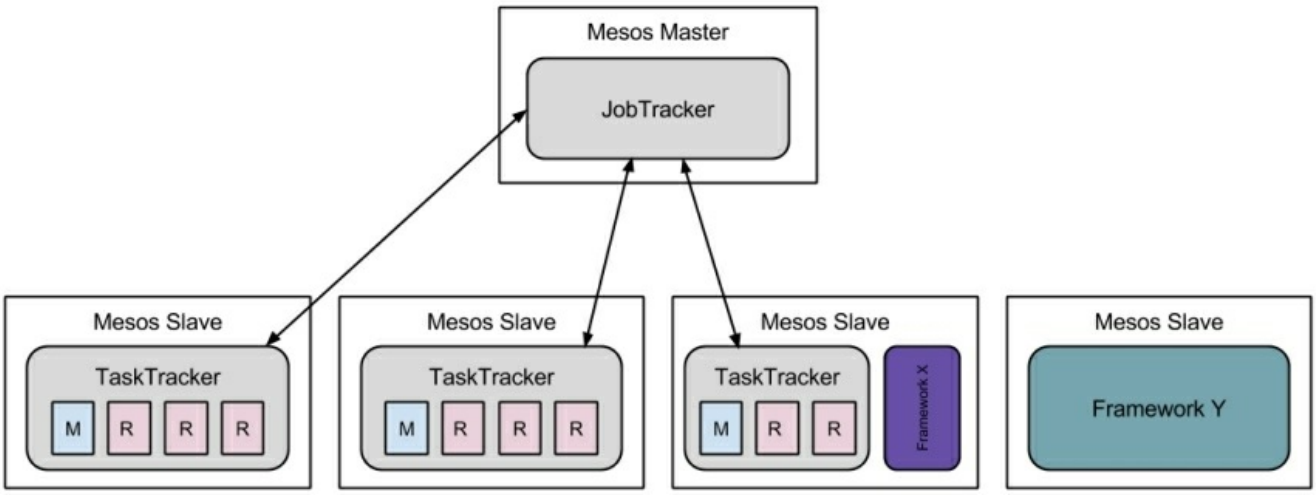
Recently, Hadoop MapReduce project is also divided into two main components: Hadoop YARN and Hadoop MapReduce, where YARN handles the resource allocation for various frameworks, including MapReduce. Hadoop on Mesos currently does not support MapReduce Version 2 (MRv2) or YARN. Traditional Hadoop is now referred to as MapReduce1 or MRv1. The Apache incubator project Myriad (https://github.com/mesos/myriad) will allow you to run elastic YARN clusters on Mesos. Running YARN on Mesos will allow us to share resources between applications running on YARN and Mesos. The Myriad project was started by eBay and at the time of writing, it is in heavy development. Also, there is an active community effort to port standalone HDFS as a Mesos framework (https://github.com/mesosphere/hdfs). This will make HDFS resilient in case of a

failure and HDFS super available while sharing resources with other applications, and it also simplifies the operational aspects.

Hadoop on Mesos

Running Hadoop on Mesos allows you to share cluster resources with other frameworks. By running Hadoop on Mesos, we can leverage the huge community and frameworks built on top of Hadoop, such as Giraph, Hama, HBase, Hive, and so on. Also, if we are already using Hadoop, by running it on Mesos, we can continue using tools built around the Hadoop ecosystem, but with improved resource utilization. Existing MapReduce code and tools will continue working with Hadoop on Mesos. With Hadoop on Mesos, we can also run multiple versions of Hadoop on the same Mesos cluster resources. Hadoop on the Mesos project implements a Mesos framework scheduler around Hadoop's JobTracker and wraps Hadoop executors to Mesos executors. The following figure shows you how Hadoop and Mesos integrate.

In the following figure, the master node is running JobTracker on the Mesos master node and some Mesos slaves are running TaskTrackers. Each TaskTracker is running some Map and Reduce task slots. Note that Hadoop on Mesos is sharing resources with other frameworks (X and Y in the figure).



Hadoop MapReduce processes running on Mesos

Installing Hadoop on Mesos

Hadoop on Mesos (https://github.com/mesos/hadoop) relies on the extension to Mesos, such as the Mesos executor to execute TaskTrackers and Hadoop JobTracker Mesos Scheduler to run Hadoop on a Mesos framework. We will run Hadoop 1.x on Mesos:

1. Install and run Mesos by following the instructions in Chapter 1, *Running Mesos*.
2. We need to compile Hadoop on the Mesos library. Hadoop on Mesos uses Maven to manage dependencies, which we will need to install along with Java and Git:

   ```
   ubuntu@master:~ $ sudo apt-get install maven openjdk-7-jdk git
   ```

3. Let's clone Hadoop on the Mesos source code from https://github.com/mesos/hadoop, and navigate to it using the following command:

   ```
   ubuntu@master:~ $ git clone https://github.com/mesos/hadoop/
   ubuntu@master:~ $ cd hadoop
   ```

4. Build Hadoop on the Mesos binaries from the code using the following command. By default, it will build the latest version of Mesos and Hadoop. If required, we can adjust the versions in the `pom.xml` file:

   ```
   ubuntu@master:~ $ mvn package
   ```

   This will build `hadoop-mesos-VERSION-jar` in the `target` folder.
5. Download the Hadoop distribution, extract, and navigate to it. We can use vanilla Apache distribution, **Cloudera Distribution Hadoop** (**CDH**), or any other Hadoop distribution. We can download and extract the latest CDH distribution using following commands:

   ```
   ubuntu@master:~ $ wget
   http://archive.cloudera.com/cdh5/cdh/5/hadoop-2.5.0-
   cdh5.2.0.tar.gz
   ubuntu@master:~ $ tar xzf hadoop-*.tar.gz
   ```

6. We need to put Hadoop on Mesos jar that we just built in a location where it's accessible to Hadoop via Hadoop CLASSPATH. We will copy it to the Hadoop `lib` folder, which is by default the `lib` folder in `share/hadoop/common/ inside` hadoop distribution:

   ```
   ubuntu@master:~ $ cp hadoop-mesos/target/hadoop-mesos-*.jar
   hadoop-*/share/hadoop/common/lib
   ```

7. By default, the CDH distribution is configured to use MapReduce Version 2

(MRv2) with YARN. So, we need to update it to point to MRv1:

```
ubuntu@master:~ $ cd hadoop-*

ubuntu@master:~ $ mv bin bin-mapreduce2
ubuntu@master:~ $ ln -s bin-mapreduce1 bin

ubuntu@master:~ $ cd etc;
ubuntu@master:~ $ mv hadoop hadoop-mapreduce2
ubuntu@master:~ $ ln -s hadoop-mapreduce1 hadoop
ubuntu@master:~ $ cd -;
```

Optionally, we can also update examples to point to the MRv1 examples:

```
ubuntu@master:~ $ mv examples examples-mapreduce2
ubuntu@master:~ $ ln -s example-mapreduce1 examples
```

8. Now, we will configure Hadoop to recognize that it should use the Mesos scheduler that we just built. Set the following mandatory configuration options in `etc/hadoop/mapred-site.xml` by adding them to the `<configuration>` and `</configuration>` tags:

```
<property>
    <name>mapred.job.tracker</name>
    <value>localhost:9001</value>
</property>
<property>
    <name>mapred.jobtracker.taskScheduler</name>
    <value>org.apache.hadoop.mapred.MesosScheduler</value>
</property>
<property>
    <name>mapred.mesos.taskScheduler</name>
    <value>org.apache.hadoop.mapred.JobQueueTaskScheduler
    </value>
</property>
<property>
    <name>mapred.mesos.master</name>
    <value>zk://localhost:2181/mesos</value>
</property>
<property>
    <name>mapred.mesos.executor.uri</name>
    <value>hdfs://localhost:9000/hadoop.tar.gz</value>
</property>
```

We specify Hadoop to use Mesos for scheduling tasks by specifying the `mapred.jobtracker.taskScheduler` property. The Mesos master address is specified via `mapred.mesos.master`, which we have set to the local ZooKeeper

address. `mapred.mesos.executor.uri` points to the Hadoop distribution path that we will upload to HDFS, which is to be used for executing tasks.

9. We have to ensure that Hadoop on Mesos is able to find the Mesos native library, which by default is located at `/usr/local/lib/libmesos.so`. We need to export the location of the Mesos native library by adding the following line at the start of the `bin/hadoop-daemon.sh` script:

   ```
   export MESOS_NATIVE_LIBRARY=/usr/local/lib/libmesos.so
   ```

10. We need to have a location where the distribution can be accessed by Mesos, while launching Hadoop tasks. We can put it in HDFS, S3, or any other accessible location, such as a NFS server. We will put it in HDFS, and for this, we need to install HDFS on the cluster. We need to start the Namenode daemon on the HDFS master node. Note that HDFS master node is independent of the Mesos master. Copy the Hadoop distribution to the node, and start the Namenode using the following command:

    ```
    ubuntu@master:~$ bin/hadoop-daemon.sh start namenode
    ```

    We need to start the Datanode daemons on each node that we want to make a HDFS slave node (which is independent of the Mesos slave node). Copy the created Hadoop distribution to all HDFS slave nodes, and start the Datanode on each using the following command:

    ```
    ubuntu@master:~$ bin/hadoop-daemon.sh start datanode
    ```

    We need to format the Namenode for the first usage with the following command on the HDFS master, where the Namenode is running:

    ```
    ubuntu@master:~$ bin/hadoop namenode -format
    ```

11. Hadoop is now ready to run on Mesos. Let's package it and upload it on HDFS:

    ```
    ubuntu@master:~ $ tar cfz hadoop.tar.gz hadoop-*
    ubuntu@master:~ $ bin/hadoop dfs -put hadoop.tar.gz
    /hadoop.tar.gz
    ubuntu@master:~ $ bin/hadoop dfs -chmod 777 /hadoop.tar.gz
    ```

12. Now, we can start the JobTracker. Note that we don't need to start TaskTracker manually, as this will be started by Mesos when we submit a Hadoop job:

    ```
    ubuntu@master:~ $ bin/hadoop jobtracker
    ```

Hadoop is now running, and we are ready to run Hadoop jobs.

An example Hadoop job

Let's run an example job that counts the frequency of each word across files:

1. We need to input some text to run the example. We will download *Alice's Adventures in Wonderland* from Project Gutenberg's site:

   ```
   ubuntu@master:~ $ wget http://www.gutenberg.org/files/11/11.txt -
   O /tmp/alice.txt
   ```

2. Create an `input` directory on HDFS and put the downloaded file into it:

   ```
   ubuntu@master:~ $ bin/hadoop dfs –mkdir input
   ubuntu@master:~ $ bin/hadoop dfs –put /temp/alice.txt input
   ```

3. Run the Hadoop WordCount example from the `hadoop-examples jar` file, which is part of the Hadoop distribution:

   ```
   ubuntu@master:~ $ bin/hadoop jar hadoop-examples.jar wordcount
   input output
   ```

4. The output of the program will be in the `output` directory of HDFS. We can see the output using the following command:

   ```
   ubuntu@master:~ $ bin/hadoop dfs –cat output/*
   ```

The output will have words with the corresponding frequencies on each line.

Advanced configuration for Hadoop on Mesos

Hadoop has many parameters that affect its performance and fault tolerance behavior. You may want to tune the following options suitable for your cluster and workload.

Task resource allocation

This is used to control the amount of resources allocated to Hadoop tasks. By setting the following parameters in `conf/core-site.xml`, Mesos will be instructed to assign an appropriate CPU, memory (in MB), and diskspace (in MB) resources.

Note

Note that the memory includes the JVM overhead (~10%).

```
<property>
    <name>mapred.mesos.slot.cpus</name>
    <value>1</value>
</property>
<property>
    <name>mapred.mesos.slot.mem</name>
    <value>1024</value>
</property>
<property>
    <name>mapred.mesos.slot.disk</name>
    <value>1024</value>
</property>
```

The following parameters control the resource policies of Mesos:

- `mapred.mesos.total.map.slots.minimum` and `mapred.mesos.total.reduce.slots.minimum`: These parameters specify the minimum number of Map and Reduce slots that Mesos will try to acquire at a given point of time. These are used as hints and do not guarantee that these slots will be available. Similarly, `mapred.tasktracker.map.tasks.maximum` and `mapred.tasktracker.reduce.tasks.maximum` control the maximum number of such slots:

  ```
  <property>
      <name>mapred.mesos.total.map.slots.minimum</name>
      <value>0</value>
  </property>
  <property>
      <name>mapred.mesos.total.reduce.slots.minimum</name>
      <value>0</value>
  </property>
  <property>
      <name>mapred.tasktracker.map.tasks.maximum</name>
      <value>50</value>
  </property>
  ```

```
<property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>50</value>
</property>
```

- If we want Mesos to always allocate a fixed number of slots per TaskTracker based on the maximum Map/Reduce slot parameters and reject other offers, you can set `mapred.mesos.scheduler.policy.fixed` to true:

```
<property>
    <name>mapred.mesos.scheduler.policy.fixed</name>
    <value>false</value>
</property>
```

Note

There is an earlier version that supports more elastic scaling of TaskTrackers by monitoring idle TaskTrackers. With this, TaskTrackers, which are idle for some time, will be killed and the resources can be used by other frameworks on the Mesos cluster. There are two configuration parameters that control this behavior:

`mapred.mesos.tracker.idle.interval` controls how often to check for the TaskTracker utilization. The interval is specified in a number of seconds and it defaults to 5 seconds.

`mapred.mesos.tracker.idle.checks` specifies how many successful idle checks can be done before killing the TaskTracker. This defaults to five, which means that TaskTracker will be killed after five idle checks succeed.

Metrics reporting

Monitoring is important for any large-scale cluster. Mesos supports metrics reporting from Hadoop to the external system. It uses Coda Hale Metrics library (http://metrics.codahale.com/) for metrics. To enable metrics, set the following in `conf/core-site.xml`. By default, it's disabled:

```
<property>
    <name>mapred.mesos.metrics.enabled</name>
    <value>true</value>
</property>
```

Metrics can be reported to a variety of systems. The following are some of the popular examples.

CSV

To enable **Comma Separated Values** (**CSV**) reporting, set the following with the appropriate path. A CSV file is created for reporting metrics on the path specified by `mapred.mesos.metrics.csv.path`, while `mapred .mesos.metrics.csv.interval` controls the frequency of the reporting:

```
<property>
    <name>mapred.mesos.metrics.csv.enabled</name>
    <value>true</value
</property>
<property>
    <name>mapred.mesos.metrics.csv.path</name>
    <value>/path/to/metrics/csv/metrics.csv</value>
</property>
<property>
    <name>mapred.mesos.metrics.csv.interval</name>
    <value>60</value>
</property>
```

Graphite

Graphite (http://graphite.wikidot.com) is a popular open source tool used for monitoring and graphing the performance data. If we want to monitor the cluster, Graphite is one of the best real-time graphing frameworks:

```
<property>
    <name>mapred.mesos.metrics.graphite.enabled</name>
    <value>true</value
```

```
</property>
<property>
    <name>mapred.mesos.metrics.graphite.host</name>
    <value>your_graphite_host_name</value>
</property>
<property>
    <name>mapred.mesos.metrics.graphite.port</name>
    <value>your_graphite_port</value>
</property>
<property>
    <name>mapred.mesos.metrics.graphite.prefix</name>
    <value>your_graphite_prefix</value>
</property>
<property>
    <name>mapred.mesos.metrics.graphite.interval</name>
    <value>60</value>
</property>
```

Cassandra

If we have another serving system for metrics, or if we want to store the metrics for later analysis, Cassandra is a really good option:

```
<property>
    <name>mapred.mesos.metrics.cassandra.enabled</name>
    <value>true</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.hosts</name>
    <value>your_cassandra_host</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.port</name>
    <value>your_cassandra_port(normally 9042)</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.interval</name>
    <value>60</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.prefix</name>
    <value>your_cassandra_prefix</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.ttl</name>
    <value>864000</value>
</property>
<property>
```

```
    <name>mapred.mesos.metrics.cassandra.keyspace</name>
    <value>metrics</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.table</name>
    <value>metrics</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.consistency</name>
    <value>QUORUM</value>
</property>
```

Authentication

Hadoop on Mesos supports the following configuration parameters for authentication.

To get the Mesos reservation for different frameworks, roles are used. To configure this, set `mapred.mesos.role` to appropriate roles. The default value is *, indicating equal access to the resources by all frameworks:

```
<property>
    <name>mapred.mesos.role</name>
    <value>*</value>
</property>
```

It controls whether Hadoop on Mesos accepts resource offers from other roles. If set to true, the framework will only accept resource offers that are configured in `mapred.mesos.role`:

```
<property>
    <name>mapred.mesos.role.strict</name>
    <value>false</value>
</property>
```

`mapred.mesos.framework.principal` specifies the Mesos framework principal used for authentication:

```
<property>
    <name>mapred.mesos.framework.principal</name>
    <value>hadoop</value>
</property>
```

`mapred.mesos.framework.secretfile` specifies the path to the file that holds the Mesos framework secret, which is also used as part of the authentication:

```
<property>
    <name>mapred.mesos.framework.secretfile</name>
    <value>/location/secretfile</value>
</property>
```

`mapred.mesos.framework.user` specifies the user on which the framework runs. The default is set to the user running the framework scheduler:

```
<property>
    <name>mapred.mesos.framework.user</name>
    <value>hadoop</value>
</property>
```

Note that we will discuss these parameters and terms in detail in the later chapters, but they are listed here for completeness.

Container isolation

JobTracker of Hadoop on Mesos can send a custom container image to be used for isolating a task on Mesos via JobConf. It's a very useful feature to isolate Hadoop software dependencies from the rest of Mesos on slaves. This feature is available starting from Mesos Version 0.19. This is optional and Hadoop on Mesos will work irrespective of whether we use container isolation or not. In this case, we do not set the following options, JobConf (from Hadoop) will not pass any `ContainerInfo` to Mesos.

Hadoop on the Mesos supports the following two configuration options to control the container. The first one controls which container image is to be used, and the second one supplies any parameters it requires:

```
<property>
    <name>mapred.mesos.container.image</name>
    <value>docker:///ubuntu</value>
</property>
<property>
    <name>mapred.mesos.container.options</name>
    <value>-v,/foo/bar:/bar</value>
</property>
```

The container image that we use should have the Mesos native library already installed on it.

Additional configuration parameters

Hadoop can leverage high-availability of Mesos, and by enabling checkpointing, it can survive restarts. We will discuss the details of slave recovery in later chapters. To enable the checkpoint feature of Hadoop on Mesos, set `mapred.mesos.checkpoint` to `true`:

```
<property>
    <name>mapred.mesos.checkpoint</name>
    <value>true</value>
</property>
```

`mapred.mesos.framework.name` specifies the name of the framework. By default, it is set to `hadoop`:

```
<property>
    <name>mapred.mesos.framework.name</name>
    <value>hadoop</value>
</property>
```

Also, there are possibilities to improve how Hadoop currently runs on Mesos. For example, sending multiple task requests to one TaskTracker instead of launching one TaskTracker for each executor. These changes are discussed within the community. While they are not yet available at the time of writing, it might be available soon.

Summary

In this chapter, we had an overview of Hadoop and saw how you can run a Hadoop cluster on Mesos. We also looked at some important configuration parameters for performance tuning and operational aspects.

In the next chapter, we will see how to do data analytics on Mesos using Spark.

Chapter 3. Running Spark on Mesos

Spark is a fast and general execution engine for large-scale data processing. One of the prime uses of large-scale clusters is running data processing jobs. Spark provides data processing of many forms as part of the **Berkeley Data Analytics Stack** (**BDAS**). Spark supports batch processing, iterative processing, near real-time processing, and stream processing. In this chapter, we will walk you through the steps of setting up Spark on the Mesos cluster:

- Introducing Spark
- Spark job scheduling
- Spark Standalone mode
- Spark on Mesos
- Tuning Spark on Mesos

Introducing Spark

Apache Spark is a fast and scalable general data processing framework (http://spark.apache.org). Spark provides a very concise syntax for writing a wide range of data processing applications. Spark became a top-level Apache project in early 2014.

The Spark project was started at Berkeley as part of the Berkeley Data Analytics Stack (https://amplab.cs.berkeley.edu/software), the same project that Mesos comes from. Spark was the first data processing framework built on Mesos and effectively leverages Mesos for resource management. Spark is one of the fastest growing data analysis frameworks and aims to unify all kinds of data analysis under a single unified API. Spark provides a unified API for doing batch, streaming, and iterative data processing.

Spark makes an aggressive use of memory to accelerate computations. Spark's **Directed Acyclic Graph** (**DAG**) execution engine is suitable for a wide range of applications, including the interactive and iterative algorithms that often arise in machine learning, real-time analysis, and graph processing. Spark's engine is quite generalized and supports various high-level tools, including, but not limited to, Spark SQL (https://spark.apache.org/sql) for data warehouse systems, Spark Streaming (https://spark.apache.org/streaming) for stream processing applications, MLlib (https://spark.apache.org/mllib) for scalable machine applications, and GraphX (https://spark.apache.org/graphx) for graph processing. Spark integrates with a wide range of data sources. Spark on Mesos is being used by a number of companies to gain insight from the data.

Spark is built around the abstraction of **Resilient Distributed Datasets** (**RDD**) (http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf). Researchers behind RDD describe RDD as *a read-only partitioned collection of records*. RDD is a distributed data collection, which supports high-level parallel operations. RDD is a distributed memory abstraction that allows in-memory computation in a distributed manner, while still providing fault tolerance. RDDs are a restricted form of shared memory that allow coarse-grained predefined transformation to the shared state. By not allowing fine-grained arbitrary updates to the shared state, RDDs can guarantee fault tolerance without incurring high overheads. By restricting the transformations, RDDs achieve fault-tolerance without sacrificing efficiency. Also, traditional **Distributed Shared Memory** (**DSM**) systems are limited to the amount of memory available, but RDDs seamlessly allow the use of disks as extension. This allows graceful degradation

in performance since RDD reads/writes happen in bulk, unlike DSM reads/writes, which are random.

RDD transformations are expressive enough to support a wide range of applications, as evidenced by the variety of frameworks built on top of Spark. RDD fault tolerance is achieved by storing the lineage (a series of operations applied to the data), rather than actual data after each operation. In case of a fault, RDD can derive it from the data by applying the transformations.

Note

Note that this chain of operations can be quite long and thus, we can checkpoint the RDD to allow a faster recovery in case of a failure. How much check pointing we do is a tradeoff between performance and recovery time.

Spark is written in Scala and supports writing programs in Scala, Java, or Python, using language bindings. The Spark programming guide gives you more details about the API and available transformations (https://spark.apache.org/docs/latest/programming-guide.html). The books, *Learning Spark* and *Fast Data Processing with Spark* cover excellent details.

Spark job scheduling

In this section, we will take a look at how Spark jobs are scheduled on the cluster. Spark's cluster mode refers to how job scheduling and resource management happens across Spark applications (https://spark.apache.org/docs/latest/job-scheduling.html). Spark's cluster manager has two strategies for resource management:

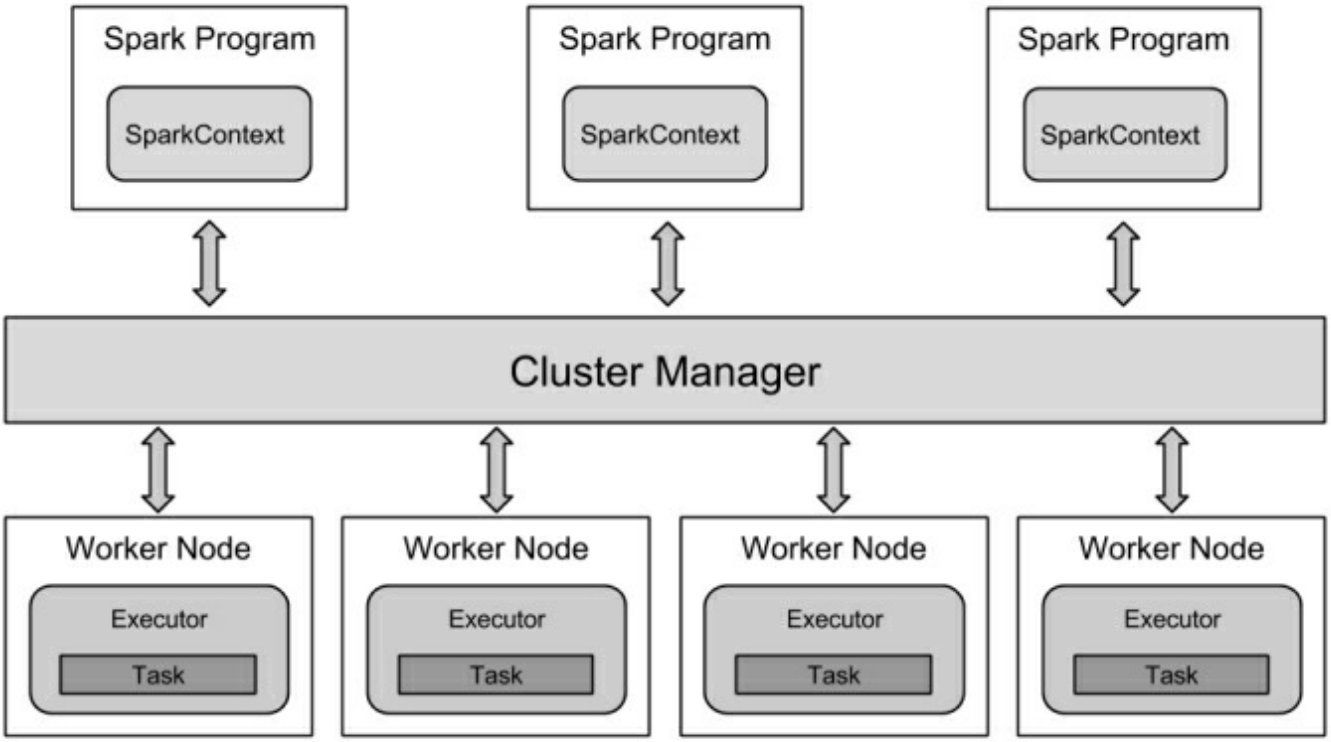
- In static partitioning, each Spark application gets a fixed amount of resources from the cluster. The applications hold on to these resources during the entire execution. This approach is used by both Spark standalone mode and Mesos coarse-grain mode.
- The dynamic sharing of CPU cores. In this mode, each Spark application has a fixed memory allocation, but CPU is allocated only when the Spark application is only running. This strategy is used by Mesos' fine-grain mode. This is useful when we have many idle Spark sessions.

Both the modes have fixed memory sharing and differ only on how CPU is shared.

Tip

Starting from Version 1.2, Spark introduced the dynamic scaling of resources. At the time of writing, dynamic resource scaling is only available in Spark Standalone mode and YARN. The dynamic resource scaling capabilities allow resources to be used more elastically. Future releases of Spark will introduce dynamic resource scaling capabilities on Mesos as well.

The following image describes the various components of Spark:



The Spark architecture

The following are important entities in a Spark cluster as described in, as described in https://spark.apache.org/docs/latest/cluster-overview.html:

- Application is a user program built to run on Spark. Application consists of a driver program and executors. Driver runs the `main()` function and creates the `SparkContext` object. `SparkContext` manages the execution of the Spark application.
- Job refers to the parallel computation triggered by Spark actions. It consists of multiple independent units of work called tasks.
- Cluster manager is a module responsible for resource coordination on the cluster. Currently, Spark supports a standalone cluster manager, YARN, and Mesos as cluster managers.
- Worker node is a node that has resources and is capable of running Spark's application code in the cluster. Worker nodes launch processes called Spark executor to run tasks from applications. Each application has its own executors responsible for managing the task execution as well as data for the task. Spark

acquires worker resources in the form of executors and sends the application code to them. After everything is ready, `SparkContext` issues tasks to run to executors.

Spark Standalone mode

Spark standalone mode uses a simple built-in cluster manager. The Spark standalone mode cluster manager can also be run on a single machine, and it is great way to try out Spark (http://spark.apache.org/docs/latest/spark-standalone.html). In Standalone mode, Spark requires Java and Git installed on the system. Let's install the dependencies. On Ubuntu, the following command will install both Java and Git:

```
ubuntu@master:~$ sudo apt-get install openjdk-7-jdk git
```

The following are the steps to install Spark in standalone mode:

1. Download the latest Spark tarball from http://spark.apache.org/downloads.html. Spark packages are prebuilt compatible with a specific version of Hadoop and Mesos. At the time of writing, the latest version is 1.2, which is compatible with 0.18 version of Mesos and many versions of Hadoop, including the latest version, 2.4.
2. Extract the downloaded tar file and go to the following directory:

   ```
   ubuntu@master:~$ tar –xzf spark-*.tar.gz
   ubuntu@master:~$ cd spark-*
   ```

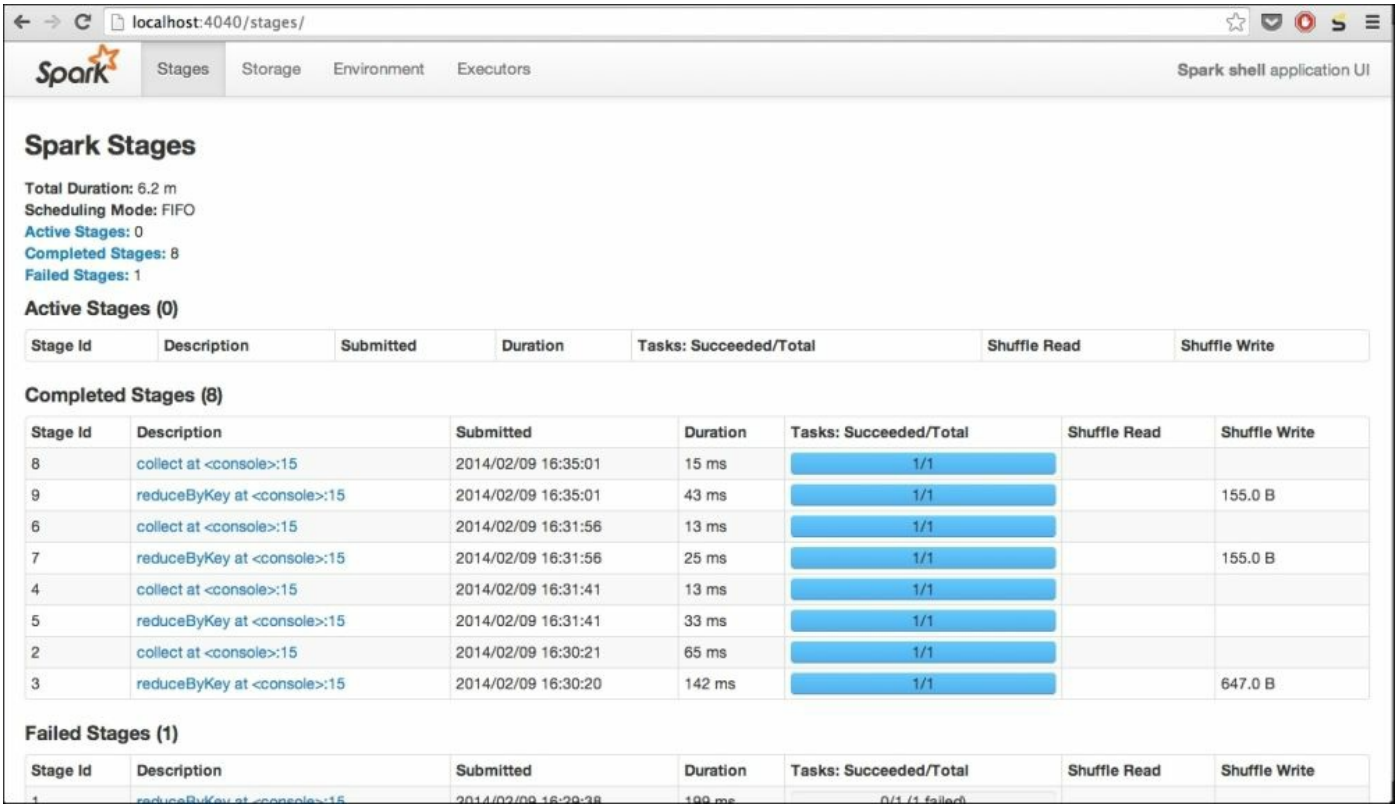
3. Now, we can launch the Spark Scala shell. The Spark shell is a modified version of a Scala shell and is meant to support interactive analysis and early exploration. Similarly, The Python Spark shell can be launched via `./bin/pyspark`. Spark greets us with a fancy shell:

   ```
   ubuntu@master:~ $ ./bin/spark-shell
   ```



Spark also includes a web interface that provides information regarding the state of the application. It can be accessed at `http://<driver-node>:4040`. Port `4040` is the default port and if multiple applications are running, the successive ports (`4041`, `4042`, and so on) are used. The driver node is where the Spark session is running.

Among other things, the Spark shell prints the URL of the Spark web interface at the start of the session, which in the preceding screenshot is `http://10.168.91.107:4040`.



Spark includes many sample Spark programs in the `examples` directory. The `bin/run-example` script runs sample examples that we can use to verify the setup. For example, to run the Spark PI approximation example:

```
ubuntu@master:~ $./bin/run-example org.apache.spark.examples.SparkPi
SLF4J: Class path contains multiple SLF4J bindings.
….
14/02/09 16:44:04 INFO SparkContext: Job finished: reduce at
SparkPi.scala:39, took 8.638717 s
Pi is roughly 3.13832
….
14/02/09 16:44:06 INFO MapOutputTrackerMasterActor:
MapOutputTrackerActor stopped!
```

Once the Spark is installed, we can see the Spark web UI when a Spark session is running. Also, the Mesos web interface will now list Spark in the active framework. We can run other Scala examples as well as examples written in Java or Python that are included with the Spark distribution.

Spark on Mesos

Mesos can act as a cluster manager for Spark. While running Spark on Mesos, Spark leverages all the resource management capabilities of Mesos, and Spark tasks are executed on Mesos worker nodes using the Spark executor. This allows the sharing of resources between multiple instances of Spark or with other frameworks. Let's see how to install Spark on Mesos:

1. Build and run Mesos, as shown in Chapter 1, *Running Mesos*.
2. Download the Spark tar file, which is similar to the steps in the earlier section.
3. The Spark archive containing executors has to be accessible from Mesos. Typically, we can use **Hadoop Distributed File System** (**HDFS**) or Amazon S3. We will use HDFS:

```
ubuntu@master:~ $ hadoop fs -mkdir /tmp
ubuntu@master:~ $ hadoop fs -put spark.tar.gz /tmp
```

4. Create `spark-env.sh` from `spark-env.sh.template`, and add the following three export lines to the file:

```
ubuntu@master:~ $ cp spark-env.sh.template spark-env.sh

ubuntu@master:~ $ vim spark-env.sh
export MESOS_NATIVE_LIBRARY=/usr/local/lib/libmesos.so
export SPARK_EXECUTOR_URI=hdfs://master/tmp/spark.tar.gz
export MASTER=mesos://master:5050
```

   `MESOS_NATIVE_LIBRARY` specifies the location of the `libmesos.so` Mesos library on the slave nodes. By default, it is installed on `/usr/local/lib/`. `SPARK_EXECUTOR_URI` is the location of the Spark distribution available in Mesos to launch as a worker process. `MASTER` specifies the Mesos master address. These are the minimum required parameters. We can also specify other parameters, such as address where Spark master listens (`SPARK_MASTER_IP` and `SPARK_MASTER_PORT`) and Sparker port (`SPARK_WORKER_PORT`), and so on.

Now, Spark is configured to run on the Mesos cluster. We can create `SparkContext` with a configuration, specifying the Mesos master URI and Spark executor path:

```
val conf = new SparkConf()
 .setMaster("mesos://master:5050")
 .set("spark.executor.uri", "hdfs://master/tmp/spark.tar.gz")
```

```
val sc = new SparkContext(conf)
```

Now, if we take a look at the Mesos active frameworks, we should be able to find Spark listed there. We can drill down the details of the active tasks, and the sandbox link provides you with easy access to `stderr` and `stdout`.

Tuning Spark on Mesos

Spark running on Mesos shares resources with other frameworks running on Mesos. It is important to understand the trade-offs of resource sharing and tuning Spark on Mesos for realizing the maximum benefits without sacrificing the application requirements. While running Spark on the Mesos cluster, resource sharing happens in two modes:

- Fine-grain: By default, Spark will run each task as a separate Mesos task. This means that cluster resources are shared with the other frameworks running alongside Spark (or other instances of Spark itself). In this fine-grained mode, Mesos can share cluster resources among all applications in an elastic manner, but this elasticity comes at the cost of a higher overhead when launching tasks. This overhead is not significant for most applications but can be noticeable for applications with stringent timing requirements.

  Tip

  Mesos version below 0.21 had a bug in which the Spark application would not make progress while running in fine-grained mode (https://issues.apache.org/jira/browse/MESOS-1688) with high memory utilization. The workaround for this issue is to make sure that our application executors always leave at least 32 MB free memory on the slaves.

- Coarse-grain: The coarse-grain mode will instead reserve all the resources by launching a long-running Spark task on each Mesos node and will allocate these reserved resources to the Spark job, as requested. Thus, when running in coarse-grained mode, the jobs don't have the overhead of launching tasks on the Mesos nodes and are suitable for applications with low latency requirements, such as interactive querying.

The mode is controlled by the `spark.mesos.coarse` property. We can run Spark in coarse-grained mode by setting it to `true` in `SparkConf`, as follows:

```
conf.set("spark.mesos.coarse", "true")
```

In addition to run modes, there are a few other configuration parameters while running Spark on Mesos:

- `spark.cores.max`: This specifies the maximum number of cores that Spark will acquire. By default, it will acquire all the resources offered by Mesos. We can control how greedy the resource reservation can be, using this property. The

default behavior is not ideal, unless Spark is the only application running on Mesos. We should limit the resources used by Spark by setting `spark.cores.max` to a meaningful value:

```
conf.set("spark.cores.max", "8")
```

This limits the number of cores used by Spark to 8. Similarly, the `spark.executor.memory` property controls the memory usage of executors.

Tip

Configuring an appropriate mode and tuning the values of `spark.cores.max` and `spark.executor.memory` to cap the resources allocated to Spark, for your environment, will have a significant effect on the resource utilization of the cluster and the response time of applications.

- `spark.mesos.extra.cores`: This property controls the extra amount of cores requested per task, while running in coarse-grain mode. By default, it's set to 0. The amount of cores requested will be the number of cores in the offer plus the extra cores, according to this parameter. The total amount of cores requested is still bound by the `spark.cores.max` property.
- `spark.mesos.executor.home`: This specifies where the Spark installation is located on the executors. By default, it's set to the same value as SPARK_HOME on the driver node. This is only used if `spark.executor.uri` is not set.
- `spark.mesos.executor.memoryOverhead`: This specifies the extra memory overhead to be added to the total requested memory of a task (specified by `spark.executor.memory`). By default, its value is set to the executor memory, * 0.07, with a minimum of 384. These values are specified in MiB. At the minimum, there is a hardcoded 7 percent overhead. The final overhead will be a maximum of `spark.mesos.executor.memoryOverhead, 0.07* spark.executor.memory`.

A list of all Spark configuration options can be found in the Spark documentation at http://spark.apache.org/docs/latest/configuration.html.

Summary

In this chapter, we introduced Spark and discussed how to run it in standalone mode. We also set up the Spark cluster on Mesos. In the next chapter, we will see how to build a complex data analysis infrastructure on Mesos.

Chapter 4. Complex Data Analysis on Mesos

In this chapter, we will walk you through the different frameworks for complex data analysis on Mesos. We will set up Storm and Spark Streaming for processing real-time streams of data on Mesos and also see how to run NoSQL database Cassandra on Mesos.

We will cover the following topics:

- Complex data and the rise of the Lambda architecture
- Storm
- Spark Streaming
- NoSQL on Mesos

Complex data and the rise of the Lambda architecture

The explosive growth in big data is not only in terms of the volume being generated but also in terms of the variety and speed at which the results have to be generated in order to be meaningful. Thus, the velocity of the data and computation has forced developers toward real-time stream processing frameworks, and at the same time, the variety and unstructured nature of data has led to the NoSQL movement.

With the rise of the **Internet of Things** (**IoT**), sensors, social media, machine transactions, monitoring data, and so on are being produced at a very large scale and velocity. The insights provided by this data can be very valuable, but the analysis and the data itself do not make sense if results are produced with a delay, or analysis is done on the stale data. In the previous chapters, we looked at how large amounts of data can be processed using Hadoop and Spark. These traditional tools are very well suited for batch or offline analysis, but they are not designed for real-time stream processing or other low-latency applications; for example, SQL-like query processing.

While stream processing is increasingly part of the modern data architecture, there are other components in the modern data architecture. Modern data architecture consists of different components for serving different needs. The Lambda architecture (http://en.wikipedia.org/wiki/Lambda_architecture) is a very popular architecture for designing data architectures. It mainly consists of three layers:

1. Batch layer
2. Speed layer
3. Serving layer

Running the Lambda architecture components on Mesos allows not only the sharing of resources, but also helps make them fault tolerant. In the batch layer, the idea is to derive insights by processing all the data collected over time; for example, building predictive models. We have already seen how to use Hadoop and Spark for data processing on Mesos in the earlier chapters. Apache Hama (https://hama.apache.org) is another framework in the batch layer. It is a general **Bulk Synchronous Processing** (**BSP**) that is useful for graph processing and matrix computations.

With the variety and the velocity of the new data, it is no longer sufficient to only rely

on offline models and process data offline periodically. In the speed layer, the idea is to process the data as it arrives. This layer mainly deals with stream processing and is also known as **Complex Event Processing** (**CEP**) or real-time processing. Mesos supports Apache Samza, Apache Storm, and Spark Streaming frameworks for implementing the speed layer. Apache Samza (http://samza.apache.org) is a stream processing framework built on top of Apache Kafka framework. There is an ongoing effort to port Samza on Mesos (https://github.com/Banno/samza-mesos). We will discuss Apache Storm and Spark Streaming in the next section. Note that different stream processing frameworks have different architectures, reflecting different tradeoffs in terms of speed, supported operations, consistency semantics, and scalability.

The serving layer is responsible for storing the outputs of the batch and speed layer and serving queries based on them. Mesos provides increasingly more options to build a scalable layer for storing and serving data:

- **HDFS** (**Hadoop Distributed File System**) provides a distributed file system running on commodity hardware, as we mentioned in Chapter 2, *Running Hadoop on Mesos*. There is an ongoing effort for running HDFS on Mesos to provide highly available HDFS.
- Tachyon (http://www.tachyonproject.org) is a memory-centric storage system. There is a prototype version of Tachyon on Mesos at https://github.com/mesosphere/tachyon-mesos.
- Riak (https://github.com/basho/riak) is a decentralized key value store. There is an ongoing effort to make Riak work on Mesos (https://github.com/edpaget/riak-mesos).
- Elasticsearch (https://elasticsearch.org) is a distributed full text search engine. Elasticsearch runs on Mesos (https://github.com/mesosphere/elasticsearch-mesos).
- Apache Cassandra (http://cassandra.apache.org) is a NoSQL database that we will see how to run on Mesos.

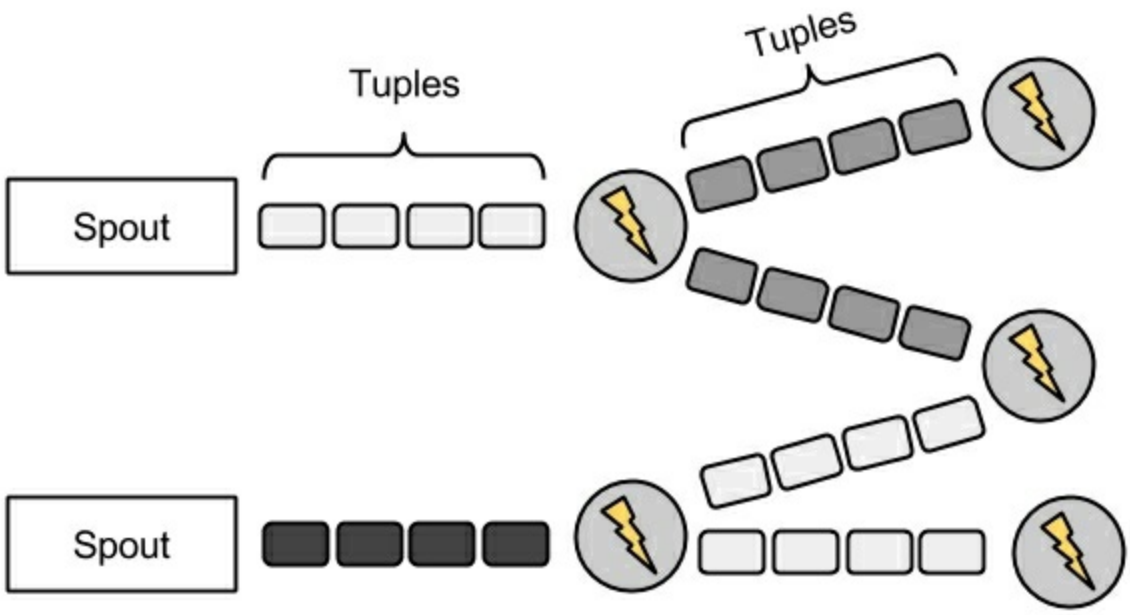
Apart from these three layers, we also need connectors between these layers to ingest data and exhaust the other layers. Apache Kafka (http://kafka.apache.org) is a distributed publish-subscribe (as known as pub/sub) messaging system. Pub/sub systems form the backbone of modern data architectures. They connect different data processing frameworks in a loosely coupled manner and can be used for a number of use cases. There is an active effort to try to port Kafka on Mesos (https://github.com/stealthly/kafka-mesos). Also, as more frameworks leverage the upcoming support for persistent storage in Mesos (https://issues.apache.org/jira/browse/MESOS-1554), complex data processing will be

even better adopted on Mesos. This is by no means a complete set of tools when it comes to complex data analysis, and Mesos supports a number of other frameworks for addressing the various needs of complex data processing. At the time of writing, many of these projects are in an early phase, but are making substantial progress.

Storm

Apache Storm is a real-time distributed stream event-processing engine (https://storm.apache.org). Storm has transactional, reliable, scalable, fault-tolerant properties and provides simple-to-use API. It has a very different architecture compared to MapReduce. MapReduce systems, such as Hadoop, Spark, and so on, move the code near the data. This means that in MapReduce, each node has some data, and an identical code is given to each node that produces results. While in Storm, each node implements different kinds of processing and data flows through these nodes.

The main abstraction of Storm is of a stream (of tuples) flowing through nodes, each one performing some transformations. Tuples are very generic and can contain a list of any type of serializable objects. In Storm, the sequence of transformations is described by a topology. Topologies run forever, processing stream data as they arrive.



Storm topology

The preceding image shows the basic topology concept. A topology consists of spouts and bolts. Spouts are sources of data. They listen to the source and emit the tuples into the topology. Each circle represents a bolt that implements some processing, and tuples that flow through the DAG of such bolts are called a topology.

Storm has a master-slave architecture. Nimbus is a master node daemon responsible for coordination and monitoring. Worker nodes run supervisors and execute some part of the topology. Nimbus and supervisors talk through ZooKeeper or via a local disk.

Storm has many advanced features, such as support for detailed monitoring, transactional semantics using Trident, and so on, which is beyond the scope of this book. For a detailed coverage on Storm, please refer to *Storm Real-time Processing Cookbook* by Quinton Anderson.

Storm on Mesos

Nathan Marz developed the first version of scheduler and executors, which was further developed by the community at https://github.com/mesos/storm. For running any framework on Mesos, we need a scheduler for resource negotiation with Mesos on behalf of the framework tasks, and we need executors for running these tasks. The following steps will install Storm on Mesos.

1. Install Mesos.
2. Clone the storm-mesos repository and traverse to it:

```
ubuntu@master:~ $ git clone https://github.com/mesos/storm
ubuntu@master:~ $ cd storm
```

3. The repository includes the `build-release.sh` script in the `bin` directory. The script has various subcommands that can be seen using the `-h` flag. Let's download the unmodified Apache Storm distribution. The script downloads the version specified under the `version` property in the `pom.xml` file. By default, it is set to the latest version, which should be appropriate for most cases. If we need the specified version of Storm, we need to set the version property to the desired version. At the time of writing, the default version is `0.9.3`. We can also specify the download mirror to be used by setting the `MIRROR` environment variable:

```
ubuntu@master:~/storm $ ./bin/build-release.sh
downloadStormRelease
```

4. The preceding command will download the file named `apache-storm-VERSION.zip`. We can now use the downloaded archive as the argument to the script:

```
ubuntu@master:~/storm $ ./bin/build-release.sh apache-storm-*.zip
```

5. The preceding command will create the `storm-mesos` distribution in the current directory with the name `storm-mesos-VERSION.tgz`.
6. We need to update the storm configuration to match our cluster settings. Storm uses the YAML configuration format. Update `conf/storm.yaml` to reflect the following changes:

```
mesos.master.url: "zk://master:2181/mesos"
storm.zookeeper.servers:

- "master"
nimbus.host: "master"
```

Here, the `mesos.master.url` parameter specifies `<host:pair>`, where the Mesos master is running. `storm.zookeeper.servers` lists the zookeeper servers to be used by Storm. `nimbus.host` specifies the master of the Storm cluster. The next section describes all the configuration options for `storm-mesos`.

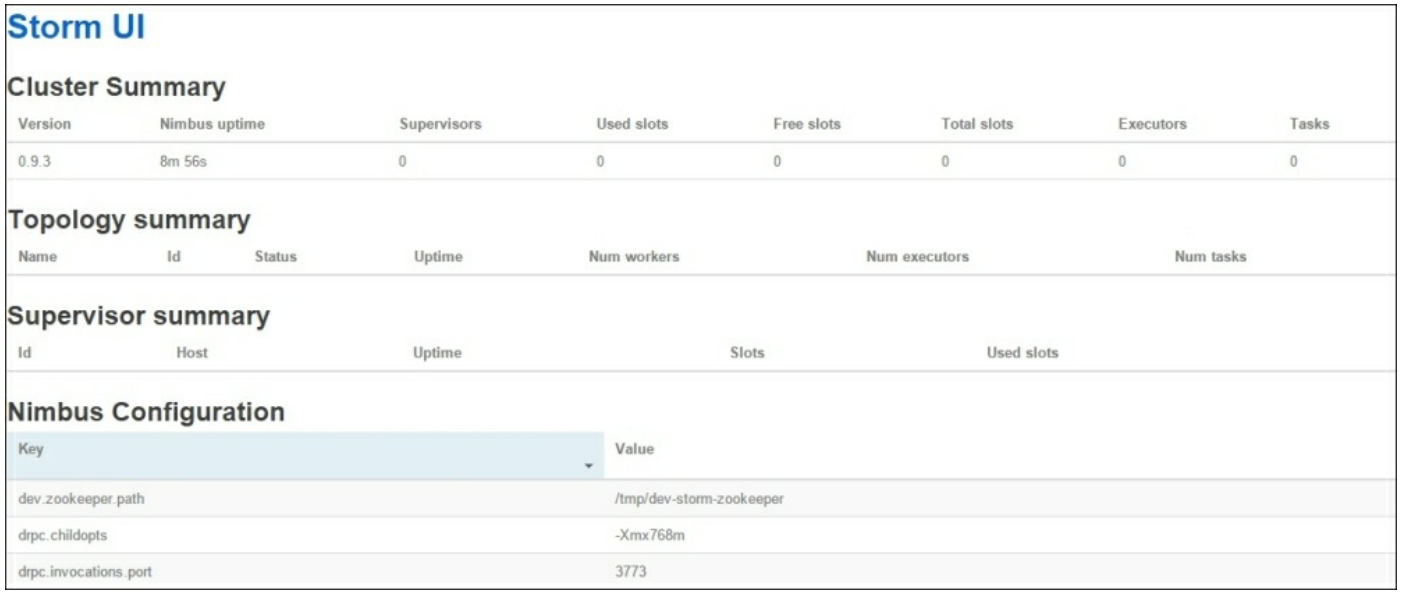
7. Start Storm master-nimbus by running the following command:

```
ubuntu@master:~/storm-mesos $ bin/storm-mesos nimbus
```

8. Optionally, we can also start Storm UI, which is accessible at `<storm-master:port>`; in our case, it is accessible at `http://master:8080`:

```
ubuntu@master:~/storm-mesos $ bin/storm ui
```

At this point, we have Storm running on Mesos. The Storm web UI would look as follows, showing the cluster configuration with 0 supervisors, as the supervisors are created on demand while running topology.



Storm web interface

Also, Storm will be listed as an active framework on the Mesos UI. We can now run various stream processing jobs. Storm project includes a rich set of example topologies under the `examples/storm-starter` directory. Let's run `ExclamationTopology`, which adds exclamations after the input words. `ExclamationTopology` is a basic topology with a spout `word`, and two bolts, `exclaim1` and `exclaim2`, connected in a

linear fashion:

```
ubuntu@master:~/storm-mesos $ bin/storm-mesos jar examples/storm-
starter/storm-starter-topologies-*.jar
storm.starter.ExclamationTopology mytopology
```

Note

Note that Storm requires that the topology names be unique in the given cluster.

The preceding command will submit `ExclamationTopology` to the Storm cluster with the name `mytopology`. We can make sure that our topology is running using the Storm command-line interface as follows:

```
ubuntu@master:~/storm-mesos $ bin/storm list
```

We can also use the Storm web interface to see the various details about the topology. The output, along with other logs, is stored in the `logs` directory, which will show the names of the Storm project contributors followed by exclamations. https://storm.apache.org/documentation is a great place to understand the various Storm concepts in more detail and various Storm commands as well.

Storm-Mesos configuration

Storm-mesos uses YAML-based configuration. At the least, we need to set the following configuration parameters:

- `mesos.executor.uri`: This is the URI of the executors
- `mesos.master.url`: This is the address of the Mesos master
- `storm.zookeeper.servers`: These are the ZooKeeper servers to be used by the Storm master
- `nimbus.host`: This is the hostname where Storm nimbus is running

Controlling the resources configuration is very important, as it can have a significant impact on the scalability and latency of Storm. Storm-mesos has the following configurations for tuning resources:

- `toplogy.mesos.executor.cpu` and `topology.mesos.worker.mem.mb`: This is the CPU and memory per worker respectively. The default values are 1 and 1000 MB respectively. This should be set to at least 25 percent higher than `worker.childopts` to accommodate the task overhead. For example, if `worker.childopts` is set to `-Xmx1000m`, we should set `topology.mesos.worker.mem.mb` to at least 1250.
- `topology.mesos.executor.cpu` and `topology.mesos.executor.mem.mb`: This is the CPU and memory per executor respectively. The default values are 1 and 1000 MB respectively.

Storm-mesos also supports the following optional configuration parameters:

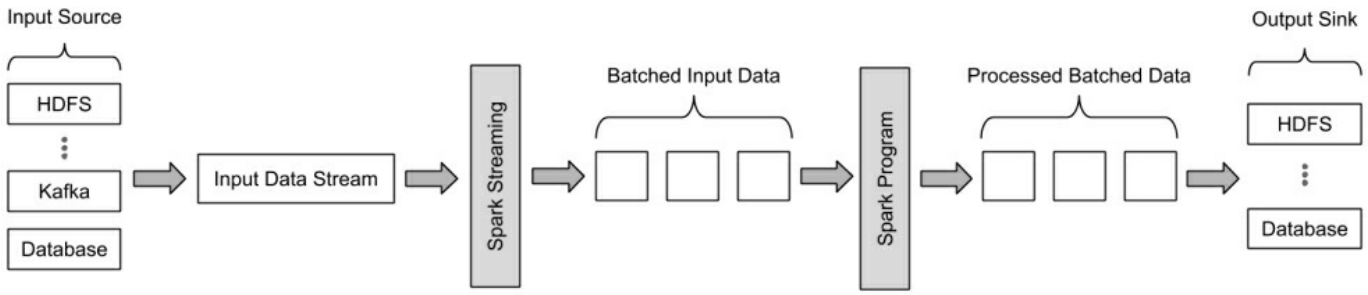
Parameter	**Description**	**Default value**
`mesos.framework.name`	This is the name of the Mesos framework	Storm!!!
`mesos.framework.role`	This is the framework role that can be used for authentication	*
`mesos.framework.checkpoint`	This checks whether to enable framework check pointing	False
`mesos.allowed.hosts (and mesos.disallowed.hosts)`	This is the list of hosts allowed (or disallowed) from running topology	

mesos.offer.lru.cache.size	This is the size of the offer LRU cache	1000
mesos.local.file.server.port	This is the port to be used for the local file server.	Random port
mesos.master.failover.timeout.secs	This is the framework failover timeout for the master in seconds	3600
mesos.supervisor.suicide.inactive.timeout.secs	This is the time (in seconds) to wait before the supervisor decides to kill itself when idle	120

Spark Streaming

We already know that Spark can be used for processing a large amount of data. Spark Streaming is an extension of Spark API to enable the processing of stream data. It supports a large variety of input data sources, including Twitter, HDFS, Kafka, Flume, Akka Actor, TCP sockets, and ZeroMQ. Spark Streaming breaks up the input data stream in small batches, and this discretized stream is then processed by the Spark program. The processed batches can be routed for further processing or can be stored on HDFS, databases, and so on.

Spark Streaming has a basic abstraction of DStream or discretized streams (http://www.cs.berkeley.edu/~matei/papers/2012/hotcloud_spark_streaming.pdf). Internally, DStreams are represented as a sequence of RDDs, and the operations on DStreams are applied to the operations on the RDDs in the DStreams. It has all the benefits of the RDDs, such as persistence, check-pointing, and so on. The following figure shows how Spark enables stream processing.



Spark Streaming architecture

Spark Streaming supports a wide range of operations, both stateless and stateful. The following is a list of currently supported operations:

Transformation	**Description**
`cogroup(stream2, [numTasks])`	This groups tuples from both the streams for the given key; that is, when called for streams of (k, v1) and (k, v2), it returns (k, list<v1>, list<v2>).
`count()`	This returns a new DStream of counts of elements in the source RDDs.

`countByValue()`	This counts the frequencies of keys in the source RDDs and returns (key, frequency) pairs as a DStream.
`filter(f)`	This returns a new DStream with RDDs on which the `f` function returns the value true.
`join(stream2, [numTasks])`	This joins the two streams; that is, for input DStreams of (k, v1) and (k, v2), it returns DStreams of all pairs of (k, (v1, v2)).
`map(f)` and `flatMap(f)`	The `map()` function returns DStream of RDDs that passes through the `f`. function. `flatMap(f)` is similar except that it can be mapped to 0 or more records.
`reduce(f)` and `reduceByKey(f, [numTasks])`	This creates a DStream of single element RDDs, where the value is aggregated by applying the `f` function. `reduceByKey` also takes an argument to control parallelism.
`repartition(numPartitions)`	This repartitions the DStream to control parallelism.
`transform(f)`	This applies an arbitrary f function to each RDD in the DStream and produces a new DStream.
`union(stream2)`	This unions RDDs from the DStream stream2 with the current DStream.
`updateStateByKey(f)`	This produces a new DStream with values updated by applying the update function to each key and the current state of the value for that key.

Transformations supported by Spark Streaming

Spark Streaming also supports window-based operations, which means that we can apply operations over the sliding window of data. The `WindowLength` and `slideInterval` parameters control how the window and operation interval is defined. The following is the list of window-based operations supported:

Transformation	**Description**
`countByWindow(windowLength, slideInterval)`	This returns a sliding window count of elements in the Stream.
`countByValueAndWindow(windowLength, slideInterval, [numTasks])`	This returns a new DStream, where the value of the key is equal to the frequency of the key on the window.

`window(windowLength, slideInterval)`	This creates a new DStream of windowed batches of the given parameter.
`reduceByWindow(f, windowLength, slideInterval)`	This creates a new DStream, where only the element value is aggregated using the `f` function.
`reduceByKeyAndWindow(f, windowLength, slideInterval, [numTasks])` and `reduceByKeyAndWindow(f, invFunc, windowLength, slideInterval, [numTasks])`	This creates a new DStream, where the value of each key is obtained by applying the `f` function to elements in the sliding window. We can also pass the inverse reduce function to make the processing incremental and thus more efficient.

Window-based transformations supported by Spark Streaming

Similar to Spark, these operations are executed lazily, and the following output operations trigger the computations:

Output operations	**Description**
`foreachRDD(f)`	This is the base output operator and must have a side effect (such as print, saving to external systems, and so on).
`print()`	This prints the first ten elements of every batch of data in a DStream.

`saveAsObjectFiles(prefix, [suffix])`
 	
`saveAsTextFiles(prefix, [suffix])`
 	
`saveAsHadoopFiles(prefix, [suffix])`
	This saves the DStream in a sequence file, text file, and HDFS file respectively. The filename is `prefix-TIME_IN_MS[.suffix]`.

Output operations supported by Spark Streaming

Running Spark Streaming on Mesos

If we are running Spark on Mesos, we can start using Spark Streaming right away. We don't have to do anything special to use Spark Streaming. Spark distribution has various examples in the `examples` directory, including Spark Streaming examples. Let's run one of the Spark Streaming examples, `NetworkWordCount`. It counts the number of times a given word occurs in the given network stream every second.

First, we need to create a network stream that can send text on a TCP port using Netcat. Open a terminal and type the following command to start a Netcat server on the port `9999`:

```
ubuntu@master:~ $ nc -lk 9999
```

Now on another terminal, start the Spark Streaming example:

```
ubuntu@master:~ $ cd spark
ubuntu@master:~ $ ./bin/run-example streaming.NetworkWordCount
localhost 9999
```

Now, `NetworkWordCount` is running and listening to any input we type in the first terminal with Netcat. It will print the word and its frequency after every second. For example, if we type in `hello world` in the Netcat window, we will see the following output in the Spark Streaming window:

```
(hello,1)
(world,1)
```

Tuning Spark Streaming

Spark Streaming runs out of the box on Mesos. However, it is very important to tune the systems to meet the real-time stream processing requirements as well as optimize resource usage. There are a few things that we should look for.

Selecting the batch size

Selecting the batch size of Streaming can be a deciding factor in whether we will be able to keep up with the incoming data. The batch size should not be very small otherwise the cluster resources may be wasted. Also, if the batch size is very large, the streaming computations may not make real-time sense. Thus, the recommended approach is to start with a conservative batch size that is meaningful for the application and keep experimenting with smaller sizes. The system is capable of sustaining the current rate if the end-to-end time in processing each batch is comparable to the input batch rates. To measure this time, one can use the `org.apache.spark.scheduler.StreamingListener` interface provided by Spark. A continuous increase in delay is a sign of the system not being able to keep up.

Garbage collection

While running Spark Streaming in production, an important configuration parameter to note is `spark.cleaner.ttl`. It controls how long Spark remembers the metadata. By default, Spark never removes the metadata, which can cause a problem for the streaming applications with huge metadata. Also, remember that if it is set too low, windowed operations might not be able to persist with the RDDs for the length of the window. So, we should set `spark.cleaner.ttl` to a value larger than the largest window length of the streaming applications. If `spark.cleaner.ttl` is not set, Spark will clear all the RDDs in a **Least Recently Used** (**LRU**) fashion. Also, a smarter unpersisting can be enabled by setting `spark.streaming.unpersist` to `true`, where the system figures out which RDDs it can remove from the memory. Also, the use of a concurrent-mark-and-sweep garbage collector for Java virtual machine is recommended, as it allows many smaller GC pauses rather than one big one, which makes the stream processing latency less variable.

Concurrency

It's important that we use available cluster resources to parallelize processing. We should pass appropriate `numTask` parameters to operations, which by default is 8. We can also change the default value that is specified by `spark.default.parallelism`.

Handling failures

We have to consider what happens if the driver node or a worker node fails, as all the intermediate data can be recomputed from the lineage of RDDs. To enable the recovery of the Driver node, we have to enable check-pointing (via the `ssc.checkpoint` option), and the application should check the existence of a previous check-pointed state. If the input source is a network connection and a worker node fails before replicating, it has a chance of replicating the data to the other nodes, and a tiny fraction of data might be lost. For persistent input stores, such as HDFS, no data will be lost due to worker node failures. Spark Streaming documentation (http://spark.apache.org/docs/latest/streaming-programming-guide.html) has a more detailed discussion on fault-tolerance semantics provided by Spark Streaming.

Task overheads

Mesos task launching overhead can be dominant for low-latency applications, such as Spark Streaming. Spark Streaming should run in a fine-grained Mesos mode to reduce task launching overhead, as explained in Chapter 3, *Running Spark on Mesos*. Also, to reduce GC pauses, Spark Streaming persists the RDDs in a serialized byte format. The serialization/deserialization overhead can be non-trivial and the use of fast a serialization framework, such as Kryo (https://github.com/EsotericSoftware/kryo), is recommended. Also, serializing tasks can also reduce the network transfer time of tasks, lowering task launching overhead.

NoSQL on Mesos

SQL has been the primary tool for data analysis for decades. With the rise of big data, there are many systems that try to bring database like ease-of-use to large and complex data analysis landscape. Examples of such systems include Hive, Shark, Spark SQL and NoSQL databases, such as Cassandra, Hypertable, and so on. We can run all these workloads on Mesos and get all the benefits of Mesos, including resource sharing, fault tolerance, and so on. We will now go through the steps of installing Cassandra on Mesos as an example.

Cassandra on Mesos

Apache Cassandra (http://cassandra.apache.org) is a popular NoSQL database. Cassandra was started at Facebook and is actively used in many large-scale deployments. By running Cassandra on Mesos, we can leverage fault-tolerant and scaling behavior of Mesos. Cassandra is very well suited to Mesos due to its completely decentralized architecture.

For running Cassandra on Mesos, we need a scheduler that negotiates the resources required by Cassandra with Mesos and executors that actually execute the Cassandra daemons. The scheduler should also take care of copying the distribution and configuration across all the nodes. Cassandra configuration requires seed nodes to be specified, which is included by the scheduler once it accepts offers from Mesos. Here are the steps to get Cassandra running on Mesos:

1. Install Mesos.
2. Log in to the Mesos master and download the latest prebuilt cassandra-mesos distribution from Mesosphere. At the time of writing, the latest version is 2.0.5:

   ```
   ubuntu@master:~ $ wget
   http://downloads.mesosphere.io/cassandra/cassandra-mesos-2.0.5-
   1.tgz
   ```

3. Untar it and `cd` to the directory:

   ```
   ubuntu@master:~ $ tar xzf cassandra-mesos-*.tgz
   ubuntu@master:~ $ cd cassandra-mesos-*
   ```

4. We need to edit `conf/mesos.yaml` to reflect our cluster configuration. The default configuration is for the local Mesos cluster running with ZooKeeper. The following are the configuration options:

Option	Description	Default value
`mesos.executor.uri`	This is the location of Cassandra executors	`http://downloads.mesosphere.io/cassandra/cassandra-mesos-2.0.5.tgz`
`mesos.master.url`	This is where to find the Mesos master	`zk://localhost:2181/mesos`
`java.library.path`	This is the location of the Mesos library	`/usr/local/lib/libmesos.so`

`cassandra.noOfHwNodes`	This displays how many machines we want Cassandra to run on	1

`resource.cpus`
`resource.mem`
`resource.disk`
	These are the resource requirements of Cassandra on each node	0.1
		2048
		1000

Cassandra-mesos configuration options

5. Start Cassandra on the Mesos scheduler:

```
ubuntu@master:~/cassandra-mesos $ bin/cassandra-mesos
set -o errexit -o pipefail

FRAMEWORK_HOME='dirname $0'/..
dirname $0

export MESOS_NATIVE_LIBRARY=$(sed -e 's/:[^:\/\/]/="/g;s/$/"/g;s/
*=/=/g' "$FRAMEWORK_HOME"/conf/mesos.yaml | tr -d $'\'' | grep -v
\# | grep java.library.path | sed
's/java.library.path=//g;s/"//g')
…
# Start Cassandra on Mesos
…
0 [main] INFO mesosphere.cassandra.Main$  - Starting Cassandra on
Mesos.
…
114 [Thread-0] INFO mesosphere.cassandra.CassandraScheduler  -
Starting Cassandra cluster ${clusterName} for the first time.
Allocating new ID for it.
…
I0429 19:36:36.742849 27508 sched.cpp:391] Framework registered
with 20140429-193514-580538634-5050-25835-0000
175 [Thread-1] INFO mesosphere.cassandra.CassandraScheduler  -
Framework registered as 20140429-193514-580538634-5050-25835-0000
437 [Thread-2] INFO mesosphere.cassandra.CassandraScheduler  -
Got new resource offers ArrayBuffer(slave1)
455 [Thread-2] INFO mesosphere.cassandra.CassandraScheduler  -
resources offered: List((cpus,2.0), (mem,6489.0), (disk,7935.0),
(ports,0.0))
455 [Thread-2] INFO mesosphere.cassandra.CassandraScheduler  -
resources required: List((cpus,0.1), (mem,2048.0), (disk,1000.0))
```

```
464 [Thread-2] INFO mesosphere.cassandra.CassandraScheduler  -
Accepted offer: slave1
…
```

Here, the truncated output shows that a Cassandra cluster is being registered with Mesos and receives resource offers from `slave1`. Now Cassandra is up and running, and this should be listed in the frameworks in the Web UI with the name `Cassandra Test Cluster`. We can interact with it through the **Cassandra Query Language** (**CQL**) shell. Pick any hosts running Cassandra through a command line or through Web UI (`Host` field on UI), and connect a CQL session to it using following command:

```
ubuntu@master:~/cassandra-mesos $ bin/cqlsh <cassandra-host>
```

We should see the Cassandra prompt (`>cqlsh`). We can now run Cassandra queries. Also, note that there are many possibilities and features supported by Cassandra on Mesos (for example, scaling), which we will not go into in detail here.

Summary

Batch processing systems are no longer the only data processing tools available to developers and increasingly, new applications along with use cases require different forms of data analysis than the one supported by the traditional ETL tools and frameworks, such as Hadoop. Thus, rather than trying to make batch processing systems faster or trying to scale traditional databases for unstructured data, using the right tool for the job makes developing and scaling these applications easier.

In this chapter, we explored options for processing real-time and streaming data using Storm and Spark Streaming running on Mesos. We also saw how to do more exploratory data analysis, using Cassandra on Mesos, which also serves as an example of a more general kind of an application on Mesos. This concludes data processing framework coverage on Mesos. In the next section of the book, we will explore the internal working of Mesos and its operational aspects, starting with various scheduling frameworks in the next chapter.

Chapter 5. Running Services on Mesos

In the previous chapters, we have seen how to run various data processing frameworks on Mesos, but services are another big part of the enterprise environment. Enterprises run a wide variety of services and they play a very important role in their success. In this chapter, we will introduce services and different scheduler frameworks for running services on Mesos:

- Introduction to services
- Marathon
- Chronos
- Aurora
- Service discovery
- Packaging chronos

Introduction to services

Traditional enterprises run many long-running services as part of their daily operations, such as running web services, data pipelines, and so on. A service is a self-contained, independently deployed and managed unit of functionality. **Services oriented architectures** (**SOA**) and recently, microservice architectures, encourage you to compose applications out of loosely coupled services. More and more modern applications are composed of multiple microservices, as they offer a number of advantages, such as code reuse, ease of scaling, independent failures, support for multiple platforms, flexibility in deployment, and greater agility.

Until now, we have seen that Mesos is great at handling batch, real-time, and other processing frameworks, whose jobs are typically short-lived. Enterprise infrastructure runs a lot applications and services that are long-running in nature and have different requirements than the data processing frameworks. These long-running services are not only critical for business, but they also consume a large portion of infrastructure resources. Thus, the ability to run services on Mesos is very important.

To run services at scale, the infrastructure needs to be able to support the following:

1. Deployment is concerned about where a given service should run. Deployment of a service can be complex if service depends on other services and there are constraints about where the service can be deployed.
2. Configuration management and packaging is about making sure all the dependencies for a service are met and the environment is configured properly for the service before the service starts.
3. Service discovery and load balancing become important when multiple instances of a service are running. Service discovery answers the questions *where are the instances of a particular service running?* and load balancing is about deciding *which instance should a given request go to?*
4. Once the service is deployed, it is important to do health monitoring of the service. Health monitoring information can be used to take further actions, such as scaling a service up or down or relaunching them on a failure.
5. Availability requirement demands that the service needs to be available in the case of high load and failures.

If we think of Mesos as the data center kernel, service schedulers are the next most important thing in the large-scale infrastructure. Although early frameworks developed

on Mesos were mostly targeted toward data processing, running services is essential for enterprise environments. There are definitely many applications that need to be written or rewritten to take full advantage of the distributed infrastructure that Mesos provides, but there are many applications that just need to start executing on nodes and need to restart if they fail. Scheduler frameworks allow you to achieve this. Service schedulers are also known as meta-frameworks or meta-schedulers, as they act as a mediator for the applications running on top. Service schedulers are responsible for making services and making them run by taking care of the complexities of handling failures, restarting them, and scaling them as required.

Marathon

Marathon is a framework used for running long-running services on Mesos (https://github.com/mesosphere/marathon). These services have high availability requirements, which means that Marathon should be able to start the service instance on other machines in case of a failure and should be able to scale elastically. This would include anything that can run on a standard shell as well as special executors. Marathon acts like `init.d` for your data center or cluster. This means it can make sure that services running on top of it are always running. Marathon is designed to run tasks and guarantees that they stay running. Marathon can run other frameworks, such as Hadoop, as well as itself.

The Marathon API

Marathon has a very comprehensive REST API to manage the life cycle of the services as well as clients available in Java, Scala, Python, and Ruby:

Type	Endpoint	Description
Apps	`GET /v2/apps`	This lists all the running applications. It also accepts a `cmd` parameter, which if specified, lists only applications matching the value of the `cmd` parameter.
	`GET /v2/apps/appid`	This lists the application that has the `appid` application ID.
	`GET /v2/apps/appid/versions`	This lists all the versions of the application for the application.
	`GET /v2/apps/appid/versions/version`	This returns a configuration of the application that has the specified `version`.
	`GET /v2/apps/appid/tasks`	This lists all the tasks for the application.
	`GET /v2/tasks`	This lists all the tasks of all the applications.
	`POST /v2/apps`	This starts a new application. All the application details are specified in the JSON format along with the request.
	`POST /v2/apps/appid/restart`	This specifies the rolling restarts of all the tasks of the specified application.
	`PUT /v2/apps/appid`	This modifies the configuration of the future tasks of a particular application.
	`DELETE /v2/apps/appid`	This destroys the application.
	`DELETE /v2/apps/appid/tasks`	This kills all the tasks of the application. If the host's parameter is specified, only tasks run on those hosts. If the scale parameter is set to true, the configuration of the application is updated to scale down by the number of tasks killed by the request.
		This kills a task with the specified `taskid` of the application. If

	`DELETE /v2/apps/appid/tasks/taskid`	the scale parameter is set to true, the application is scaled down by 1 on a successful request.
Groups	`GET /v2/groups`	This lists all the groups.
	`GET /v2/groups/groupid`	This lists the groups with the specified `groupid`.
	`POST /v2/groups`	This creates and starts a new application group.
	`PUT /v2/groups/groupid`	This changes parameters of a deployed application group.
	`DELETE /v2/groups/groupid`	This destroys the specified group.
Deployments	`GET /v2/deployments`	This lists all the running deployments.
	`DELETE /v2/deployments/deploymentid`	This cancels the deployment with the specified `deploymentid`.
Server info	`GET /v2/info`	This will display `info` about the current Marathon instance.
Event subscriptions	`GET /v2/eventSubscriptions`	This lists callback URLs of all the event subscriptions.
	`POST /v2/eventSubscrptions`	This registers the callback URL, which is passed as a parameter, for event subscriptions.
	`DELETE /v2/eventSubscriptions`	This unregisters the callback URL, which is passed as a parameter from event subscriptions.
Queue	`GET /v2/queue`	This lists the content of the task queue.
Miscellaneous	`GET /ping`	This returns the `ping` status.
	`GET /logging`	This displays the `logging` status.
	`GET /metrics`	This returns `metrics` from Marathon.
	`GET /help`	This prints `help`.

Running Marathon

Here are the steps to run Marathon:

1. Install Mesos.
2. Download the Marathon distribution. You can choose to build from a source, which is also straightforward, following the documentation. At the time of writing, the latest version is `0.8`:

   ```
   ubuntu@master:~ $ wget
   http://downloads.mesosphere.io/marathon/v0.8.0/marathon-0.8.0.tgz
   ```

3. Extract the distribution and cd to it:

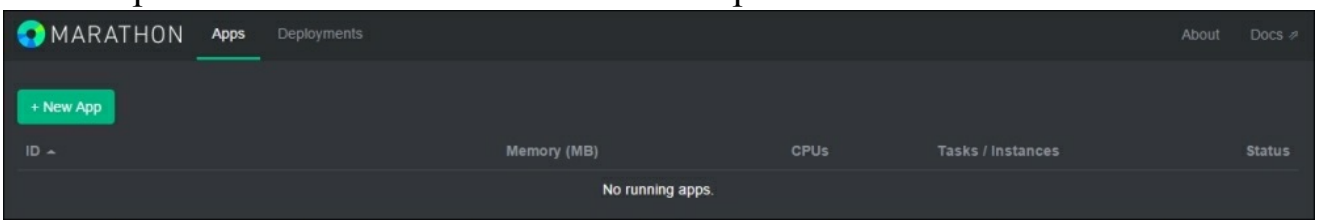
   ```
   ubuntu@master:~ $ tar xzf marathon-*.tgz
   ubuntu@master:~ $ rm marathon-*.tgz; cd marathon-*
   ```

4. Start Marathon using the script. You need to pass the address of the Mesos master and nodes running ZooKeeper to the script:

   ```
   ubuntu@master:~/marathon $./bin/start --master
   zk://master:2181/mesos --zk master:2181
   ```

   Marathon provides you with many options to adapt to your requirements. A complete list of command-line flags is available at http://mesosphere.github.io/marathon/docs/command-line-flags.html.

5. The script also starts the Marathon web UI on port `8080`:

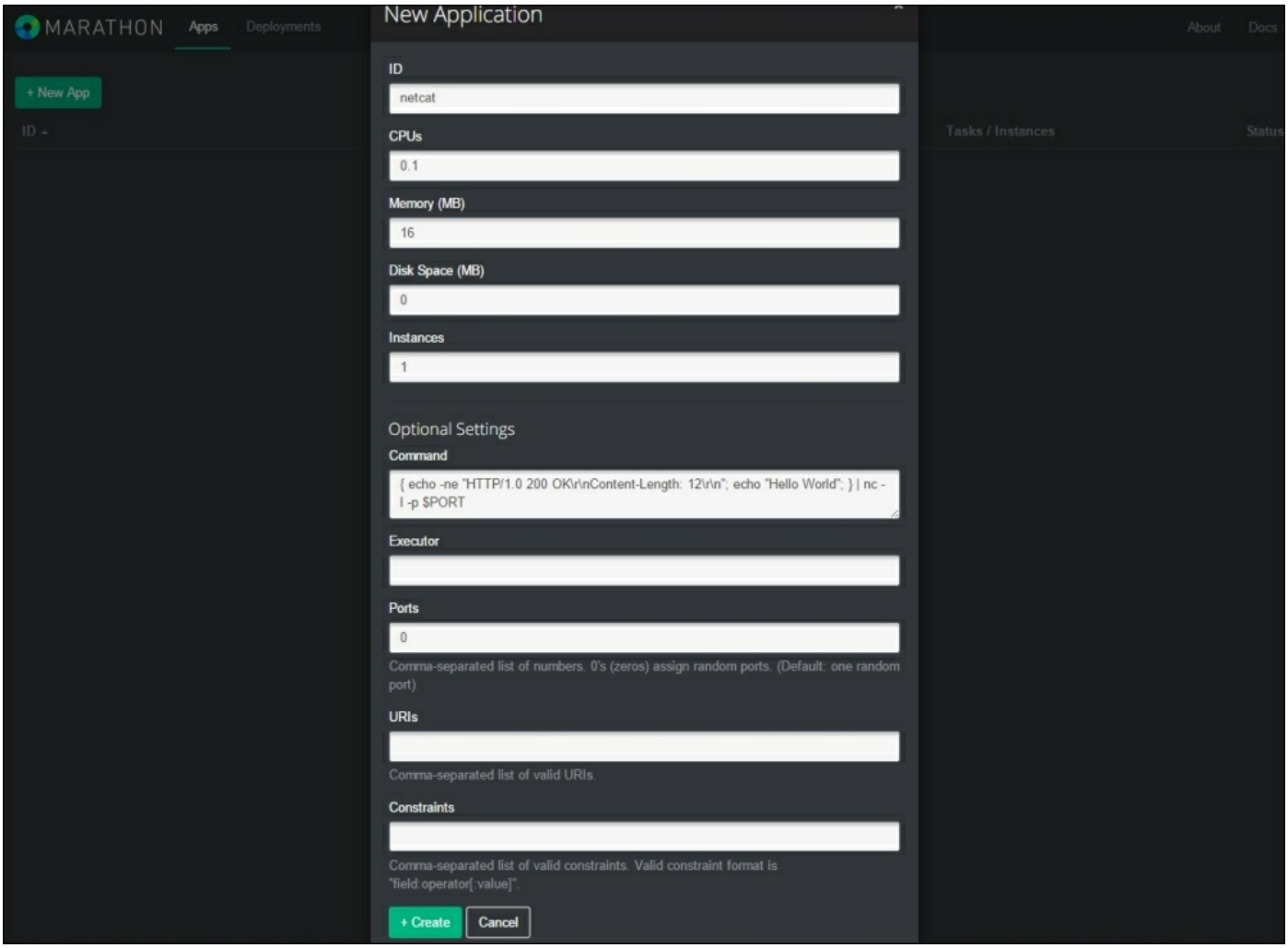


Marathon example

The Marathon web UI provides a convenient interface with Marathon. Let's run a very simple web server as a job to verify the installation. Create a new App and provide the details about the resources required for the task and how to run that task. Provide some names for our application in the ID field. For our simple task, 0.1 CPU and a 10 MB memory is more than enough for resources.

Insert the following string in the **Command** field:

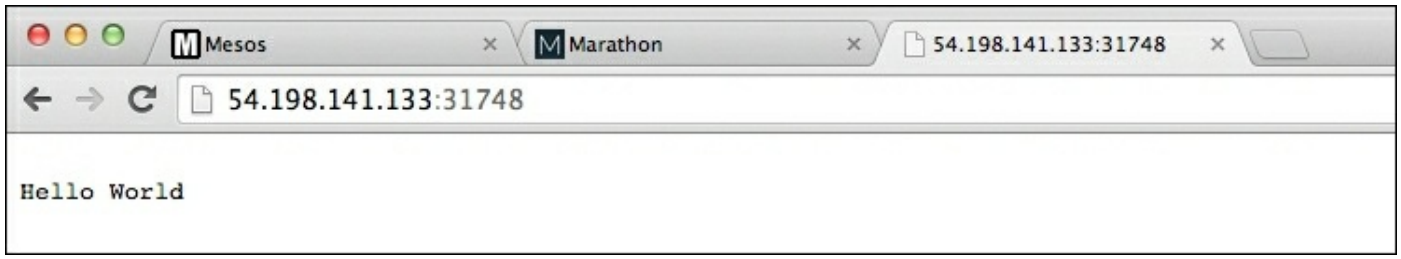
```
{ echo -ne "HTTP/1.0 200 OK\r\nContent-Length: 12\r\n"; echo "Hello
World"; } | nc -l -p $PORT
```



This command sends a simple **Hello World** to the browser using the `netcat` utility; note that it picks the port via the `$PORT` variable. The Marathon UI will now show you the

job status. You can choose to scale the job from web UI. Also, the Mesos web UI will also show you the task corresponding to the Marathon jobs running. Note the host and port on which the task is running, so that we can check our server. We can also use Marathon's REST API to find out this information. This command finishes the execution after serving a single request.

So, it's Marathon that is keeping our single request server available by creating new tasks each time:



The Marathon distribution includes many example jobs in the `examples` directory. Note that we could have started the same job using the Marathon REST API, instead of using a web interface. Here is `hello_world.json` that describes our example server:

```
{
  "id": "netcat",
  "cmd": "{ echo -ne "HTTP/1.0 200 OK\r\nContent-Length: 12\r\n";
echo "Hello World"; } | nc -l -p $PORT",
  "mem": 10,
  "cpus": 0.1,
  "instances": 1,
}
```

We can use `hello_world.json` to launch our example server using the Marathon REST API:

```
ubuntu@master:~ $ curl -X POST -H "Content-Type: application/json"
http://master:8080/v2/apps -d@hello_world.json
```

Now we will take a look at some of the features of Marathon.

Constraints

One of the most desired features of any scheduler is to control where and how the tasks are spawned. Constraints in Marathon allow restricting where a particular application runs, and thus, you can benefit from the locality, fault tolerance, or any other constraint. Constraints are enforced at the time of starting the application. A constraint consists of a field, operator, and optional parameter. A field is any attribute that is present on the Mesos slave or its hostname. Marathon allows constraints to be specified as `--constraint` via the command-line option or as constraint's field via the REST interface.

There are three operators that we can use to specify constraints:

- UNIQUE: This applies uniqueness to the field. For example: running only one instance/task on each host.
- CLUSTER: This restricts tasks to nodes with a particular attribute. This can be useful to enforce locality, special h/w, and so on.
- GROUP_BY: This distributes tasks evenly across given fields (racks/data centers). An optional argument specifies a minimum number of groups to try.
- LIKE: This operator allows you to filter hosts based on a regular expression in a field.
- UNLIKE: This operator is the inverse of the like operator. It filters the host not matching the given regular expression.

Event bus

Marathon's internal bus captures all the events of Marathon, API requests, scaling events, and so on. This can be used to integrate load balancers, monitoring and other external system with Marathon. This can be enabled at the start time by supplying `--event_subscriber <subscription>` at the start. Subscription is a pluggable interface, which defaults to an HTTP callback subscriber. This will POST events in the JSON format to hosts specified at `--http_endpoint`. The following events occur on the event bus:

- API requests that are create/update/delete requests for applications
- Status updates are received every time the task status changes
- Framework message events are for each framework message
- Event subscription events are triggered when new event subscribers are added or removed
- Health check events are triggered for various health check events, such as add, remove, failed, or status change
- Deployment events are generated for various deployment events, such as success or failure of various deployments

The artifact store

The artifact store is the location for storing application-specific resources for deployment that an application needs in order to run, like certain files, for example. An application running on Marathon can use the `storeUrls` field as part of the application definition. `storeUrls` is a list of URLs. Each URL is downloaded and stored in the artifact store. Once the download is done, the path of the artifact is added to the application definition using the `uris` field. The path is unique for the content and is not downloaded again, which also makes the application run faster. The artifact store has its own REST API that allows the **CRUD** (**Create, Update, Read, Delete**) operations on artifacts. The artifact API allows more automation and simplifies deployments. The artifact store can run on various backend storage systems, including the local file system and HDFS. To configure Marathon with the artifact store, use `--artifact_store` that provides the path artifact store location. Note that all the functionality of an artifact store can be achieved manually without an artifact store and thus, the use of an artifact store is optional.

Application groups

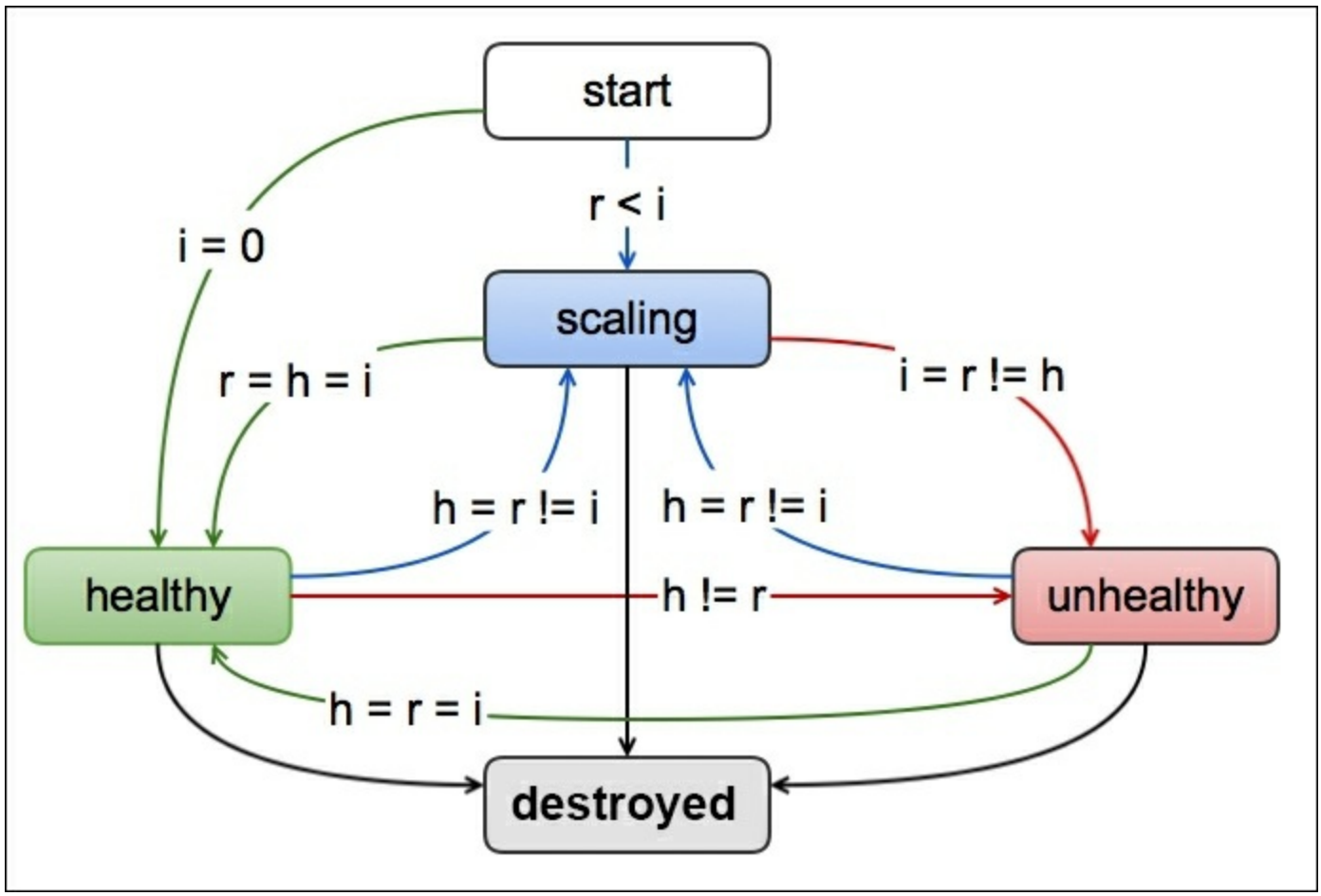
Application groups are used to partition applications into disjoint sets for management. Groups make it easy to manage logically related applications and allow you to perform actions on the entire group, such as scaling. For example, the scaling of the group by 2 will double the count of all the applications in that group. Groups can be composed of applications or groups. Groups are hierarchical and can be expressed via relative or absolute path. So groups can be composed of groups. Here is an example of a group definition:

```
{
  "id": "/product",
  "groups": [
    {
      "id": "/product/db",
      "apps": [
         { "id": "/product/mongo", ... },
         { "id": "/product/mysql", ... }
       ]
    },{
      "id": "/product/service",
      "dependencies": ["/product/db"],
      "apps": [
         { "id": "/product/rails-app", ... },
         { "id": "/product/play-app", ... }
      ]
    }
  ]
}
```

Marathon takes dependencies into account while starting, stopping, upgrading, and scaling. Dependencies can be defined at the level of applications or at the level of groups. Dependencies defined at the level of groups are applicable to all the members of the group (other groups or applications).

Application health checks

Detailed health-checking features are essential for long-running services. Marathon defaults to a health check that assumes the application state to be healthy as long as the Mesos task status is `TASK_RUNNING`. While this simple health check might be sufficient for simple applications, many applications have different semantics when its status can be considered `healthy` and when it needs to be restarted. The application life cycle in Marathon is depicted in the following figure, where `i` is the number of requested instances, `r` is number of instances running, and `h` is the number of healthy instances, and labels on the arrows indicate when and under which condition that transition would happen:



Health checks start immediately after the task is up, but initial failures for `gracePeriodSeconds` are ignored. If the task is not in a healthy state after `maxConsecutiveFailures`, it is killed. Currently, Marathon supports HTTP and TCP health checks, but more are under development.

Chronos

Marathon make it possible to run long-running services. Another important piece of infrastructure is the recurring jobs. Typical enterprises run dozens of jobs that do not need to always be running but have to be triggered at a particular time repeatedly, such as backups, **Extract-Transform-Load** (**ETL**) jobs, running other frameworks, and so on.

The repeated execution functionality has been traditionally achieved via cron utility and shell scripts. This is not only error prone and difficult to maintain, but is also not reliable. If the node on which the cron job is scheduled dies, the job will not be executed and the enterprise workflow might halt. Chronos (https://github.com/mesos/chronos) is a fault-tolerant job scheduler that handles dependencies and ISO8601-based schedules and acts as a cron for the Mesos data center kernel. ISO8601 is an international standard used to represent and exchange date and time-related data. Chronos uses ZooKeeper to achieve fault tolerance and relies on Mesos to execute the job. Chronos allows you to run shell scripts and also supports dependencies and retries.

The Chronos REST API

Chronos has a REST API for the entire job management and monitoring. Chronos REST API is very useful for automation and is used by the web UI. The following list shows you some important Chronos APIs:

Endpoint	Description
`GET /scheduler/jobs`	This lists all jobs. The result is that JSON contains executor, invocationCount, and schedule/parents
`DELETE /scheduler/jobs`	This deletes all jobs
`DELETE /scheduler/task/kill/jobName`	This deletes tasks for a given job
`DELETE /scheduler/job/jobName`	This deletes a particular job based on jobName
`PUT /scheduler/job/jobName`	This manually starts a job
`POST /scheduler/iso8601`	This adds a new job. The JSON passed should contain all the information about the job, including any dependency on jobs
`GET /scheduler/graph/dot`	This returns the dependency graph in the form of a dot file

Chronos uses JSON to describe the jobs which can have the following fields:

- name: This specifies the job name.
- owner: This specifies the e-mail address of the person who owns the job.
- command: This specifies the command to be run by Chronos.
- schedule: This specifies the ISO8601 format schedule for the job. It contains three parts separated by /:
    - Number of times the job should be run. By specifying "R", this repeats the job forever.
    - Start time of the job. Empty start time will start the job immediately.
    - Interval between runs.
- async: This checks whether or not the job runs in the background.
- epsilon: This specifies the interval within which it is ok to start the job, in case

Chronos misses the scheduled runtime. This is also formatted according to the ISO8601 notation.

Here is an example of JSON that specifies a job with the `HelloWorldJob` name. It will append **Hello World** to the `/tmp/HelloWorldOut` file on every run. The job runs 10 times every 2 seconds, starting from February 1, 2015, and a run can be delayed by 15 minutes:

```
{
  "schedule": "R10/2015-02-01T05:52:00Z/PT2S",
  "name": "HelloWorldJob",
  "epsilon": "PT15M",
  "command": "echo 'Hello World' >> /tmp/HelloWorldOut",
  "owner": "me@example.com",
  "async": false
}
```

Running Chronos

Chronos includes a lot of helper scripts in the bin folder; for example, wrapper scripts allow you to transfer files and then execute them on remote machines. The distribution also includes scripts to install Mesos:

1. Install Mesos.
2. We can build Chronos from a source by following the documents in Github. We will use a prebuild version available at:

   ```
   ubuntu@master:~ $ wget
   http://downloads.mesospher.io/chronos/chronos-2.1.0_mesos-0.14.0-
   rc4.tgz
   ```

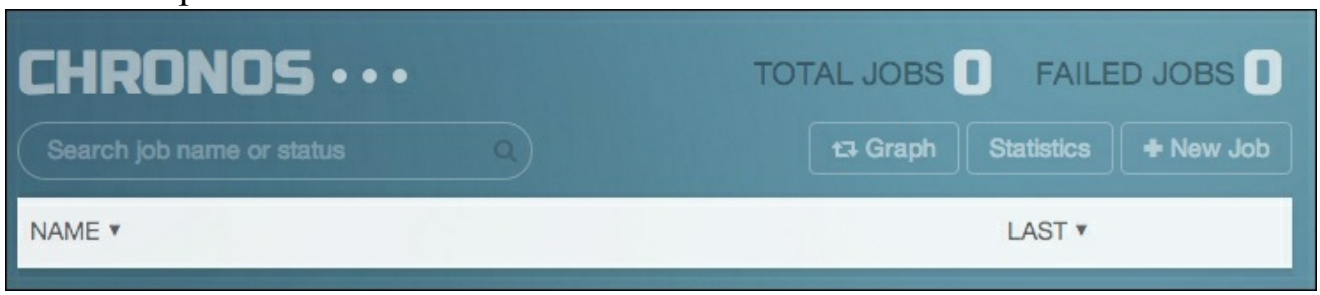
3. Extract and `cd` to it:

   ```
   ubuntu@master:~ $ tar xzf chromos-*.tgz
   ubuntu@master:~ $ cd chronos
   ```

4. Start the Chronos using the bash script. It requires the address of the Mesos master and ZooKeeper hosts:

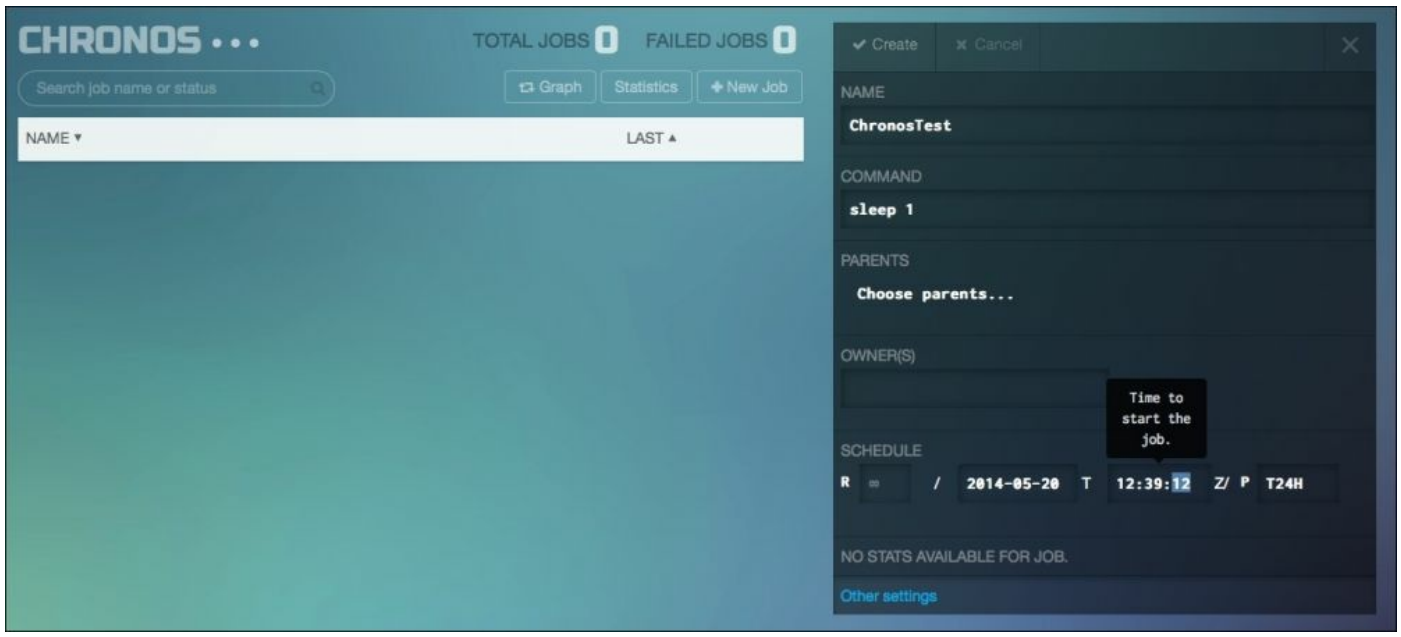
   ```
   ubuntu@master:~ $./bin/start-chronos.bash --master
   zk://localhost:2181/mesos --zk_hosts zk://<Mesos>:2181/mesos --
   http_port 8081
   ```

   Note that `start-chronos.bash --help` lists all the configuration options for Chronos. In the default configuration, Chronos runs on port `8080`. Here, we have specified to run it on port `8081` instead.
5. This starts Chronos and registers it as a framework with Mesos. It also starts the web UI on port `8081`.



A Chronos example

Let's create a simple job running at scheduled times. On the web UI, click on **+New Job**, and it will pop up a panel with details of the job. We will run a simple sleep command every day and choose the time you want the job to trigger and the cycle time of the job. Note that the time is in **UTC** (**Coordinated Universal Time**). If you want your job to be triggered after some job is finished instead of being triggered on time, you can specify the other job in the parent field. Once you create a job, it will be listed as `fresh`. Once the job is scheduled, it will change the status to `success` or `failure`. The tasks will also be listed in the Mesos UI, and you can access `stdout`, `stderr`, and `logs` from there:



We can create the same job using the Chronos REST API by providing the appropriate JSON.

```
ubuntu@master:~ $ curl -L -H 'Content-Type: application/json' -X POST
-d '{<job-config>}' chronos-node:8080/scheduler/iso8601
```

Chronos includes a lot of useful tools and features and can schedule very complex jobs pipelines, which we will not explore in detail here.

Aurora

Apache Aurora (http://aurora.apache.org) is another service framework that is used to schedule jobs to Apache Mesos. It provides many of the primitives that allow stateless services to quickly deploy and scale, while providing good assurances for fault tolerance. Aurora is a feature-rich job scheduling framework that is optimized for a large number of jobs and high user count. It was developed at Twitter and ran mission-critical infrastructure at many organizations even before becoming an Apache project.

An Aurora job consists of multiple similar task replicas. An Aurora task can be a single process or a set of processes running on the same machine. Because Mesos only provides the abstraction of a task, Aurora provides a component called Thermos to manage multiple processes within a single Mesos task. As each task runs within a sandbox, the Thermos process can share a state with each other. Thermos enables the execution of arbitrary processes and provides an interface where it can be observed/monitored. Thermos basically consists of two parts. The executor is responsible for launching and managing the task, and the observer is responsible for monitoring a daemon, and providing information about the running tasks. Aurora architecture consists of the following components:

- Client
- State machine
- Event bus
- Storage
- Event bus subscribers

The client contacts to the state machine for all the information. State machine events are published on an event bus. Different components, such as scheduling, throttling, stats collection, and so on, can subscribe to different events on the event bus. State machine is backed by storage. Aurora uses storage to persist different data (task configuration, job configuration, update progress, and so on) for failover, as described in http://aurora.apache.org/documentation/latest/storage. Aurora uses storage for saving various information such as:

- What jobs are running?
- Where they are running?
- Configuration of the job
- The Mesos master, resource quotas, locks, and other metadata

At a high level, architecture for storage in Aurora consists of four major components:

- Volatile storage is responsible for caching data in-memory. Aurora implements volatile storage using the H2 database.
- Log manager acts as an interface to Mesos' replicated log. The log manager storage is in the form of binary data using Thrift.
- Snapshot manager is responsible for checkpointing data to Mesos replicated log. This helps in speeding up the recovery process.
- Backup manager backs up snapshots into backup files. This guards against loss or corruption of the Mesos log.

All the stored data is immutable and any modification is done by storing new versions of the data. The storage interface provides three kinds operation:

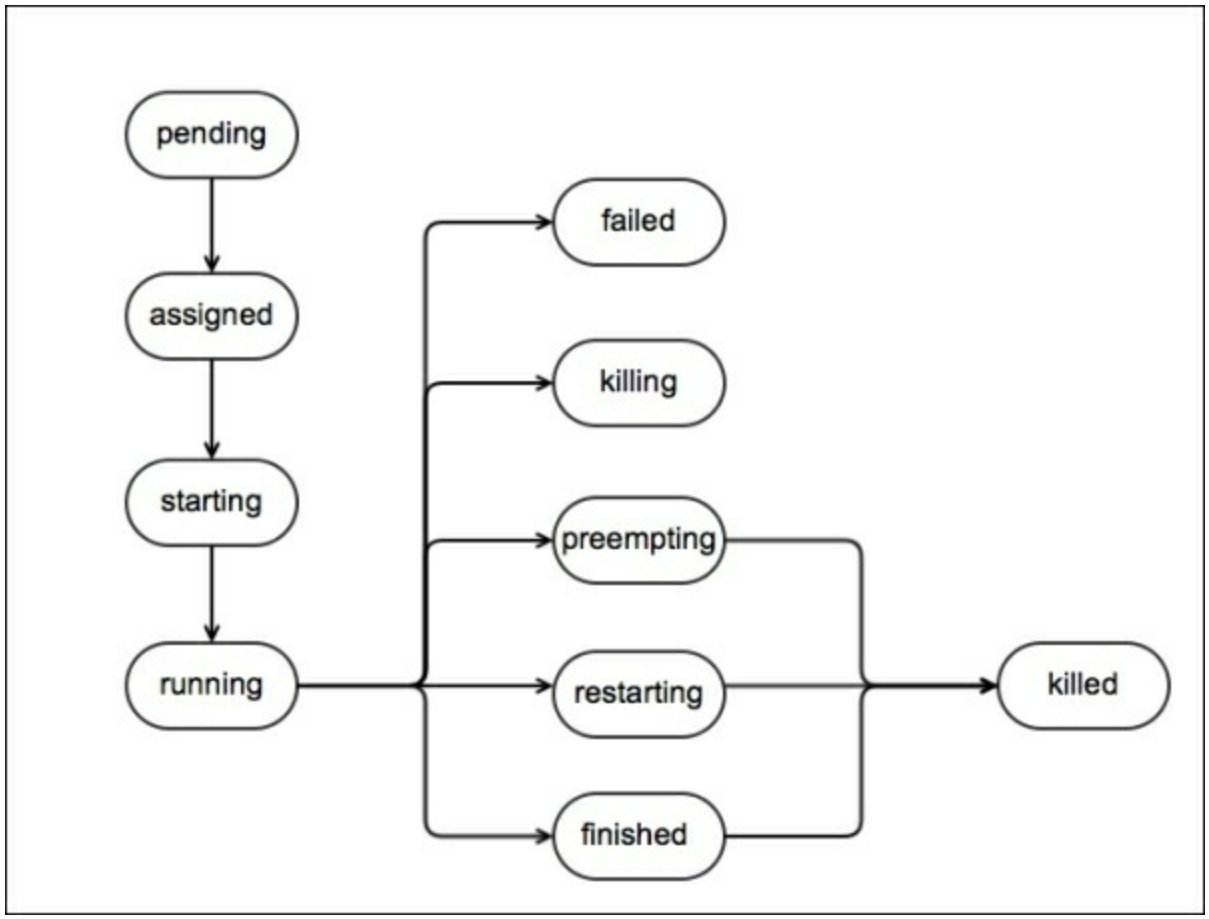
- consistentRead provides a consistent view of the data using reader's lock.
- weaklyConsistentRead provides a view of the data without locking and might not be consistent.
- All writes are serialized with the writer's lock and are first stored in the replicated log and then in the volatile storage. This form of write-ahead-logging is used to ensure consistency, which means write to replicated log and volatile storage has to succeed for the write operation to succeed.

Aurora has master-slave architecture for ensuring high availability, where the slaves follow log writes and can become masters in the case of failures. Aurora discovers the master via ZooKeeper. The quorum size should be a majority of the number of instances running and can be specified using flag `-native_log_quorum_size`. After a failure, Aurora expects some external service to restart the Aurora process. External utilities, such as Monit, can be used to restart the Aurora process. Aurora announces services to ZooKeeper that can be used for service discovery.

Aurora has many features that are invaluable in production. Rolling updates is one such feature. Rolling updates allow the job configuration to be staged in a small group of processes at a time. Any tasks running with the old configurations are killed, and the tasks with a new configuration are created. The Aurora client will continue to update the required number of tasks. If a significant percentage of updated tasks have failed, then the client triggers a rollback. Rollback or update cancellation restores the old configuration and destroys any tasks with a new configuration and creates tasks with the old configuration.

Job life cycle

The following image depicts the life cycle of an Aurora task. A new task starts its life as a **pending** task. The scheduler tries to find a machine that satisfies the task's constraints. Once such a machine is found, the task status is now **assigned**. The slave receives the configuration of the task from the scheduler and spawns the executor. Once the scheduler receives the acknowledgement from the slave, the task's status is changed to **starting**. Once the task is fully initialized, Thermos starts running the processes at which point the scheduler marks the status as **running**. If the task takes too long in the **pending** or **assigned** state, the scheduler marks it as **lost** and creates a new task in the **pending** state. The natural termination leads to a **finished** or **failed** state, while a forced termination will lead to a **killing** or a **restarting** state.



Aurora task life cycle

In case of higher resource requirements or shrinking hardware resources (for example,

due to power outage), certain important jobs need to be given priority over other jobs. Aurora uses the notion of a production job for the jobs that are more important than other jobs that are nonproduction and will be killed if required to free some resources. These tasks are marked as the `PREEMPTING` state, which will eventually transition to the `KILLED` state.

Running Aurora

Here are the steps to install Aurora on Mesos:

1. Install Mesos.
2. At the time of writing, there is no distribution available for Aurora, and we have to build it from source. Let's clone the Aurora source code:

```
ubuntu@master:~ $ git clone http://git-wip-
us.apache.org/repos/asf/aurora.git
ubuntu@master:~ $ cd aurora
```

3. Aurora uses Gradle for builds, which means the Gradle script will download all the dependency and create a distribution:

```
ubuntu@master:~/aurora $ ./gradlew distZip
```

4. The previous step will generate the `auroa-scheduler-*.zip` ZIP file in the `dist/distribution` folder. We need to copy it to all the Mesos nodes and extract it to `/usr/local`, and we are ready to use Aurora:

```
ubuntu@master:~/aurora $ sudo unzip dist/distributions/aurora-
scheduler-*.zip -d /usr/local
ubuntu@master:~/aurora $ sudo ln -nfs "$(ls -dt
/usr/local/aurora-scheduler-* | head -1)" /usr/local/aurora-
scheduler
```

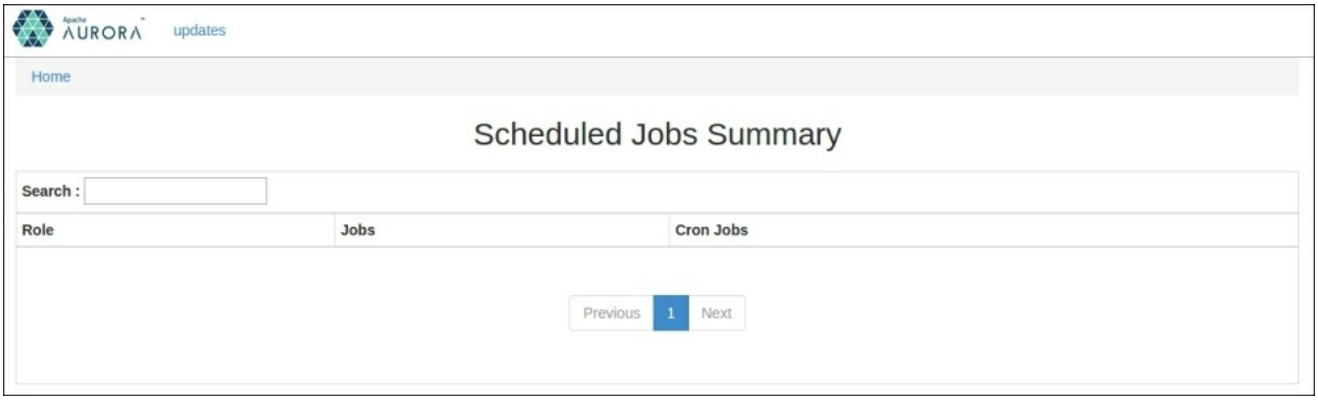
5. We can now start Aurora by specifying the appropriate flags. Running the `-help` aurora-scheduler will list all the Aurora flags:

```
ubuntu@master:~/aurora $ /usr/local/aurora-scheduler/bin/aurora-
scheduler -help
```

6. We need to initialize the replicated log to be used with Aurora.
7. Here, we will use only minimal configuration and provide only the required options. The examples directory provides comprehensive example configurations:

```
ubuntu@master:~ $ /usr/local/aurora-scheduler/bin/aurora-
scheduler -cluster_name=test -
mesos_master_address=zk://localhost:2181/mesos/ -
serverset_path=/aurora/scheduler
```

   The previous command will also start the scheduler and web interface on port 8081:



It is worth noting that Aurora provides a vagrant setup, which is a great way to play around with Aurora. To try this, `cd` to the extracted directory and run the `vagrant up` command. This will install Mesos and Aurora in a single VM. We can see more details on this in the documentation (http://aurora.apache.org/documentation/latest/vagrant).

Aurora cluster configuration

Aurora cluster configuration describes Aurora clusters, and Aurora clients can use it to refer to different clusters using a short name. Cluster configuration is in the JSON format and have the following fields:

- `name`: This is a user-friendly name of the cluster that is represented by this configuration. This is a required field.
- `slave_root`: This is the working directory of the Mesos slave. The Aurora client uses this to locate the executing tasks on the slave. This is a required field.
- `slaverundirectory`: This specifies the name of the directory where the task runs can be found. For most cases, its value should be "latest". This is a required field.
- `zk`: This is the hostname where ZooKeeper is running.
- `zk_port`: This is the port where ZooKeeper is running. The default value is `2181`.
- `schedulerzkpath`: This is the path in ZooKeeper where the Aurora server sets are registered.
- `scheduler_uri`: This is the URI of the Aurora scheduler. If specified, ZooKeeper-related configuration is not used and `scheduler_uri` is directly used to connect to the scheduler. Thus, using `scheduler_uri` is only recommended while testing and not in production.
- `proxy_url`: If specified, `proxy_url` will be used, instead of the URI of the leading master.
- `auth_mechanism`: This is the identifier used by the client for authentication with the Aurora scheduler. This feature is a work in progress and currently, the only value supported is `UNAUTHENTICATED`.

The Aurora client can load two cluster configuration files:

1. System-wide cluster configuration whose path is specified by the `AURORA_CONFIG_ROOT` environment variable (defaults to `/etc/aurora/cluster.json`).
2. User-specific cluster configuration file is located at `~/.aurora/clusters.json`.

Aurora job configuration

Before we can run an Aurora job, we have to create a configuration file, detailing all the information required to schedule a job. An Aurora configuration file is just a Python file that uses the pystachio library to specify the configuration. This allows the configuration to be composed by reusing building blocks. The configuration file must have an `.aurora` extension. There are three types of object in a configuration file:

- Process
- Task
- Job

A configuration file defines templates, processes, tasks, and jobs, in that order. Templates encapsulate all the differences between jobs in the configuration that are not directly manipulated in the form of attributes. Here is an example of a configuration file from the Aurora documentation:

```
# --- templates here ---
class Profile(Struct):
  package_version = Default(String, 'live')
  java_binary = Default(String, '/usr/lib/jvm/java-1.7.0-
openjdk/bin/java')
  extra_jvm_options = Default(String, '')
  parent_environment = Default(String, 'prod')
  parent_serverset = Default(String,
'/foocorp/service/bird/{{parent_environment}}/bird')

# --- processes ---
main = Process(
  name = 'application',
  cmdline = '{{profile.java_binary}} -server -Xmx1792m '
            '{{profile.extra_jvm_options}} '
            '-jar application.jar '
            '-upstreamService {{profile.parent_serverset}}'
)
# --- tasks ---
base_task = SequentialTask(
  name = 'application',
  processes = [
    Process(
      name = 'fetch',
cmdline = 'curl -O
https://packages.foocorp.com/{{profile.package_version}}/application.
jar'),
  ]
```

```
)
# --- job template ---
job_template = Job(
  name = 'application',
  role = 'myteam',
  contact = 'myteam-team@foocorp.com',
  instances = 20,
  service = True,
  task = production_task
)
# -- job --
jobs = [
      job_template(cluster = 'cluster1', environment = 'prod')
               .bind(profile = PRODUCTION),

      job_template(cluster = 'cluster2', environment = 'prod')
                .bind(profile = PRODUCTION),

      job_template(cluster = 'cluster1',
                   environment = 'staging',
        service = False,
        task = staging_task,
        instances = 2)
        .bind(profile = STAGING),
]
```

A `Process` object has two key attributes: `name` and `cmdline`. The `name` attribute should be a unique string and `cmdline` should be anything that can be executed by bash shell. You can chain multiple bash commands just like running on bash shell. The `Process` object allows you to specify the following optional attributes:

Attribute	**Description**
`max_failures`	This denotes the maximum number of retries after failures, after which the process is marked as permanently failed and not retired. This defaults to 1. A value of 0 will mean infinite retries.
`daemon`	Daemon processes are reinvoked after `min_duration` period. With `max_failures=0` and `daemon=True` the process will retry indefinitely.
`ephemeral`	If set to `true`, the exit status of the processes is not used to determine the task's completion. This defaults to `False`.
`min_duration`	This specifies the time in seconds between rerunning the same task. This defaults to `15`. This is used to guard the scheduler from Denial of Service attack.

`final`	This process is a finalizing process and should run after all the processes have finished or failed. Thermos invokes the finalizing processes in the finalization stage. This is a typical use case of cleanup and bookkeeping. This defaults to `Fals e`.

Aurora includes many helper objects and functions to simplify the creation of complex tasks. A task object consists of three key attributes:

- `name`: This is the name of the task. If unspecified, this will be set to the name of the first process.
- `processes`: This is an unordered list of processes.
- `resource`: This determines what resources are needed to execute the task. A resource object has CPU, RAM, and disk attributes.

It also has the following optional attributes:

- `constraints`: This allows specification of dependence between processes of a task. Currently, only `order` constraint is supported that specifies the order in which to run the processes.
- `max_failures`: This controls after how many process failures, the task is marked as failed, which defaults to `1`.
- `max_concurrency`: This specifies the maximum number of processes Thermos can run concurrently in the task. Its default value is `0`, specifying infinite concurrency. This can be useful for executing expensive processes.
- `finalization_wait`: This controls the number of seconds to perform cleaning. After waiting for this amount of time, the process will be sent a SIGKILL signal.

A job object is a group of identical tasks that can be run on a cluster. The required attributes of a job object are:

- `task`: This specifies which task the job should bind to
- `role`: This is the user account used for running the job on Mesos
- `environment`: This denotes which environment the job should be submitted to. The typical values are devel, test, prod, and so on.
- `cluster`: This is the name of the Aurora cluster used for submitting a job. This can be any value defined in `/etc/aurora/clusters.json` or `~/.clusters.json`.

The following are the optional attributes of a job object:

Attribute	**Description**

`instances`	This is the number of instances/replicas required to run this job's task. This defaults to `1`.
`priority`	This is the task preemption priority. Higher values specify priority over lower values, thus jobs with higher values can preempt jobs with lower values. This defaults to `0`.
`production`	This checks whether the job is a production job. All production jobs have higher priority than all the nonproduction jobs. This defaults to `False`.
`update_config`	This is the object specifying the rolling update policy.
`health_check_config`	This is the object controlling the task's health check.
`constraints`	This is a Python map object that specifies the scheduling constraint of the job. This defaults to empty.
`contact`	This is the e-mail ID of the owner of the job.
`max_task_failures`	This is the maximum number of task failures, after which the job is marked as failed. This defaults to `1`. A value of `-1` indicates that infinite failures are allowed.
`service`	This checks whether the job is a service job. A service job runs continuously and is restarted after a successful or failed exit. This defaults to `False`.

Here is a complete configuration file for a very simple job, which outputs `Hello world`:

```
import os
hello_world_process = Process(name = 'hello_world', cmdline = 'echo
Hello world')

hello_world_task = Task(
  resources = Resources(cpu = 0.1, ram = 16 * MB, disk = 16 * MB),
  processes = [hello_world_process])

hello_world_job = Job(
  cluster = 'cluster1',
  role = os.getenv('USER'),
  task = hello_world_task)

jobs = [hello_world_job]
```

An Aurora client

Once we have a configuration file, we can use an Aurora client to launch and interact with the job. Aurora client commands are organized in various subcommands according to the operations:

Command	Subcommand	Description
job	`create`	This creates a job or a service.
	`list`	This lists jobs matching the `jobkey` or jobkey pattern.
	`inspect`	This parses the job configuration file and displays the information.
	`diff`	This compares a job configuration file with the running job.
	`status`	This gives status information about a scheduled job or group of jobs.
	`kill and killall`	This kills instances of the scheduled job.
	`update`	This starts a rolling upgrade of the running job.
	`cancel-update`	This cancels the update in progress.
	`restart`	This starts a rolling restart of the instances of the job.
task	`run`	This executes a given shell command on machines that are running instances of the job.
	`ssh`	This opens an SSH session to the machines that are running task instances.
sla	`get-task-up-count`	This prints the log-scale distribution of task uptime percentages. This specifies the duration.
	`get-job-uptime`	This prints the distribution of uptime percentiles of the job. This specifies a percentile to get the job uptime at a specified percentile.
quota	`get`	This prints quotas for a given role.

cron	start	This starts a cron job immediately, irrespective of its normal schedule.
	show	This shows the scheduling status of the cron job.
	schedule	This creates a cron schedule for a job. If the job already has an existing schedule, it will be replaced.
	deschedule	This removes a cron schedule for a job.
config	list	This lists all the jobs defined in the configuration file.
beta-update	start	This starts a scheduler-driven rolling upgrade on a running job.
	status	This displays a status of scheduler-driven updates.
	list	This lists all the scheduler-driven job updates in progress that match the query.
	pause	This pauses a scheduler-driven update.
	resume	This resumes a scheduler-driven update.
	abort	This aborts a scheduler-driven update that is in progress.

The `job` subcommand provides commands to interact with the `create` subcommand, which creates and runs an Aurora job. A job key is of the form `<cluster>/<role>/<env>/<jobname>` (for example, `cluster1/dev-team/devel/ad-exp`). This will output, among other information, the job URL:

```
ubuntu@master:~ $ aurora create jobKey conf.aurora
```

At the time of writing, the Aurora project does not provide a REST API for Aurora, but there are ongoing efforts in the Aurora community to provide a REST interface. However, the Aurora command-line client is very powerful. The Aurora client is very rich in terms of functionality and the preceding list consists of key commands. Also, each of the preceding commands supports a wide range of options. Aurora has a lot of other useful features, such as hooks that allow us to change or extend the behavior of Aurora API, performance monitoring, and web UI providing detailed information of a job and cluster to name but a few. An extensive coverage of all the Aurora features is

beyond the scope of this book, and Aurora documentation is a good source of information (http://aurora.apache.org/documentation/latest).

An Aurora example

An Aurora project has a few example Aurora job definitions in the `examples` directory. Let's run a simple Aurora job, which prints **hello world** every 10 seconds. Here is the `example/jobs/hello_world.aurora` file:

```
hello = Process(
  name = 'hello',
  cmdline = """
    while true; do
      echo hello world
      sleep 10
    done
  """)

task = SequentialTask(
  processes = [hello],
  resources = Resources(cpu = 1.0, ram = 128*MB, disk = 128*MB))

jobs = [Service(
  task = task, cluster = 'devcluster', role = 'www-data', environment
= 'prod', name = 'hello')]
```

The Aurora `create` command takes a `JobKey` and `.aurora` configuration file. As noted, a job key is of the form `<cluster>/<role>/<env>/<jobname>`; in our preceding example, (`/devcluster/www-data/devel/hello_world`). The following command will launch the hello service in the prod environment using the `hello_world.aurora` configuration file:

```
ubuntu@master:~ $ aurora create /devcluster/www-
data/devel/hello_world aurora/examples/jobs/hello_world.aurora
```

This will launch the hello world job and print the URL, where we can see the details of the job. We will be able to see the **hello world** output in the job output.

Aurora cron jobs

Aurora also supports the cron-style recurring execution of jobs. Aurora identifies the job as a cron job by the `cron_schedule` attribute in the job object. The value follows a restricted subset of crontab syntax and indicates the repetition schedule for the job. The `cron_collision_policy` field specifies the behavior when the previous run of the job has not finished at the time of the launch of the job. The `KILL_EXISTING` value will kill any old instances and will create a new instance with the current configuration, while `CANCEL_NEW` will cancel the new run of a job. `KILL_EXISTING` is the default policy. Note that Aurora will try any failed cron jobs `max_task_failures` times only. The Aurora service behavior of run until failure is achieved by setting `max_task_failures` to `-1`. The Aurora cron documentation (http://aurora.apache.org/documentation/latest/cron-jobs) provides more details on running cron jobs with Aurora.

Service discovery

When we are running multiple copies of your application, we should have the ability to discover where they are running and should be able to connect to them. This is especially important in case of failures, where Mesos relaunches services on other machines, but clients should still be able to connect. There are two popular ways to discover services:

- Domain Name Service based
- Specialized service discovery solutions (ZooKeeper, Consul, Serf, SkyDNS, and so on)

Domain Name Service (**DNS**) based discovery is very general and thus, it is easier to integrate with a lot of tools, while specialized solutions require understanding different framework-specific APIs. On the other hand, with DNS-based discovery methods, it might be harder to provide richer information about services, while specific solutions can provide extra information regarding service health and load. We will cover one DNS-based service discovery mechanism here—Mesos-DNS, but there are other service discovery solutions that integrate well with Mesos. One advantage Mesos-DNS has over generic DNS-based service discovery frameworks is that it can leverage Mesos capabilities without reinventing them. Also, starting from Mesos version 0.22, Mesos includes discovery information within tasks and executor messages that makes integration with service discovery tools even easier.

Mesos-DNS

Mesos-DNS (http://mesosphere.github.io/mesos-dns) is a DNS-based service discovery mechanism for Mesos. DNS is a hierarchical distributed naming mechanism used by the Internet. For example, a browser uses DNS to discover the IP address of a particular website. Fundamentally, DNS is designed to translate names to addresses. Similarly, Mesos-DNS translates names assigned to Mesos applications to the IP address and port where they are running.

Tip

At the time of writing, Mesos-DNS is in alpha release and still under active development.

Mesos-DNS has a very simple interface: it receives DNS requests and replies with DNS records. This design has a benefit that the working of Mesos-DNS is transparent to frameworks. Frameworks only use standard DNS queries to resolve names to addresses. Thus, as the cluster state changes, we can easily switch to some other mechanism to update DNS records, without making any change to the frameworks. Mesos-DNS has two primary components:

- DNS record generator: A record generator is responsible for generating DNS A and SRV records for all the running applications. It periodically queries the master for all the running applications and all the running tasks for each of the application. So, the record generator knows the status of the tasks as they start, finish, fail, or restart, and based on this, it generates or updates appropriate DNS records. The record generator keeps the latest status of various services in the form of DNS records.
- DNS resolver: A resolver is responsible for handling DNS requests and replying to them. A resolver replies directly to tasks running on Mesos and chooses a random external DNS server for external requests, similar to what a normal DNS server would do.

Currently, Mesos-DNS only support ANY, A, and SRV DNS records. For any other types of record in the mesos domain, it will return NXDOMAIN. This means that it cannot be used for reverse lookup, which requires PTR records. An A record associates a hostname to an IP address. Mesos-DNS generates an A record for the `task.framework.domain` hostname that provides the IP address of the specific slave, running the task launched by the framework. For example, other Mesos tasks can

discover the IP address for the `db` service launched by the Marathon framework by looking up for `db.marathon.mesos`. Also, note that SRV records are only generated for tasks that have been allocated a port through Mesos. In case the framework launches multiple tasks with the same name, Mesos-DNS will return multiple DNS records, one for each task and shuffles the order of the records randomly. This provides basic load balancing.

Mesos-DNS does not implement other features, such as health monitoring or life cycle management of applications. This makes Mesos-DNS architecture simple and stateless. It does not use a persistent storage, replication, or consensus. The stateless architecture makes it easy to scale in case of the large Mesos cluster. Mesos-DNS can be scaled very easily by launching multiple copies of Mesos-DNS. One consequence of this design is that Mesos-DNS is not fault-tolerant by itself, but relies on external entities for it. For example, launching Mesos-DNS as a Marathon or Aurora framework will monitor it, and it can be relaunched in case of a failure. Mesos-DNS will continue serving stale DNS records for existing services when the Mesos master is not reachable. Note that Mesos-DNS can deal with the Mesos master failovers because it will connect to the new Mesos master.

Installing Mesos-DNS

Mesos-DNS is written in Go and can be installed anywhere, as long it can reach the Mesos master. The following steps will install Mesos-DNS:

1. We need to install Go to be able to compile it. The following commands will install Go and set appropriate environment variables:

```
ubuntu@master:~ $ wget
https://storage.googleapis.com/golang/go1.4.linux-amd64.tar.gz
ubuntu@master:~ $ tar xzf go*
ubuntu@master:~ $ sudo mv go /usr/local/.
ubuntu@master:~ $ export PATH=$PATH:/usr/local/go/bin
ubuntu@master:~ $ export GOROOT=/usr/local/go
ubuntu@master:~ $ export PATH=$PATH:$GOROOT/bin
ubuntu@master:~ $ export GOPATH=$HOME/go
```

2. Now, let's install Mesos-DNS and the `go` libraries it depends on:

```
ubuntu@master:~ $ go get github.com/miekg/dns
ubuntu@master:~ $ go get github.com/mesosphere/mesos-dns
ubuntu@master:~ $ cd $GOPATH/src/github.com/mesosphere/mesos-dns
ubuntu@master:~ $ go build -o mesos-dns main.go
ubuntu@master:~ $ sudo mkdir /usr/local/mesos-dns
ubuntu@master:~ $ sudo mv mesos-dns /usr/local/mesos-dns
```

Mesos-DNS configuration

The configuration of Mesos-DNS is minimal. Mesos-DNS takes a JSON configuration file. By default, Mesos-DNS will look for the `config.json` file in the current directory. The configuration file path can also be specified through the `-config` command-line parameter. Mesos-DNS has the following configuration options:

- `masters`: This specifies a comma-separated list of addresses of Mesos masters.
- `refreshSeconds`: This specifies the frequency at which Mesos-DNS updates DNS records based on the information from the Mesos master. The default value is 60 seconds.
- `ttl`: This value specifies the time for which the DNS record, given by Mesos-DNS, is valid. This is used as a standard `ttl` field in the DNS record. The `ttl` value should be at least as large as `refreshSeconds`. The default value of `ttl` is 60 seconds. This means that, once a service has queried Mesos-DNS for address of a service, it can cache the result for 60 seconds before querying it again. Thus, the larger the `ttl` value, the smaller the load on Mesos-DNS. A larger `ttl` value also means that it will take longer to propagate the changes in service addresses to the clients using it.
- `domain`: This is the domain name of the Mesos cluster. The domain name can use a combination of upper and lowercase letters, numbers, and "-" if it is not the first or last character of a domain portion. "." can be used as a separator of the textual portions of the domain name. The default value is `mesos`.
- `port`: This specifies the port number that Mesos-DNS listens for the incoming DNS requests. The requests can be sent over TCP or UDP. The default value is 53.
- `resolvers`: This is a comma-separated list of IP addresses of the external DNS servers. These DNS servers are used to resolve all the DNS requests outside the "domain". We can use public DNS servers, such as 8.8.8.8 (provided by Google), or other site-specific DNS servers. If the resolvers are not set properly, tasks will not be able to resolve any external names.
- `timeout`: This value specifies the time taken to wait for the connection and response from external DNS resolvers.
- `email`: This specifies the e-mail address of the administrator who maintains the Mesos domain. It's in the mailbox-name.domain format.

Here is an example of a configuration file that can be found in the `build` directory:

```
ubuntu@master:~ $ cat /usr/local/mesos-dns/config.json
{
```

```
  "masters": ["zk://localhost:5050"],
  "refreshSeconds": 60,
  "ttl": 60,
  "domain": "mesos",
  "port": 53,
  "resolvers": ["169.254.169.254","10.0.0.1"],
  "timeout": 5
  "resolvers": "10.101.160.16",
  "listener": "root.mesos-dns.mesos"
}
```

Running Mesos-DNS

The following command will start `mesos-dns` in the background, using `config.json` in the current directory. Since `mesos-dns` binds to a privileged port 53, we have to start `mesos-dns` as a root user:

```
ubuntu@master:~ $ sudo mesos-dns &
```

As noted earlier, it's recommended that you launch `mesos-dns` using external launchers (Marathon/Aurora/runit, and so on) for fault tolerance. Once `mesos-dns` is running, we can check it using dig to query any existing running services. Note that `mesos-dns` needs to be restarted for any changes to take effect in the configuration. `mesos-dns` has a verbose mode that can be very useful while debugging. To enable verbose mode, start the `mesos-dns` with `-v flag`.

For Mesos tasks, to be able to use Mesos-DNS to resolve addresses of services running on Mesos, Mesos-DNS must be a primary DNS server on the slaves. This can be achieved by adding the addresses of hosts running `mesos-dns` as name servers on every slave. For example, if the `mesos-dns` is running on machines with `10.2.3.4` and `10.2.3.5` IP addresses, adding the following lines to `/etc/resolv.conf` at the top of every node will make sure that tasks launched on slaves contact `mesos-dns` first to resolve service addresses. No changes are required on the master:

```
ubuntu@master:~ $ cat /etc/resolv.conf
nameserver 10.2.3.4
nameserver 10.2.3.5
...
```

Packaging

Modern application code does not operate in isolation and depends on other applications and libraries. Thus, the deployment has to make sure that these dependencies are met while deploying. Jar (Java Archive), Python eggs, and static binaries are all forms of packaging. There is an on-going effort to port Google's Kubernetes project (http://kubernetes.io) to Mesos (https://github.com/mesosphere/kubernetes-mesos). Kubernetes also allow running services at scale. Recently, Docker (https://www.docker.com) has emerged as a preferred mechanism for packaging applications in a portable manner. We will see Docker Mesos integration in Chapter 6, *Understanding Mesos Internals*.

Summary

In this chapter, we covered Marathon and Aurora scheduler frameworks, allowing long-running services on the Mesos cluster. We also looked at Chronos enabling recurring jobs on Mesos. Note that there are other frameworks for deploying services on Mesos, which we could not cover here, most notably, the singularity framework (https://github.com/HubSpot/Singularity) developed by HubSpot. These frameworks bring Mesos' vision of treating the entire data center as a single computer one step closer.

This chapter also concludes our coverage of various frameworks on Mesos. We covered a wide variety of frameworks aimed at solving various business problems, but the list of frameworks covered is by no means an exhaustive list of Mesos frameworks. A complete list of frameworks is available at https://github.com/dharmeshkakadia/awesome-mesos. In the next chapter, we will delve deeper into the internal workings of Mesos.

Chapter 6. Understanding Mesos Internals

This chapter explains how Mesos works internally in detail. We will start off by understanding Mesos architecture, cluster scheduling and fairness concepts, and we will move on towards resource isolation and fault tolerance implementation in Mesos. In this chapter, we will cover the following topics:

- The Mesos architecture
- Resource allocation
- Resource isolation
- Fault tolerance
- Extending Mesos

The Mesos architecture

Modern organizations have a lot of different kinds of applications for different business needs. Modern applications are distributed and they are deployed across commodity hardware. Organizations today run different applications in siloed environments, where separate clusters are created for different applications. This static partitioning of cluster leads to low utilization, and all the applications will duplicate the effort of dealing with distributed infrastructures. Not only is this a wasted effort, but it also undermines the fact that distributed systems are hard to build and maintain. This is challenging for both developers and operators. For developers, it is a challenge to build applications that scale elastically and can handle faults that are inevitable in large-scale environment. Operators, on the other hand, have to manage and scale all of these applications individually in siloed environments.

The preceding situation is like trying to develop applications without having an operating system and managing all the devices in a computer. Mesos solves the problems mentioned earlier by providing a data center kernel. Mesos provides a higher-level abstraction to develop applications that treat distributed infrastructure just like a large computer. Mesos abstracts the hardware infrastructure away from the applications.

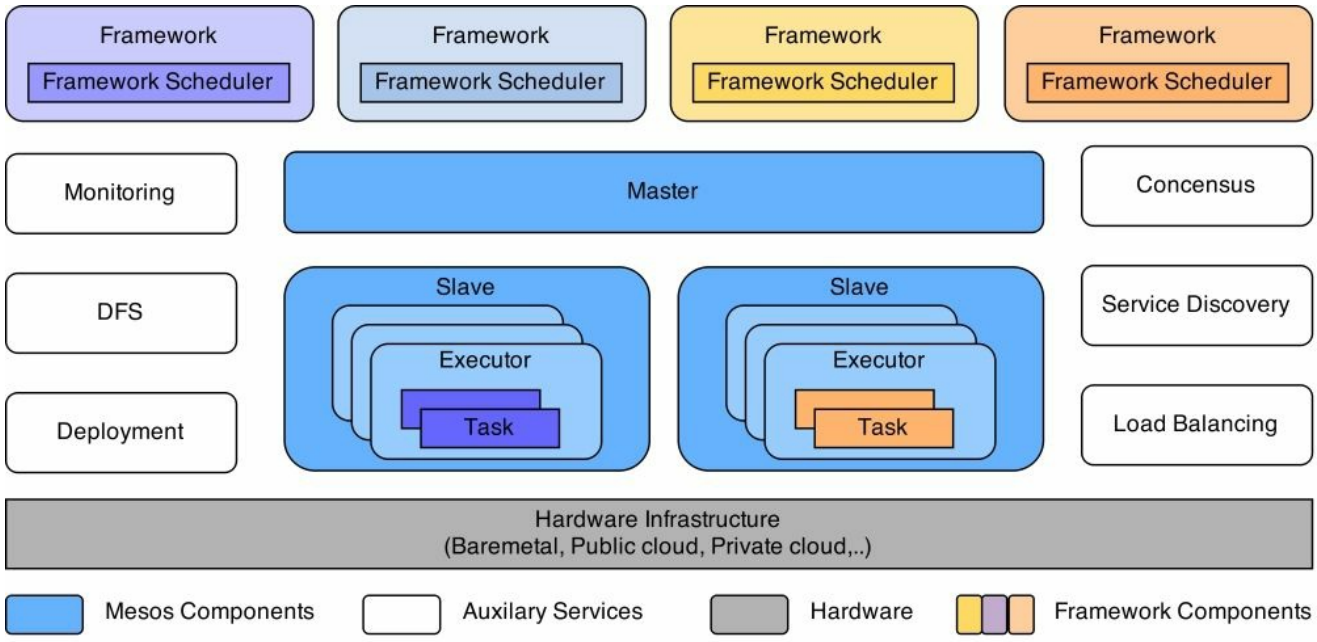
- Mesos makes developers more productive by providing an SDK to easily write data center scale applications. Now, developers can focus on their application logic and do not have to worry about the infrastructure that runs it. Mesos SDK provides primitives to build large-scale distributed systems, such as resource allocation, deployment, monitoring and isolation. They only need to know and implement what resources are needed, and not how they get the resources. Mesos allows them to treat the data center just as a computer.
- Mesos makes the infrastructure operations easier by providing elastic infrastructure. Mesos aggregates all the resources in a single shared pool of resources and avoids static partitioning. This makes it easy to manage and increases the utilization.

As we saw in Chapter 1, *Running Mesos*, the data center kernel has to provide resource allocation, isolation, and fault tolerance in a scalable, robust, and extensible way. We will discuss how Mesos fulfills these requirements, as well as some other important considerations of modern data center kernel:

- Scalability: The kernel should be scalable in terms of the number of machines and

number of applications. As the number of machines and applications increase, the response time of the scheduler should remain acceptable.

- Flexibility: The kernel should support a wide range of applications. It should be able to accommodate diverse frameworks currently running on the cluster and future frameworks as well. It should also be able to cope up with the heterogeneity in the hardware, as most clusters are built over time and have a variety of hardware running.
- Maintainability: The kernel would be one of the very important pieces of modern infrastructure. As the requirements evolve, It should be able to accommodate new requirements.
- Utilization and dynamism: The kernel should adapt to the changes in resource requirements and available hardware resources and utilize resources in an optimal manner.
- Fairness: The kernel should be fair in allocating resources to the different users and/or frameworks. We will see what it means to be fair in detail in the next section.



The design philosophy behind Mesos is to define a minimal interface to enable efficient resource sharing across frameworks and defer the task scheduling and execution to the frameworks. This allows the frameworks to implement diverse approaches toward scheduling and fault tolerance. It also makes the Mesos core simple, and the frameworks and core can evolve independently. The preceding figure shows the overall architecture (http://mesos.apache.org/documentation/latest/mesos-architecture) of a Mesos cluster. It

has the following entities:

- Mesos masters
- Mesos slaves
- Frameworks
- Communication
- Auxiliary services

We will describe each of these entities and their role, followed by how Mesos implements different requirements of the data center kernel.

Mesos slave

The Mesos slaves are responsible for executing tasks from frameworks using the resources they have. The slave has to provide proper isolation while running multiple tasks. The isolation mechanism should also make sure that the tasks get resources that they are promised, and not more or less.

The resources on slaves that are managed by Mesos can be described using slave resources and slave attributes. Resources are elements of slaves that can be *consumed* by a task, while we use attributes to tag slaves with some information. Slave resources are managed by the Mesos master and are allocated to different frameworks. Attributes identify something about the node, such as the slave having a specific OS or software version, it's part of a particular network, or it has a particular hardware, and so on. The attributes are simple key-value pairs of strings that are passed along with the offers to frameworks. Since attributes cannot be *consumed* by a running task, they will always be offered for that slave. Mesos doesn't understand the slave attribute, and interpretation of the attributes is left to the frameworks. More information about resource and attributes in Mesos can be found at https://mesos.apache.org/documentation/attributes-resources.

A Mesos resource or attribute can be described as one of the following types:

- Scalar values are floating numbers
- Range values are a range of scalar values; they are represented as [minValue-maxValue]
- Set values are arbitrary strings
- Text values are arbitrary strings; they are applicable only to attributes

Names of the resources can be an arbitrary string consisting of alphabets, numbers, "-", "/", ".", "-". The Mesos master handles the `cpus`, `mem`, `disk`, and `ports` resources in a special way. A slave without the `cpus` and `mem` resources will not be advertised to the frameworks. The `mem` and `disk` scalars are interpreted in MB. The `ports` resource is represented as ranges. The list of resources a slave has to offer to various frameworks can be specified as the `--resources` flag. Resources and attributes are separated by a semicolon. For example:

```
--resources='cpus:30;mem:122880;disk:921600;ports:[21000-29000];bugs:
{a,b,c}'
--attributes='rack:rack-2;datacenter:europe;os:ubuntuv14.4'
```

This slave offers 30 `cpus`, 120 GB `mem`, 900 GB `disk`, `ports` from 21000 to 29000,

and have `bugs` resource with values a, `b` and `c`. The slave has three attributes: `rack` with value `rack-2`, `datacenter` with value `europe`, and `os` with value `ubuntu14.4`.

Mesos does not yet provide direct support for GPUs, but does support custom resource types. This means that if we specify `gpu(*):8` as part of `--resources`, then it will be part of the resource that offers to frameworks. Frameworks can use it just like other resources. Once some of the GPU resources are in use by a task, only the remaining resources will be offered. Alternately, we can also specify which slaves have GPUs using attributes, such as `--attributes="hasGpu:true"`. Mesos does not yet have support for GPU isolation, but it can be extended by implementing a custom isolator.

Mesos master

The Mesos master is primarily responsible for allocating resources to different frameworks and managing the task life cycle for them. The Mesos master implements fine-grained resource sharing using resource offers. The Mesos master acts as a resource broker for frameworks using pluggable policies. The master decides to offer cluster resources to frameworks in the form of resource offers based on them.

Resources offer represents a unit of allocation in the Mesos world. It's a vector of resource available on a node. An offer represents some resources available on a slave being offered to a particular framework.

Frameworks

Distributed applications that run on top of Mesos are called frameworks. Frameworks implement the domain requirements using the general resource allocation API of Mesos. A typical framework wants to run a number of tasks. Tasks are the consumers of resources and they do not have to be the same. A framework in Mesos consists of two components: a framework scheduler and executors. Framework schedulers are responsible for coordinating the execution. An executor provides the ability to control the task execution. Executors can realize a task execution in many ways. An executor can choose to run multiple tasks, by spawning multiple threads, in an executor, or it can run one task in each executor. Apart from the life cycle and task management-related functions, the Mesos framework API also provides functions to communicate with framework schedulers and executors. We will see the details of the framework API in Chapter 7, *Developing Frameworks on Mesos*.

Communication

Mesos currently uses an HTTP-like wire protocol to communicate with the Mesos components. Mesos uses the `libprocess` library to implement the communication that is located in `3rdparty/libprocess`. The `libprocess` library provides asynchronous communication between processes. The communication primitives have semantics similar to actor message passing. The `libprocess` messages are immutable, which makes parallelizing the `libprocess` internals easier. Mesos communication happens along the following APIs:

- Scheduler API: This is used to communicate with the framework scheduler and master. The internal communication is intended to be used only by the SchedulerDriver API.
- Executor API: This is used to communicate with an executor and the Mesos slave.
- Internal API: This is used to communicate with the Mesos master and slave.
- Operator API: This is the API exposed by Mesos for operators and is used by web UI, among other things. Unlike most Mesos API, the operator API is a synchronous API.

To send a message, the actor does an HTTP POST request. The path is composed by the name of the actor followed by the name of the message. The `User-Agent` field is set to "`libprocess/...`" to distinguish from the normal HTTP requests. The message data is passed as the body of the HTTP request. Mesos uses protocol buffers to serialize all the messages (defined in `src/messages/messages.proto`). The parsing and interpretation of the message is left to the receiving actor.

Here is an example header of a message sent to `master` to register the framework by `scheduler(1)` running at `10.0.1.7:53523` address:

```
POST /master/mesos.internal.RegisterFrameworkMessage HTTP/1.1
User-Agent: libprocess/scheduler(1)@10.0.1.7:53523
```

The reply message header from the master that acknowledges the framework registration might look like this:

```
POST /scheduler(1)/mesos.internal.FrameworkRegisteredMessage HTTP/1.1
User-Agent: libprocess/master@10.0.1.7:5050
```

At the time of writing, there is a very early discussion about rewiring the Mesos Scheduler API and Executor API as a pure HTTP API (https://issues.apache.org/jira/browse/MESOS-2288). This will make the API standard

and integration with Mesos for various tools much easier without the need to be dependent on native `libmesos`. Also, there is an ongoing effort to convert all the internal messages into a standardized JSON or protocol buffer format (https://issues.apache.org/jira/browse/MESOS-1127).

Auxiliary services

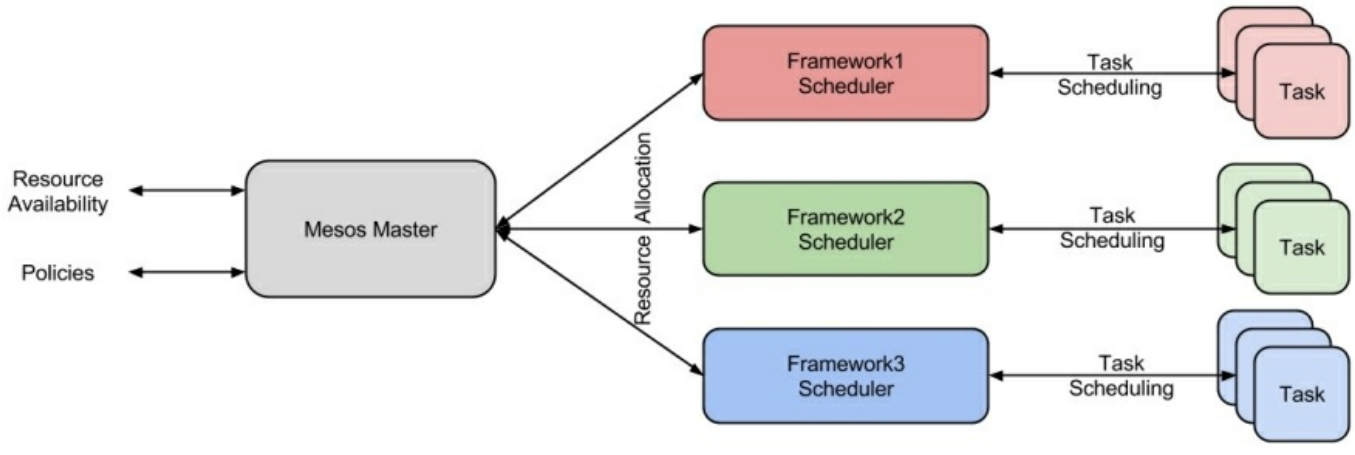
Apart from the preceding main components, a Mesos cluster also needs some auxiliary services. These services are not part of Mesos itself, and are not strictly required, but they form a basis for operating the Mesos cluster in production environments. These services include, but are not limited to, the following:

- Shared filesystem: Mesos provides a view of the data center as a single computer and allows developers to develop for the data center scale application. With this unified view of resources, clusters need a shared filesystem to truly make the data center a computer. HDFS, **NFS** (**Network File System**), and Cloud-based storage options, such as S3, are popular among various Mesos deployments.
- Consensus service: Mesos uses a consensus service to be resilient in face of failure. Consensus services, such as ZooKeeper or etcd, provide a reliable leader election in a distributed environment.
- Service fabric: Mesos enables users to run a number of frameworks on unified computing resources. With a large number of applications and services running, it's important for users to be able to connect to them in a seamless manner. For example, how do users connect to Hive running on Mesos? How does the Ruby on Rails application discover and connect to the MongoDB database instances when one or both of them are running on Mesos? How is the website traffic routed to web servers running on Mesos? Answering these questions mainly requires service discovery and load balancing mechanisms, but also things such as IP/port management and security infrastructure. We are collectively referring to these services that connect frameworks to other frameworks and users as service fabric.
- Operational services: Operational services are essential in managing operational aspects of Mesos. Mesos deployments and upgrades, monitoring cluster health and alerting when human intervention is required, logging, and security are all part of the operational services that play a very important role in a Mesos cluster. We will see the operational details of Mesos in Chapter 8, *Administering Mesos*.

Resource allocation

As a data center kernel, Mesos serves a large variety of workloads and no single scheduler will be able to satisfy the needs of all different frameworks. For example, the way in which a real-time processing framework schedules its tasks will be very different from how a long running service will schedule its task, which, in turn, will be very different from how a batch processing framework would like to use its resources. This observation leads to a very important design decision in Mesos: separation of resource allocation and task scheduling. Resource allocation is all about deciding who gets what resources, and it is the responsibility of the Mesos master. Task scheduling, on the other hand, is all about how to use the resources. This is decided by various framework schedulers according to their own needs. Another way to understand this would be that Mesos handles coarse-grain resource allocation across frameworks, and then each framework does fine-grain job scheduling via appropriate job ordering to achieve its needs.

The Mesos master gets information on the available resources from the Mesos slaves, and based on resource policies, the Mesos master offers these resources to different frameworks. Different frameworks can choose to accept or reject the offer. If the framework accepts a resource offer, the framework allocates the corresponding resources to the framework, and then the framework is free to use them to launch tasks. The following image shows the high-level flow of Mesos resource allocation.



Mesos two level scheduler

Here is the typical flow of events for one framework in Mesos:

1. The framework scheduler registers itself with the Mesos master.
2. The Mesos master receives the resource offers from slaves. It invokes the allocation module and decides which frameworks should receive the resource offers.
3. The framework scheduler receives the resource offers from the Mesos master.
4. On receiving the resource offers, the framework scheduler inspects the offer to decide whether it's suitable. If it finds it satisfactory, the framework scheduler accepts the offer and replies to the master with the list of executors that should be run on the slave, utilizing the accepted resource offers. Alternatively, the framework can reject the offer and wait for a better offer.
5. The slave allocates the requested resources and launches the task executors. The executor is launched on slave nodes and runs the framework's tasks.
6. The framework scheduler gets notified about the task's completion or failure. The framework scheduler will continue receiving the resource offers and task reports and launch tasks as it sees fit.
7. The framework unregisters with the Mesos master and will not receive any further resource offers. Note that this is optional and a long running service, and meta-framework will not unregister during the normal operation.

Because of this design, Mesos is also known as a two-level scheduler. Mesos' two-level scheduler design makes it simpler and more scalable, as the resource allocation process does not need to know how scheduling happens. This makes the Mesos core more stable and scalable. Frameworks and Mesos are not tied to each other and each can iterate independently. Also, this makes porting frameworks easier, as we will see in the next chapter.

The choice of a two-level scheduler means that the scheduler does not have a global knowledge about resource utilization and the resource allocation decisions can be nonoptimal. One potential concern could be about the preferences that the frameworks have about the kind of resources needed for execution. Data locality, special hardware, and security constraints can be a few of the constraints on which tasks can run. In the Mesos realm, these preferences are not explicitly specified by a framework to the Mesos master, instead the framework rejects all the offers that do not meet its constraints.

The Mesos scheduler

Mesos was the first cluster scheduler to allow the sharing of resources to multiple frameworks. Mesos resource allocation is based on online **Dominant Resource Fairness** (**DRF**) called HierarchicalDRF. In a world where resources are statically partitioned based on a single resource, fairness is easy to define: divide the cluster based on this single resource (say, CPU) equally among all the users. DRF extends this concept of fairness to multi-resource settings without the need for static partitioning. Resource utilization and fairness are equally important, and often conflicting, goals for a cluster scheduler. The fairness of resource allocation is important in a shared environment, such as data centers, to ensure that all the users/processes of the cluster get nearly an equal amount of resources.

Min-max fairness provides a well-known mechanism to share a single resource among multiple users. Min-max fairness algorithm maximizes the minimum resources allocated to a user. In its simplest form, it allocates *1/Nth* of the resource to each of the users. The weighted min-max fairness algorithm can also support priorities and reservations. Min-max resource fairness has been a basis for many well-known schedulers in operating systems and distributed frameworks, such as Hadoop's fair scheduler (http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html), capacity scheduler (https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html), Quincy scheduler (http://dl.acm.org/citation.cfm?id=1629601), and so on. However, it falls short when the cluster has multiple types of resources, such as the CPU, memory, disk, and network. When jobs in a distributed environment use different combinations of these resources to achieve the outcome, the fairness has to be redefined. For example, the two requests `<1 CPU, 3 GB>` and `<3 CPU, 1 GB>` come to the scheduler. How do they compare and what is the fair allocation?

DRF generalizes the min-max algorithm for multiple resources. A user's dominant resource is the resource for which the user has the biggest share. For example, if the total resources are `<8 CPU, 5 GB>`, then for the user allocation of `<2 CPU, 1 GB>`, the user's dominant share is maximumOf(2/8,1/5) means CPU since 2/8 > 1/5. A user's dominant share is the fraction of the dominant resource that's allocated to the user. In our example, it would be 25 percent (2/8). DRF applies the min-max algorithm to the dominant share of each user. It has many provable properties:

1. Strategy proofness: A user cannot gain any advantage by lying about the demands.

2. Sharing incentive: DRF has a minimum allocation guarantee for each user, and no user will prefer exclusive partitioned cluster of size `1/N` over DRF allocation.
3. Single resource fairness: In case of only one resource, DRF is equivalent to the min-max algorithm.
4. Envy free: Every user prefers his allocation over any other allocation of other users. This also means that the users with the same requests get equivalent allocations.
5. Bottleneck fairness: When one resource becomes a bottleneck, and each user has a dominant demand for it, DRF is equivalent to min-max.
6. Monotonicity: Adding resources or removing users can only increase the allocation of the remaining users.
7. Pareto efficiency: The allocation achieved by DRF will be pareto efficient, so it would be impossible to make a better allocation for any user without making allocation for some other user worse.

We will not further discuss DRF but will encourage you to refer to the DRF paper for more details at http://static.usenix.org/event/nsdi11/tech/full_papers/Ghodsi.pdf.

Mesos uses role specified in `FrameworkInfo` for resource allocation decision. A role can be per user or per framework or can be shared by multiple users and frameworks. If it's not set, Mesos will set it to the current user that runs the framework scheduler.

Weighted DRF

DRF calculates each role's dominant share and allocates the available resources to the user with the smallest dominant share. In practice, an organization rarely wants to assign resources in a complete fair manner. Most organizations want to allocate resources in a weighted manner, such as 50 percent resources to ads team, 30 percent to QA, and 20 percent to R&D teams. To satisfy this functionality, Mesos implements weighted DRF, where masters can be configured with weights for different roles. When weights are specified, a client's DRF share will be divided by the weight. For example, a role that has a weight of two will be offered twice as many resources as a role with weight of one.

Mesos can be configured to use weighted DRF using the `--weights` and `--roles` flags on the master startup. The `--weights` flag expects a list of role/weight pairs in the form of `role1=weight1` and `role2=weight2`. Weights do not need to be integers.

Tip

We must provide weights for each role that appear in `--roles` on the master startup.

Reservation

One of the other most requested feature is the ability to reserve resources. For example, persistent or stateful services, such as memcache, or a database running on Mesos, would need a reservation mechanism to avoid being negatively affected on restart. Without reservation, memcache is not guaranteed to get a resource offer from the slave, which has all the data and would incur significant time in initialization which can result in downtime for the service. Reservation can also be used to limit the resource per role.

Tip

Reservation provides guaranteed resources for roles, but improper usage might lead to resource fragmentation and lower utilization of resources.

Note that all the reservation requests go through a Mesos authorization mechanism to ensure that the operator or framework requesting the operation has the proper privileges. Reservation privileges are specified to the Mesos master through ACL along with the rest of the ACL configuration. Mesos supports the following two kinds of reservation:

- Static reservation
- Dynamic reservation

Static reservation

In static reservation, resources are reserved for a particular role. If a resource is reserved to role A, then only frameworks with role A are eligible to get an offer for that resource. Static reservation is typically managed by operators using the `--resources` flag on the slave. The flag expects a list of `name(role):value` for different resources

Any resources that do not include a role or resources that are not included in the `--resources` flag will be included in the default role (default *). For example, `--resources="cpus:4;mem:2048;cpus(ads):8;mem(ads):4096"` specifies that the slave has 8 CPUs and 4096 MB memory reserved for "ads" role and has 4 CPUs and 2048 MB memory unreserved. The restart of the slave after removing the checkpointed state is required to change static reservation

Tip

Nonuniform static reservation across slaves can quickly become difficult to manage.

Dynamic reservation

Dynamic reservation allows operators and frameworks to manage reservation more dynamically. Frameworks can use dynamic reservations to reserve offered resources, allowing those resources to only be reoffered to the same framework.

Note

At the time of writing, dynamic reservation is still being actively developed and is targeted toward the next release of Mesos (https://issues.apache.org/jira/browse/MESOS-2018).

When asked for a reservation, Mesos will try to convert the unreserved resources to reserved resources. On the other hand, during the unreserve operation, the previously reserved resources are returned to the unreserved pool of resources.

To support dynamic reservation, Mesos allows a sequence of `Offer::Operations` to be performed as a response to accepting resource offers. A framework manages reservation by sending `Offer::Operations::Reserve` and `Offer::Operations::Unreserve` as part of these operations, when receiving resource offers. For example, consider the framework that receives the following resource offer with 32 CPUs and 65536 MB memory:

```
{
  "id" : <offer_id>,
  "framework_id" : <framework_id>,
  "slave_id" : <slave_id>,
  "hostname" : <hostname>,
  "resources" : [
    {
      "name" : "cpus",
      "type" : "SCALAR",
      "scalar" : { "value" : 32 },
      "role" : "*",
    },
    {
      "name" : "mem",
      "type" : "SCALAR",
      "scalar" : { "value" : 65536 },
      "role" : "*",
    }
  ]
}
```

The framework can decide to reserve 8 CPUs and 4096 MB memory by sending the `Operation::Reserve` message with `resources` field with the desired resources state:

```
[
  {
    "type" : Offer::Operation::RESERVE,
    "resources" : [
      {
        "name" : "cpus",
        "type" : "SCALAR",
        "scalar" : { "value" : 8 },
        "role" : <framework_role>,
        "reservation" : {
          "framework_id" : <framework_id>,
          "principal" : <framework_principal>
        }
      }
      {
        "name" : "mem",
        "type" : "SCALAR",
        "scalar" : { "value" : 4096 },
        "role" : <framework_role>,
        "reservation" : {
          "framework_id" : <framework_id>,
          "principal" : <framework_principal>
        }
      }
    ]
  }
]
```

After a successful execution, the framework will receive resource offers with reservation. The next offer from the slave might look as follows:

```
{
  "id" : <offer_id>,
  "framework_id" : <framework_id>,
  "slave_id" : <slave_id>,
  "hostname" : <hostname>,
  "resources" : [
    {
      "name" : "cpus",
      "type" : "SCALAR",
      "scalar" : { "value" : 8 },
      "role" : <framework_role>,
      "reservation" : {
        "framework_id" : <framework_id>,
        "principal" : <framework_principal>
```

```
      }
    },
    {
      "name" : "mem",
      "type" : "SCALAR",
      "scalar" : { "value" : 4096 },
      "role" : <framework_role>,
      "reservation" : {
        "framework_id" : <framework_id>,
        "principal" : <framework_principal>
      }
    },
    {
      "name" : "cpus",
      "type" : "SCALAR",
      "scalar" : { "value" : 24 },
      "role" : "*",
    },
    {
      "name" : "mem",
      "type" : "SCALAR",
      "scalar" : { "value" : 61440 },
      "role" : "*",
    }
  ]
}
```

As shown, the framework has 8 CPUs and 4096 MB memory reserved resources and 24 CPUs and 61440 MB memory unreserved in the resource offer. The unreserve operation opposite to reserve. The framework on receiving the resource offer can send the unreserve operation message, and subsequent offers will not have reserved resources.

The operators can use `/reserve` and `/unreserve` HTTP endpoints of the operator API to manage the reservation. The operator API allows operators to manually change the reservation configuration without the need of slave restart. For example, the following command will reserve 4 CPUs and 4096 MB memory on slave1 for role1 with the operator authentication `principal ops`:

```
ubuntu@master:~ $ curl -d slaveId=slave1 -d resources="{
        {
          "name" : "cpus",
          "type" : "SCALAR",
          "scalar" : { "value" : 4 },
          "role" : "role1",
          "reservation" : {
            "principal" : "ops"
```

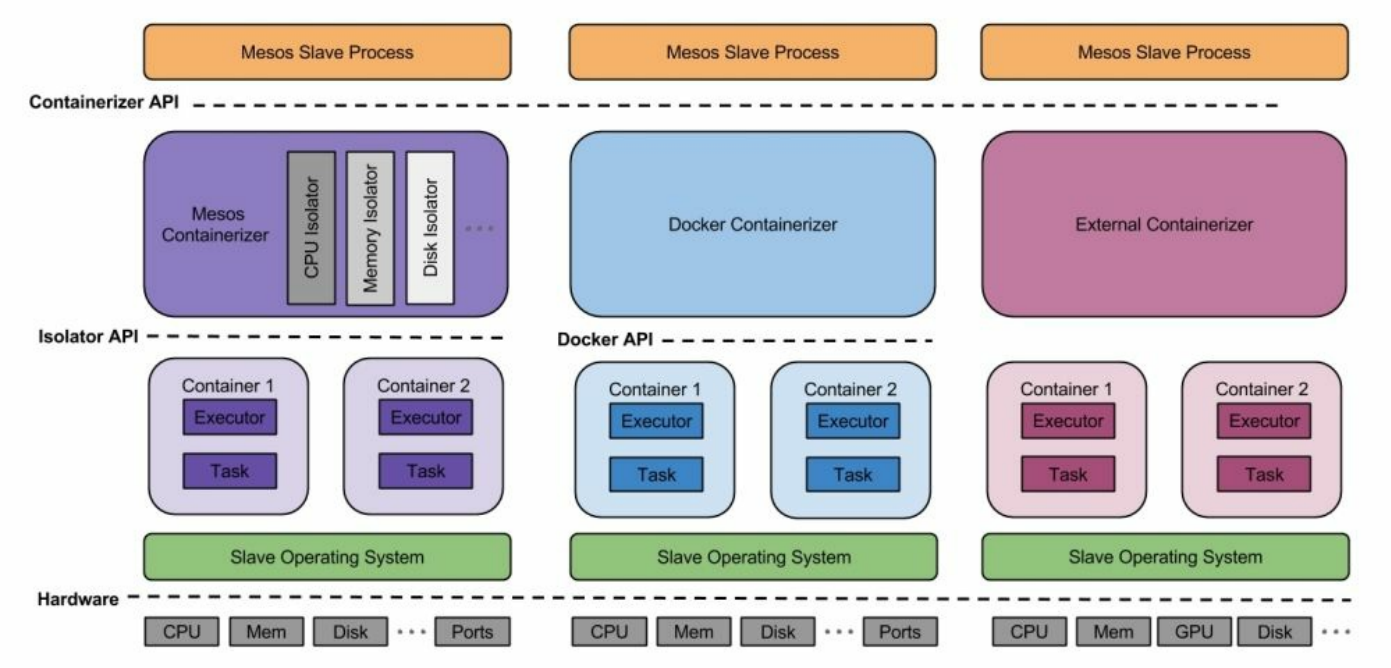
```
      }
    },
    {
      "name" : "mem",
      "type" : "SCALAR",
      "scalar" : { "value" : 4096 },
      "role" : "role1",
      "reservation" : {
        "principal" : "ops"
      }
    },
  }"
  -X POST http://master:5050/master/reserve
```

Before we end this discussion on resource allocation, it would be important to note that the Mesos community continues to innovate on the resource allocation front by incorporating interesting ideas, such as oversubscription (https://issues.apache.org/jira/browse/MESOS-354), from academic literature and other systems.

Resource isolation

Mesos provides various isolation mechanisms on slaves for sandboxing different tasks. The allocation of resources to one framework/job or user should not have any unintended effects on the running jobs. Containers act as lightweight virtual machines, providing the necessary isolation mechanism without the overhead of virtual machines. Using containers, we can limit the amount of resources that the process and all its child processes can access. To explain the detailed working of containers is beyond the scope of this book. However, to understand Mesos isolation mechanism, it is sufficient to assume that the containers provide a set of features that provide resource isolation.

Mesos resource isolation mechanism has a pluggable design and has evolved a lot since the creation of the project. The Mesos slave uses containerizer to provide an isolated environment to run an executor and its tasks. The following figure shows the different isolation mechanisms used by slaves.



Mesos slaves with different isolation mechanisms

The goal of the containerizer API is to support a wide range of containerizer implementations. This means that we can provide our own containerizers and isolators. When a slave process starts, we can specify the containerizer to use for launching

containers and a set of isolators to enforce the resource constraints. The containerizer interface has details about the current containerizer API. The containerizer API is defined in `src/slave/containerizer/containerizer.hpp`.

Mesos containerizer

The Mesos containerizer is an internal containerizer implementation (http://mesos.apache.org/documentation/latest/mesos-containerizer). It provides two types of isolators: process-based isolation, which is the basic isolation on POSIX-based systems, and cgroups-based isolation relies on cgroup (also known as control group) features available in Linux kernel. Cgroups-based isolation provides better resource isolation and accounts for CPU and memory, and also provides features such as checkpointing. The isolator API allows us to control what isolation mechanism is used by a slave for isolation. Note that the isolator API is important only in the case of `MesosContainerizer`, as external containerizers may use their own mechanism for isolation. It is composable so that operators can pick and choose different isolators. Apart from cgroups-based CPU and memory isolators, `MesosContainerizer` provides disk isolator, SharedFileSystem isolator, and PID namespace isolator.

Disk space isolation has been supported since the initial versions of Mesos. Mesos allows scheduling of disk resources. Starting from Version 0.23, Mesos operators can also enforce the disk quota using an isolator. It periodically inspects the task disk usage and can kill the tasks whose usage exceeds their share. It can be enabled by including `posix/disk` in the `--isolation` flag and using the `--enforce_container_disk_quota` flag on slaves. By default, the disk quota is not enforced and the task is not killed on exceeding its share. The POSIX disk isolator reports the disk usage statistics on the `/monitor/statistics.json` endpoint by periodically running the `du` command. This interval is controlled by the `--container_disk_watch_interval` flag on the slave. The default interval is 15 seconds.

Note

At the time of writing, Mesos support for persistent storages is continually being improved. You can track the progress at https://issues.apache.org/jira/browse/MESOS-1554.

The `ShreadFilesystem` isolator can be used to modify each container's view of the shared filesystem. The modifications can be specified in `ContainerInfo` as part of `ExecutorInfo` by the framework or via the `--default_container_info` slave flag. `ContainerInfo` specifies volumes that maps parts of the shared filesystem (`host_path`) to the container's view of the filesystem (`container_path`). Volumes can be mounted with read/write or read-only permissions. If the `host_path` is absolute, and

the filesystem subtree rooted at `host_path` will be accessible under `container_path` for each container.

If `host_path` is relative, then it is considered as a directory relative to the executor's work directory. The directory will be created and will have the same permissions as the corresponding directory in the shared filesystem. The primary use-case for the `SharedFilesystem` isolator is to make parts of the shared filesystem private to each container. For example, a private `/tmp` directory in each container can be achieved with `host_path="tmp"` and `container_path="/tmp"`. This will create the `tmp` directory inside the executor's work directory with the permissions mode 1777 and mount it as `/tmp` inside the container. This is transparent to processes running inside the container. Containers will not be able to see the host's `/tmp` or any other container's `/tmp`.

The PID namespace isolator can be used to isolate each container in a separate PID namespace. This limits the visibility of the processes spawned in a container. Thus, a process from one container cannot see or signal processes in another container. This also avoids the problem of unclean termination, where the termination of a leading process in a PID namespace does not kill other processes in a namespace.

Docker containerizer

Docker (https://www.docker.com) is an open platform used to build, ship, and run applications. It makes it easy to assemble applications from its components and provides a high-level API to run a lightweight Docker container in a portable manner. Docker container can use cgroups, LXC, OpenVZ, and kernel namespaces isolation. It is gaining widespread popularity as a way of packaging applications. Mesos and Docker makes a great combination as Docker provides a way of packaging applications in containers, and Mesos runs containers at scale. Starting from Mesos 0.20, Docker is a first-class citizen of Mesos (http://mesos.apache.org/documentation/latest/docker-containerizer) and has a native implementation of Docker built into Mesos, which integrates with Mesos without any external dependencies.

Note

Mesos 0.19.0 supports Docker as an external containerizer. In this mode, Mesos doesn't know anything about Docker, and all the Docker-specific features are delegated to other sub-processes.

`DockerContainerizer` primarily translates the task or executor launch and destroy calls to the appropriate Docker CLI commands. `DockerContainerizer` separates the `docker run` argument names so that, in the future, Mesos can transparently switch to the Docker remote API. Following are the steps taken when using the Docker containerizers:

1. The task launch will fetch all the files specified in `CommandInfo` in the sandbox.
2. Performs a Docker pull on the remote image repository.

   ### Tip

   This may take time based on the image size and network connection. If this takes longer than the `executor_registration_timeout` value (default 1 minute) on the slave, the containerizer will fail. In this case, we might consider setting `--executor_registration_timeout` to a higher value.

   ### Note

   Starting from Mesos-0.20.1, the need to do a Docker pull on every container run is removed, and Docker pull will be performed only when the image contains `tag :latest`. This avoids cases when the task launch will fail because of a failure to

connect to the Docker remote image repository.

3. Run the Docker image with the DockerExecutor. It also maps the sandbox directory to the Docker container and sets the directory mapping to the `MESOS_SANDBOX` environment variable.
4. Stream the Docker logs to the `stdout` and `stderr` files in the sandbox.
5. On container exit or containerizer destroy, stop and remove the Docker container.

To enable Docker, we need to install Docker CLI client (Version >= 1.0.0) on slaves, and start Mesos slaves with Docker included in the `--containerizers` option. Docker containerizer allows us to specify the Docker image to be used by setting `containerInfo.DockerInfo.image`. We can include the URI pointing to a `.dockercfg` file containing the login information for a private Docker repository. The `.dockercfg` file will be pulled into the sandbox and since the Docker Containerizer sets the HOME environment variable pointing to the sandbox, Docker CLI will automatically pick up the provided configuration file. `TaskInfo` and `ExecutorInfo` include the `ContainerInfo` field that can be used to configure Docker.

Note

Till Version 0.20, `TaskInfo` only contained `CommandInfo` (to launch a task as a `bash` command) or `ExecutorInfo` (to launch a custom executor which will run the task).

An image is essentially a snapshot of a container stored as an immutable file. The user can choose to launch a Docker image as a task or as an executor.

To run a Docker image as a task, we must set both the `command` and the `container` fields in `TaskInfo`. The Docker containerizer will use the command to launch the Docker image. `ContainerInfo` should be the type set as `docker` and should have a `DockerInfo` that has the Docker image we want to launch.

To run a Docker image as an executor, we must set `ExecutorInfo` that contains `ContainerInfo`. `ContainerInfo` should have the type set to docker and `CommandInfo` that will be used to launch the executor. Note that in this case, the Docker image is expected to launch as the Mesos executor that will register with the slave after the launch. Also, note that when launching the Docker image as an executor, Mesos does not launch the command executor but uses the PID of the Docker container executor.

A Docker image currently supports having an entrypoint and/or a default command. To run a Docker image with an entrypoint defined, `CommandInfo` must have the shell option

that must be set to `false`. If the shell option is set to true, `DockerContainerizer` will run the user command wrapped with `/bin/sh -c`, which will also become parameters to the image entrypoint.

Tip

To run a Docker image with the default Docker command (`docker run image`), the `CommandInfo` file must not be set. If set, it will override the default command.

`DockerContainerizer` launches all containers with "mesos-" prefixed to `SlaveId`, and assumes that all the containers with the "mesos-" prefix can be stopped or killed by it. The Docker containerizer can recover Docker containers on slave restart, with the `docker_mesos_image` flag enabled, the Docker containerizer assumes that the containerizer is running in a container itself and the recovery mechanism takes place accordingly. Currently, the default networking mode is host networking when running a Docker image. Starting from 0.20.1, Mesos supports the bridge networking mode.

External containerizer

In case of external containerizers, the isolation between all the executors running on the slave has to be managed by the external containerizer implementation (http://mesos.apache.org/documentation/latest/external-containerizer). External containerizer implementation has two parts to it. **External containerizer** (**EC**) provides an API for containerizing via an external plugin on the Mesos slave. **External Containerizer Program** (**ECP**) is an executable external plugin that implements the actual containerizing logic.

EC invokes ECP as a shell process with the command as a parameter to the ECP. Any additional data is passed via `stdin` and `stdout`. The input and output are protocol buffer messages. The `containerizer::XXX` messages are defined in `include/mesos/containerizer/containerizer.proto` and `mesos::XXX` are defined in `include/mesos/mesos.proto`.

Here are the commands that ECP has to implement via commands:

`launch < containerizer::Launch`

The `launch` command is used to start the executor. The `Launch` message contains all the information that is needed to launch a task via an executor. This call should not wait for the executor/command to return:

`update < containerizer::Update`

The `update` command is used to update the resource constraints for the container. It takes `ContainerID` and `Resources` via the `Update` message and does not produce any output:

`usage < containerizer::Usage > mesos::ResourceStatistics`

The `usage` command is used to collect the current resource usage of a container. It takes the `Usage` message that contains `ContainerID` and returns the `ResourceStatistics` protocol message. `ResourceStatistics` contains the timestamp, CPU, and memory resource usage:

`wait < containerizer::Wait > containerizer::Termination`

The `wait` command is blocked until the executor/command is terminated. It takes the `Wait` protocol buffer message and produces the `Termination` protocol buffer message.

The `Wait` message contains `ContainerID` and the `Termination` message contains the exit status, a Boolean flag `killed`, indicating whether the container was killed due to resource enforcement, and a string `message` field containing a detailed termination message:

`destroy < containerizer::Destroy`

The `destroy` command destroys terminates the executor whose `ContainerID` is contained in the input message. It does not return any output:

`containers > containerizer::Containers`

The `conatainers` command does not expect any input and returns all containers currently running.

`recover`

The `recover` command allows the ECP to recover the state of the executor. ECP can do checkpointing and can use this call to use the checkpointed state in recovery.

When launching a task, the EC will make sure that ECP first receives a launch command before any other commands for that container. All the commands are queued until launch returns from the ECP. Other than this, commands can be executed in any order. Logs from the ECP are redirected to sandbox logger. The return status of the ECP indicates whether it was successful. A nonzero status code signals an error. Mesos containerizer documentation has detailed sequence diagrams, explaining the container life cycle in context of EC and ECP.

All the containerizers can receive extra information from frameworks with `TaskInfo` to configure the containers that will be launched. For example, we can specify an image to be used with external containerizers by `containerInfo.image`. ECP can also use the following environment variables:

- `MESOS_LIBEXEC_DIRECTORY`: This is the location of the mesos-executor, mesos-usage, and so on.
- `MESOS_WORK_DIRECTORY`: This specifies the working directory of the slave and is used for distinguishing the slave instances.
- `MESOS_DEFAULT_CONTAINER_IMAGE`: This is only provided to the launch command. This specifies the default image to be used.

To use external containerizers with Mesos, the slaves need to be configured with the `--`

`isolation=external` and `--containerizer_path=/path/to/external/containterizer` options.

Fault tolerance

Fault tolerance is an important requirement for a data center OS. The ability to keep functioning in the event of a failure becomes indispensable, when working with large-scale systems. Mesos has no single point of failure in the architecture and can continue to operate in case of faults of various entities. There are three modes of fault tolerance that we have to deal with: machine failures, bugs in Mesos processes, and upgrades. Note that all of the points mentioned earlier can happen with any entity in Mesos. There are mainly three components of Mesos that need to be resilient to these faults: master, slave, and framework.

In case of machine running the Mesos slave fails, the master will notice, and inform the frameworks about the slave failure event. The framework can choose to reschedule the tasks running on that slave to other healthy slaves. Once the machine is fixed and the slave process is restarted in a healthy mode, it will reregister with the master and will, again, be part of the Mesos cluster. In case of a failure of the slave process or during slave upgrade, the slave process might be unavailable, but executors running on the slave will be unaffected. The slave process on restarting will *recover* these tasks, which is called slave recovery.

When a framework scheduler fails due to a machine failure, process failure, or during upgrade, the tasks from the framework will continue to being executed. If the framework implements the scheduler failover, it will reregister with the master and get information about the status of all its tasks.

Mesos uses leader election for master high availability. This means that in case of a master failure due to a machine failure, fault in the master process or while upgrading, one of the standby masters will be elected as a leading master and all the slaves and frameworks, including their executors and tasks will keep running. Slaves and framework schedulers will reregister with the new leading master and thus fault in the Mesos master will not affect the Mesos cluster. Mesos currently supports ZooKeeper as the only leader election service. The Mesos community is considering the addition of other services, such as etcd, for leader election. We will see how to use ZooKeeper for master high availability in the next section.

ZooKeeper

Apache ZooKeeper (http://zookeeper.apache.org) is an implementation distributed coordination service, which is required to build distributed systems. It includes a consensus algorithm called **ZooKeeper Atomic Broadcast** (**ZAB**). ZAB is a high-performant broadcast algorithm used for primary-backup systems that provide strong guarantees (http://dl.acm.org/citation.cfm?id=2056409). ZooKeeper is battletested and a number of projects rely on ZooKepeer for high availability.

To start Mesos in high-availability mode, we need to install and run a ZooKeeper cluster. An `N` node ZooKeeper cluster will be able to survive `ceil(N/2)` ZooKeeper nodes. We will install ZooKeeper on three nodes at `zoo1`, `zoo2`, and `zoo3`. The following are the steps to install ZooKeeper on each node:

1. Mesos includes the ZooKeeper distribution in the `3rdparty` folder. We can also use upstream distribution from http://zookeeper.apache.org:

   ```
   ubuntu@master:~ $ cd 3rdparty
   ```

2. Unpack and `cd` to it:

   ```
   ubuntu@master:~ $ tar -xzf zookeeper-*.tar.gz
   ubuntu@master:~ $ cd zookeeper-*
   ```

3. Create the ZooKeeper configuration file `zoo.cfg` in the `conf` directory. We can use the sample configuration provided with the distribution, which will set `tickTime=2000`, `dataDir=/tmp/zookeeper`, `clientPort=2181`, `syncLimit=5`, and `initLimit=10`:

   ```
   ubuntu@master:~ $ cp conf/zoo_sample.cfg conf/zoo.cfg
   ```

   Now, we need to put all the nodes that are part of the ZooKeeper cluster, which, in our case, will be as follows:

   server.1=zoo1:2888:3888

   server.2=zoo2:2888:3888

   server.3=zoo3:2888:3888

4. On startup, each ZooKeeper node determines its identity from the `myid` file in the `conf` directory. Each ZooKeeper node should have a unique number between 1 and 255 in the file.

5. Now, we can start the ZooKeeper:

```
ubuntu@master:~ $ bin/zkServer.sh start
```

Now, we can instruct the Mesos installation to use ZooKeeper to ensure high availability:

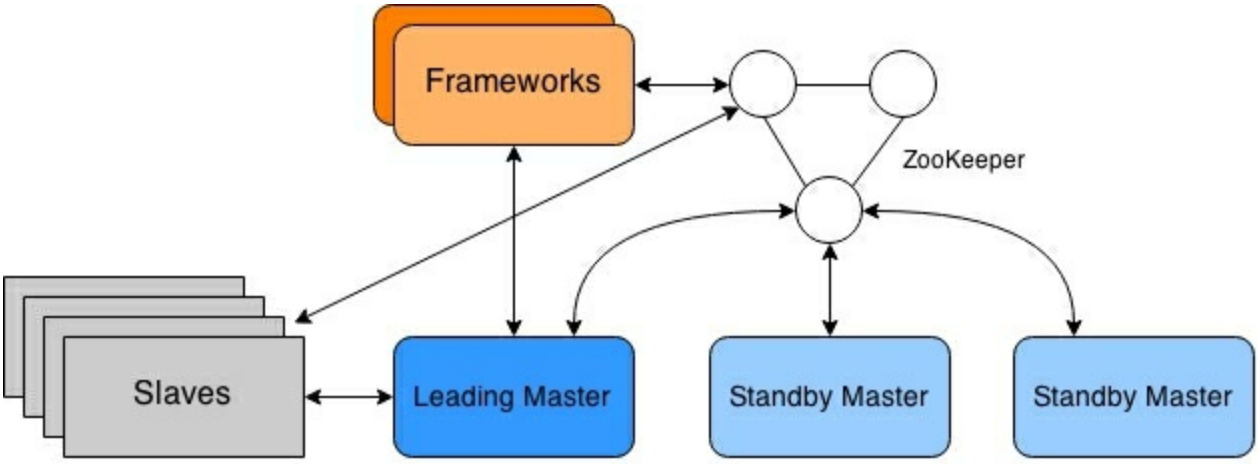
1. Start the Mesos-master with the `--zk` flag, providing the address of ZooKeeper followed by the ZooKeeper namespace to be used for Mesos (here, we will use `/mesos` as the path):

```
ubuntu@master:~ $ mesos-master --zk=zk://zoo1:2181,zoo2:2181,
zoo3:2181/mesos
```

2. Start the Mesos slaves by providing ZooKeeper URIs to the `--master` flag:

```
ubuntu@master:~ $ mesos-slave.sh --master=zk://zoo1:2181,
zoo2:2181,zoo3:2181/mesos
```

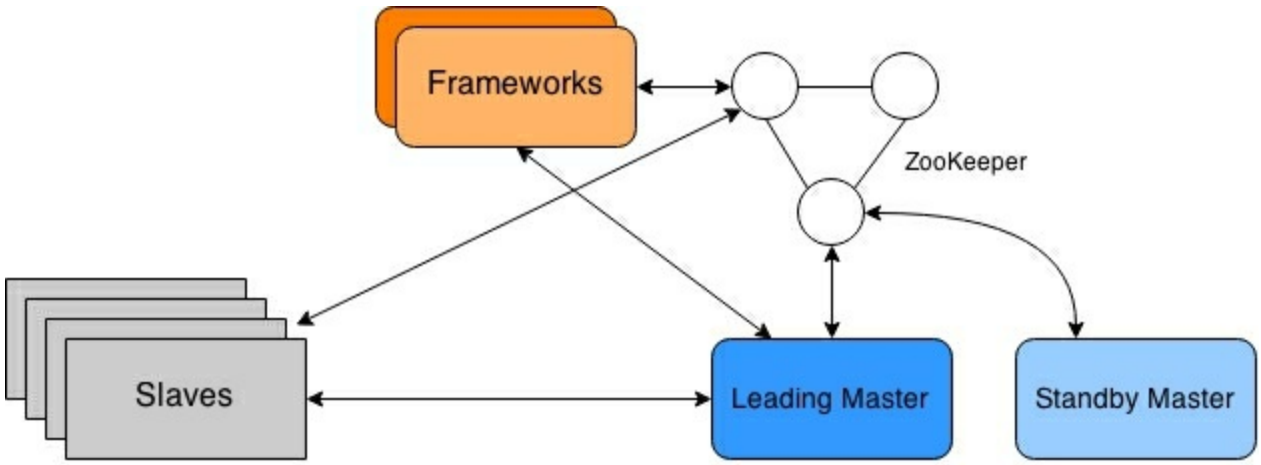
Now the Mesos slaves will use ZooKeeper to figure out which is the current Mesos master. In case of a failure, ZooKeeper will elect a new master and will start referring the new master to the slaves. The following image describes a deployment with two standby Mesos masters. When running in high-availability mode, all the slaves and frameworks schedulers contact the current Mesos master via ZooKeeper.



Mesos running in high availability

In the case of failure of the current Mesos master, ZooKeeper elects one of the standby

masters as the new master, and all the slaves and frameworks start contacting the new master, as shown in the following image:



Mesos master failure recovery using standby master

Failure detection and handling

Mesos includes a two leader election abstraction of ZooKeeper contenders and detectors. Contenders represent everyone who is trying to be the master, while detectors are only interested in knowing who the current master is. Each Mesos master uses both detector and contender to know the current master and elect itself as a master. The Mesos slaves and framework scheduler drivers use a detector to find out the current master to communicate. ZooKeeper group leader candidates are responsible for handling the membership registration, cancellation, and monitoring by watching the following the ZooKeeper session events.

- Connection
- Reconnection
- Session expiration
- ZNode creation, deletion, and updation

Mesos uses timeout to detect component unavailability. Once disconnected from ZooKeeper, after a configured time, each Mesos component will timeout. Contender and detector sessions will timeout after `MASTER_CONTENDER_ZK_SESSION_TIMEOUT` and `MASTER_DETECTOR_ZK_SESSION_TIMEOUT` amount of time, respectively. Both these constants default to 10 seconds. Different Mesos components will take different actions while disconnecting from ZooKeeper:

- If a slave disconnects from ZooKeeper, it simply stops acting on any message that it receives to ensure that it does not serve the old master. Once the slave reconnects to ZooKeeper, the slave acts on the master's messages (newly-elected or an old one).
- If a scheduler driver disconnects from ZooKeeper, it will notify the scheduler about it and it is the scheduler's responsibility to act on the situation.
- If the Mesos master disconnects from ZooKeeper, it will enter into a leaderless state.
    - If the Mesos master was a leader, it aborts itself. It is the responsibility of an administrator to start it again to act as a standby master.
    - If the Mesos master was a standby, it simply waits for a reconnection to ZooKeeper.
- If the network partition separates a slave from the current leader, the slave fails to provide health checks and the leader will mark the slave as deactivated. Deactivated slaves may not reregister with the leader and will be requested to shut down on any post-deactivation communication. All the tasks on the slave are

marked as LOST state and frameworks can act upon these tasks (like rerunning them).

For more details on Mesos and ZooKeeper usage, check out the source code in the `LeaderContender` and `LeaderDetector` classes from `src/zookeeper/`, which implements a generic ZooKeeper leader election algorithm. Group in `src/zookeeper/` implements ZooKeeper group leadership. We will also see some operational aspects of ZooKeeper in Chapter 8, *Administering Mesos*.

Registry

The original Mesos design was to keep the master stateless for simplicity and scalability. The registry adds a minimal amount of persistent state to the Mesos master. Registry was added to address the following cases:

- If a slave fails when the master is down, the task lost notification is sent to the framework. This leads to an inconsistent state, where the framework thinks that the task ran while Mesos does not know about it.
- When a new master is elected, a rough slave can reregister, resulting in inconsistency where the slave is running some task and the framework is not aware of it.

Currently, the registry stores the list of registered slaves. This allows the newly-elected master to detect the slaves that do not reregister and notifies frameworks about them and prevents the rouge slaves from reregistering. The registrar specifies the abstraction for operations that can be applied on the registry. The registrar hides all the state management for different implementations of the state storage and does some clever optimization, such as batching multiple writes, so as to avoid performance bottleneck. The Mesos master consults the registrar during registration, reregistration, and slave removal. Only the leading master can write to the registry. Mesos currently has the following backend implementations of the registrar:

- In memory: In memory implementation of storage is mainly useful for testing Mesos. It is recommended that you do not use it for any serious usage.
- LevelDB: This is a lightweight persistent library (http://leveldb.org). Users can use LevelDB storage backend when they are running Mesos in non high-availability mode, and thus, don't have access to the ZooKeeper cluster. Note that this setup will require moving LevelDB state storage (stored in master's work directory) to a new machine, when they move the Mesos master from one machine to another.
- Replicated log: Mesos uses replicated log abstraction that is implemented using the MultiPaxos algorithm. The replicated log is a distributed append-only data structure. It can store arbitrary data. The replicated log is the recommended backend storage and is the default registry.

Extending Mesos

Extensibility is an important feature for any software, as it allows it to adapt to specific requirements. Mesos tries to keep its core to the minimum and pushes all the optional features out of core Mesos, that may not be required by all Mesos users. Mesos provides integration points, where various external implementation or systems can integrate with Mesos. Almost all the components in Mesos are swappable with different implementations for specific functionalities. We have already seen how we can plug our own isolators, containerizers and registrar implementation. In this section, we will see some more features that allow Mesos to be customized.

Mesos modules

Mesos provides modular architecture, but there will always be points where Mesos needs to adapt to a particular organization needs. Mesos modules provide ways to extend the inner working of Mesos to cater to such requirements, without the need of rewriting, recompiling, or relinking Mesos (http://mesos.apache.org/documentation/latest/modules). For example, we want to try a new allocation algorithm written in Python for Mesos. With the allocation module, we can write the module and load it with the master, without imposing new dependencies to Mesos. In this way, Mesos modules are extension and integration points for third-party software, and also provide an easy way to add experimental new features.

Mesos currently supports various kinds of modules. A kind of a module decides what callbacks the given module will receive. `src/examples` includes examples for each kind of modules:

- Authentication modules: These modules in Mesos allow third alternative authentication mechanism. Authenticatee and Authenticator modules in Mesos are examples of such modules.
- Isolator modules: These modules can provide a specialized isolation and monitoring mechanism. A module providing isolation for GPU hardware would be an example of such a module.
- Anonymous modules: These modules do not receive any callbacks and just exist with their parent process. Thus, anonymous modules do not extend or replace any of the functionality, which is already provided by Mesos. They do not require any specific selector flags and will be immediately instantiated when loaded by the master or slave.
- Allocator module: This module in the Mesos master is invoked periodically to determine which framework should get the resource offers next. By implementing an allocator module, module developers can provide new mechanisms for resource allocation, such as resource oversubscription and preemption.
- Hook: Not all Mesos internal components are suitable for completely new kinds of module. Hooks allow module writers to extend the functionality of such components.

Modules are shared libraries that can be loaded using the `--modules` flag with the module name, while starting the Mesos master and slave process. `--modules` takes the inline JSON string or filepath that contains the JSON string, which specify what modules to load and other configurations. Here is the JSON schema for modules:

```
{
    "type":"object",
    "required":false,
    "properties":{
      "libraries": {
        "type":"array",
        "required":false,
        "items":
        {
          "type":"object",
          "required":false,
          "properties":{
            "file": {
              "type":"string",
              "required":false
            },
            "name": {
              "type":"string",
              "required":false
            },
            "modules": {
              "type":"array",
              "required":false,
              "items":
              {
                "type":"object",
                "required":false,
                "properties":{
                  "name": {
                    "type":"string",
                    "required":true
                  },
                  "parameters": {
                    "type":"array",
                    "required":false,
                    "items":
                    {
                      "type":"object",
                      "required":false,
                      "properties":{
                        "key": {
                          "type":"string",
                          "required":true
                        },
                        "value": {
                          "type":"string",
                          "required":true
                        }
                      }
```

```
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
```

A module library can hold many modules. Module specification mandates at least one `file` or `path` entry for each library entry. The `file` refers to a location on the machine via a relative or absolute path. The `name` specifies the module name and gets expanded into platform-specific extension (`.so` for Linux and `.dylib` on OS X). If both `file` and `name` are specified, the `file` gets the preference and the `name` is ignored. When the path is not specified, the library is searched for in the platform default directories (`LD_LIBRARY_PATH` on Linux and `DYLD_LIBRARY_PATH` on OS X). Here is an example of the module configuration:

```
{
  "libraries": [
       {
         "file": "/path/to/libab.so",
         "modules": [
           {
             "name": "org_apache_mesos_a",
             "parameters": [
               {
                 "key": "X",
                 "value": "Y",
               }
             ]
           },
           {
             "name": "org_apache_mesos_b"
           }
         ]
       }
     ]
   }
```

The preceding example configuration specifies the loading of two modules: `org_apapche_mesos_a` and `org_apapche_mesos_b` from the libraries file located at `/path/to/libab.so`. Here, the `org_apache_mesos_a` module is loaded providing

command line arguments X=Y, while the `org_apache_mesos_b` module is loaded without any arguments.

The following steps are taken while loading a Mesos module:

1. Load dynamic libraries that contain modules from a modules instance, which may have come from a command-line flag.
2. Verify version compatibility.
3. Instantiate singleton per module.
4. Bind the reference to the use case.

Tip

Note that at the time of writing, Mesos modules are very recent addition to Mesos (included in version starting from 0.21) and is still an experimental feature. Also, note that the only core Mesos modules (authentication isolation and so on.) will be maintained by the Mesos project. A separate mailing list (<modules@mesos.apache.org)> has been created for discussion regarding the Mesos modules.

Module naming

Mesos module names should have unique names. If modules JSON configuration has duplicate names, the Mesos process will abort. Mesos thus encourages module writers to use Java package naming conventions (http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html) for naming modules. In general, we should follow the following conventions:

- Keep the casing in the module name
- Reverse the domain name and keep it lowercase
- Use underscore as a separator
- Different modules from the same organization need to have different names
- Do not use kind names as module names

For example, if a module name is `fooBar` and domain name is `mesos.apache.org`, the module symbol name could be `org_apache_mesos_fooBar`.

Module compatibility

As noted earlier, before loading the module, the dynamic library containing the module has to be loaded. There are two checks that are performed to ensure compatibility:

- Library compatibility check that verifies module API version check
- Module compatibility check that verifies module kind version check

The developer of the module is responsible for conveying what versions their module is compatible with. This compatibility information is maintained in a `src/manager/manager.cpp` table. The table contains all the possible kinds of modules that Mesos recognizes with Mesos release version number that it is compatible with, which we will call kind version. So, a successful loading of the module requires the following relationship to hold : kind version <= Library version <= Mesos version.

Allocation module

The allocation module (http://mesos.apache.org/documentation/latest/allocation-module) is responsible for determining which frameworks get the resource offers. A framework can refuse some resources in offers and run tasks in others. Optionally, allocated resources can have offer operations applied to them for frameworks to alter the resource metadata. Resources will be recovered from a framework when a tasks finishes, fails, or is lost due to a slave failure, or when an offer is rescinded. The allocation module is pluggable with the API defined in `include/mesos/master/allocator.hpp`.

Method	Description
`initialize(flags, offerCallback, roles)`	This initializes the allocator.

`addFramework(frameworkId, frameworkInfo, usedResources)`
 	
`removeFramework(frameworkId, frameworkInfo, usedResources)`
	This adds/removes the framework from the allocation consideration.

`activateFramework(frameworkId)`
 	
`deactivateFramework(frameworkId)`
	This activates/deactivates the given framework. Note that offers are only sent to the active frameworks.

`addSlave(slaveId, slaveInfo, totalResources, usedResources)`
 	
`removeSlave(slaveId, slaveInfo, totalResources, usedResources)`
	This adds/removes the given slave from the allocation process.

`activateSlave(slaveId)`
 	
`deactivateSlave(slaveId)`
	This activates/deactivates the given slave. Note that only offers from active slaves are considered.

`requestResources(frameworkId, requests)`	This is the callback when a resource request is received from the framework.
`updateAllocation(frameworkId, slaveId, operations)`	This updates the allocation of resources for the given framework on the given slave.
`recoverResources(frameworkId, slaveId, resources, filters)`	This is the callback to recover resources that are considered used by the framework.

`reviveOffers(frameworkId)`	This is the callback used when the framework wants to revive the offers that were filtered previously.
`updateWhitelist(whitelist)`	This updates the whitelist of slaves.

Allocator modules are implemented in C++ and must implement an allocator interface defined in `mesos/master/allocator.hpp`. If we want to write a custom allocator in a different language, we have to implement a thin proxy allocator in C++, which, in turn, delegates calls to another language implementation of the allocator. A custom allocator can be used by loading the module on the Mesos master by including it in the `--modules` flag and specifying the allocator to be used via the `--allocator` flag.

The default allocator is `HierarchicalDRFAllocatorProcess` defined in `src/master/allocator/mesos/hierarchical.hpp`. It's an actor-based implementation making it nonblocking. Method calls will return immediately after placing corresponding actions into the actor's queue. We can reuse this actor design by extending the `MesosAllocatorProcess` class defined in `src/master/allocator/mesos/allocator.hpp` and wrap our actor-based allocator in `MesosAllocator`.

Another way to modify the behavior of the resource allocation is to implement a new sorter algorithm for the in-built hierarchical allocator. A sorter is used by the allocation module to determine the order in which clients (users roles and frameworks) are offered resources. Sorters can be implemented in C++ by following the Sorter API defined in `src/master/allocator/sorter/sorter.hpp`.

Method	**Description**
`void add(client, weight=1)` `void remove(client)`	This adds/removes the client from the allocation consideration.
`void deactivate(client)` `void activate(client)`	This removes/readds the client from the sort.

`void add(slaveId, resources)`
`void remove(slaveId, resources)`
`void update(slaveId, resources)`
	This adds/removes/updates the resources in a total pool to be allocated.

`List<string> sort()`	This returns the list of clients ordered according to the Sorter's policy, the order in which they should be allocated resources.

`void allocated(client, slaveId, resources)`
 	
`void update(client, slaveId, oldResources, newResources)`
 	
`void unallocated(client, slaveId, resources)`
	This specifies that the resources have been allocated/updated/unallocated to the given client.

`Map<SlaveId, Resources> allocation(client)`	This returns the resource allocated to the given client.
`bool contains(client)`	This returns `true` if the sorter contains the given client.
`int count()`	This returns the number of clients the sorter has.

The default sorter used by hierarchical allocator is `DRFSorter`, which is defined in `src/master/allocator/sorter/drf/sorter.hpp`. `It` implements fair sharing of resources and supports priorities through weights.

Mesos hooks and decorators

Hooks provide another way of extending the internal working of Mesos, but unlike other modules, hooks do not modify request processing. Hooks is a function that is called by Mesos during the lifetime of the Mesos components. They provide an interface to inject functionality into the Mesos request life cycle. This provides a lot of opportunity of integrating with external tools and systems. We can think of hook modules as more like event callbacks across various entities in Mesos. At the time of writing, Mesos Version 0.23 includes support for hooks in task launch actions. Mesos hooks API is defined in `mesos/hooks.hpp`, and each hook defines the insertion point of the hook along with the context. For example, `masterLaunchTaskHook` will have `TaskInfo` as part of context. A decorator is a special kind of hook that has a return type. Decorators have the ability to modify or decorate the object as it passes through various phases in life cycle. For example, a decorator can add task labels to `TaskInfo` as it passes through the Mesos master. Hooks are specified as a comma-separated list of hook names via the `--hooks` flag.

Task labels

Labeling has become a very useful feature that helps in integrating a host of tolling around service discovery, external security systems, resource monitoring, accounting, and so on. Without support for labels, prior to Version 0.22, the common pattern for conveying such information was through task names. Task labels provide a way to attach globally visible metadata to tasks. Labels are not processed by Mesos itself and its interpretation is left to external tools. Labels are exposed via master and slave endpoints, which can be queried by external tools.

Labels are arbitrary key-value pairs that can be attached to Mesos tasks as part of `TaskInfo`. Note that `TaskInfo` also has a `data` field, which can hold arbitrary data for frameworks, but that is typically used to hold executor-related information. Also, unlike data, labels are kept in memory by master and slave processes. Thus, labels should be only be used to tag lightweight metadata about the tasks.

Summary

In this chapter, we looked at the Mesos architecture in detail and learned how Mesos deals with resource allocation, resource isolation, and fault tolerance. We also saw the various ways in which we can extend Mesos.

In the next chapter, we will develop a Mesos framework.

Chapter 7. Developing Frameworks on Mesos

Mesos is a common substrate for cluster computing. This means that Mesos is able to support a wide range of frameworks running on common resources. We saw how to run various frameworks on Mesos in the preceding chapters. There are many more frameworks developed by the Mesos community that validate the generality of the Mesos API and serve the most common use cases and a wide range of workloads. We will always have some use cases that are not served by any existing frameworks. This is a good time to roll out our own framework and realize the power of Mesos. This chapter will discuss the Mesos API in detail, and we will go through the steps involved in implementing a Mesos framework in Java. We will cover the following topics:

- The Mesos API
- Developing a Mesos framework
- Building our framework
- Advanced topics
- Developer resources

The Mesos API

Mesos has a simple and intuitive API to develop applications on top of distributed infrastructure. Mesos hides most of the details on dealing with distributed infrastructure. The Mesos API is discussed in detail at http://mesos.apache.org/documentation/latest/app-framework-development-guide/. Before we dive deeper into the Mesos API, let's first take a look at the various messages used by Mesos.

Mesos messages

Mesos uses protocol buffers to define messages that are sent to different components of Mesos. Protocol buffer defines an extensible messaging format that can be interchanged across languages and platforms. The following are some of the messages exchanged between various Mesos entities, defined in `include/mesos/mesos.proto`.

- `MasterInfo`: This describes the master and contains an ID, hostname, IP, port, and master PID.
- `SlaveInfo`: This describes a slave and contains an ID, hostname, and port. It also has resource and attribute information.
- `Resource`: This describes a resource on a machine.
- `Attribute`: This describes a machine attribute.
- `Request`: This describes a resource request. It has the resource field, describing the resource on a machine, and the `slaveId` field can be used to restrict the resource request to a particular slave.
- `FrameworkInfo`: This describes a framework. It has a hostname, user name, as well as authentication information, such as role and principal. It also has other access to various framework configuration parameters, such as `failover_timeout`, checkpoint, and the `webui_url` framework. If the `frameworkID` field is set in `FrameworkInfo`, then the scheduler performs a failover.
- `CommandInfo`: This describes a command that is executed from the shell. It contains URIs that specify the location of downloadable files as URI. The `extract` field specifies whether to extract the file after downloading it. If set, the file will be extracted to the working directory of the executor. Note that currently the `tar.gz` and `zip` formats are supported in `CommandInfo` and by default, the value is set to true. If an executable field is set, then the executor permission is set on the downloaded file. It also contains `ContainerInfo` and `Environment` that contain information on the container and environment variables that will be used while running. `ContainerInfo` has an image URI and options are passed to the containerizer. `user` specifies the user used to run the command, and the `arguments` field holds the arguments. For the `shell` field, if set to true, the command value is executed via `/bin/sh -c` and arguments are ignored, and if set to false, the arguments are passed to the command, treating it as an executable.
- `ContainerInfo`: This describes an extensible container configuration. It has hostnames, volumes, and types. `Volume` describes a volume mapping from the host of a container or from a container to the host. `Volume` has an absolute directory

path of a container, the path of a host (absolute or relative to container working directory), and the mode for read-write or read-only mounting. The `Type` field describes the type of the container. The current options for `Type` are `Mesos` and `Docker`. The optional `DockerInfo` field describes the Docker configuration in case of docker containers. It contains the image and network configuration (host, bridge, or none), port mapping between the host and container, and other parameters to be passed to the docker CLI. It also has a flag to control whether or not to run Docker in the privilege mode and to force the download of a Docker image from the registry.

- `DiscoveryInfo`: This contains the information used for service discovery. The visibility field can be used to restrict the scope of service discovery for the executor. The possible values are `FRAMEWORK`, `CLUSTER`, and `EXTERNAL`. The tasks with the `FRAMEWORK` visibility can only be discovered by other tasks within the same framework. Tasks with the `CLUSTER` visibility can be discovered by any task running on Mesos, including tasks from other frameworks. `EXTERNAL` provides unrestricted discoverability for tasks and can be discovered by external services and clients outside Mesos.
- `ExecutorInfo`: This describes the information about an executor. It has the `executorID`, `frameworkID`, `CommandInfo`, `containerInfo`, `discoveryInfo`, `resources`, and configuration information, such as `name` and `source`. The data field can be used to pass on arbitrary data.
- `HealthCheck`: This describes the health-check information for an arbitrary command, process, task, or executor. It has `CommandInfo` that is used for health checking and other configuration parameters, such as timeout, interval, grace periods, and so on, as fields.
- `ResourceUsage`: This describes a snapshot of usage of resources by an executor. It has `slaveId`, `frameworkId`, `executorId`, `executorName`, `taskId`, and `ResourceStatstics`, which has detailed resource usage information.
- `Offer`: This describes the resources available on a slave using the `offerId`, `hostname`, `slaveId`, `frameworkId`, `executorId`, `attribute`, and `resource` fields.
- `TaskInfo`: This describes a task and has following fields: `name`, `taskId`, `SlaveId`, `resource`, `executorInfo`, `CommandInfo`, `containerInfo`, `discoveryInfo`, `HealthCheck`, `data`, and `Labels`. `Labels` are arbitrary key-value pairs that can have any framework-specific interpretation.
- `TaskStatus`: This describes the current status of a task. It has `taskId`, `taskState`, `slaveId`, `executorId`, optional message, and data field. `TaskState` is an enum that describes the possible states of task, such as staging, starting,

running, finished, failed, killed, and lost.

- `Filters`: This describes different filters that can be applied by the framework scheduler to indicate the unused resources. It currently has `refuse_seconds` that specifies the time for which the unused resources will be considered refused. The default value is 5 seconds.

The scheduler API

The framework scheduler is responsible for managing resources for the framework. The framework gets resource offers from the Mesos master and is responsible for responding to them as well as managing executors. The `org.apache.mesos.Scheduler` interface defines the interface that the framework must implement to receive callbacks from the Mesos master. As all the methods provide access to a driver instance, the scheduler should not store it. The following are the methods defined by the Scheduler interface, declared in `MESOS_HOME/include/mesos/scheduler.hpp`:

- `registered(SchedulerDriver, FrameworkID, MasterInfo)`: This method is invoked when the scheduler registers with the Mesos master. `FrameworkId` is a `uniqueId` assigned by Mesos. `MasterInfo` provides information on the current master. This method can be used to initialize any scheduler data structures.
- `reregistered(SchedulerDriver, MasterInfo)`: This method is invoked after the framework scheduler reregisters with a newly-elected master and `masterInfo` provides information on the new master.
- `disconnected(SchedulerDriver)`: This is invoked when the scheduler disconnects from the master.
- `resourceOffers(SchedulerDriver, List<Offer>)`: This is the meat of most framework schedulers. `resourceOffers()` is invoked when the master offers resources to the framework. Each offer contains resources from a single slave. The scheduler should either accept the offer and launch tasks using `offerIds` or reject the offers. Depending on the allocator, a resource can be offered to more than one framework. In this scenario, the first one to use the offer to launch the tasks will succeed, while the other framework will receive `offerRescinded()`.
- `offerRescinded(SchedulerDriver, OfferID)`: This method is invoked when the master invalidates offers given to the framework. This can happen because of the disappearance of the slave or another framework using the offered resources. If the scheduler fails to receive the callback and tries to launch the task using the expired `offerId`, it will receive the `TASK_LOST` status for those tasks.
- `statusUpdate(SchedulerDriver, TaskStatus)`: This method is called whenever Mesos has a message for the framework. It is called when the status of the task has changed, such as when the task finishes or the task is lost because the slave it was running on is lost, and so on. We can use this method to keep track of the progress of work by the framework, such as marking tasks as finished, marking them to be *relaunched* the event failure, or *ignored* depending on the framework.

- `frameworkMessage(SchedulerDriver, ExecutorID, SlaveID, byte[] data)`: This is invoked to deliver the executor message to the scheduler. The scheduler has access to the executor and slave IDs along with the data sent by the executor. This is based on the best effort message delivery service and the messages are not retransmitted in the event of a failure.
- `slaveLost(SchedulerDriver, SlaveID)`: This is invoked to notify the scheduler that Mesos is not able to reach the slave with the given `SlaveID`. A typical response to this method is to reschedule the tasks that had been running on the slave to a new slave.
- `executorLost(SchedulerDriver, ExecutorID, SlaveID, int status)`: This is invoked to notify the scheduler that the particular executor has terminated with the given status, indicating the exit status of the executor process. For most frameworks, the scheduler does not need to perform any specific actions, as the `TASK_LOST` status update will be generated for the tasks running in the executor.
- `error(SchedulerDriver, String message)`: This is invoked when an unrecoverable error has occurred and the scheduler or driver has terminated. This method should be used to clean up any resources, such as locks, and so on.

The SchedulerDriver API

SchedulerDriver interface defines scheduler life cycle methods and methods to interact with Mesos. It is responsible for invoking framework scheduler callbacks.

All methods return the `Status` of the driver after the call. The following are the different scheduler life cycle methods:

- `start()`: The `start()` method starts the scheduler and is required to be executed before any other methods can be executed.
- `stop(boolean failover)`: This is invoked to stop the driver. The failover flag indicates whether the framework is set for a failover or not. If the flag is set to false, then the framework will not connect to Mesos, and Mesos will unregister the framework, killing any running executors or tasks from the framework. If the flag is set to true, Mesos continues to run the executors and tasks and allows you to reconnect to them. The new driver in this case could be running on the same process or a different process on a different machine. There is a convenient no-argument `stop()` method equivalent to `stop(false)`.
- `abort()`: The `abort()` method will abort the driver and no more callbacks can then be made to the scheduler. The driver can start another instance if required (for a failover).
- `join()`: The `join()` method waits for the driver to exit (`stop()` or `abort()`) and may block indefinitely. Here, the return status is useful to know whether the driver exited normally or aborted.
- `run()`: This is a convenient method to start and join the driver.
- `requestResource(Collection<Request>)`: This requests resources from Mesos, which will be offered to the framework scheduler.
- `launchTasks(Collection<OfferID> offerIds, Collection<TaskInfo> tasks)` and

  `launchTasks(Collection<OfferID> offerIds, Collection<TaskInfo> tasks, Filters filters)`:

  This launches a set of tasks on a set of offers. Any remaining resources not used by executors/tasks are considered rejected. If present, the filter will be applied on the remaining resources. If no tasks are specified, all the resources are considered rejected.
- `declineOffer(OfferID offerId, Filters filters)` and

`declineOffer(OfferID offerId)`:

This is a convenient method used to decline resources from an offer. `Filters` will be applied, if present.

- `reviveOffers()`: This removes any previously set filters, enabling the framework to receive offers from previously filtered slaves.
- `sendFrameworkMessage(ExecutorID executorId, SlaveID slaveId, byte[] data)`: This is used to send messages from the framework to one of its executors. This is the best effort delivery and failed messages are not retried.
- `reconcileTasks(Collection<TaskStatus> statuses)`: This will make the master send back the latest status for each task in the argument. If there are no tasks specified, the master sends all nonterminal tasks currently known about. This method can be used by frameworks to query the status of tasks. Tasks that are no longer found will have the `TASK_LOST` update generated.

`MesosDriverScheduler` is an implementation of the `DriverSchdeuler` interface that uses native Mesos implementation through Java Native Interface. We will use it to interact with Mesos. All the methods in `MesosDriverScheduler` are thread-safe and blocking on it does not affect scheduler callbacks, as they run on a separate thread.

The executor API

A framework executor is responsible for launching tasks that do the work assigned by the scheduler. `org.apache.mesos.Executor` defines the interface that must be implemented by the framework executor. The following are the methods defined by the executor interface, as stated in `MESOS_HOME/include/mesos/executor.hpp`. Any executor should not block, as only one callback is invoked at a time. The executor has access to the `ExecutorDriver` instance that ran the executor. Executors can use the `MESOS_HOME` variable to find the location of Mesos:

- `registered(ExecutorDriver, ExecutorInfo, FrameworkInfo, SlaveInfo)`: This method is invoked once the executor driver successfully connects to Mesos. A scheduler can use the data field in `ExecutorInfo` to pass any data to executors. `FrameworkInfo` contains the framework-related information, and `SlaveInfo` describes the slave that will be running the executor.
- `reregistered(ExecutorDriver, SlaveInfo)`: This is invoked when the executor reregisters with a slave after a restart.
- `disconnected(ExecutorDriver)`: This is invoked when the executor is disconnected from the slave, for example, due to slave restart.
- `launchTask(ExecutorDriver, TaskInfo)`: This method is the meat of the executor implementation. `launchTask()` is invoked when a task is launched on this executor. Note that `launchTask` also blocks the call, like all other methods in the executor, and no other callbacks will be invoked on this executor until this callback returns. Thus, if we require any time-consuming computation, we should do it in a separate thread. The task can be realized with a thread or process.
- `killTask(ExecutorDriver, TaskID)`: This is invoked when a task running within this executor is killed. Note that there will be no status update sent on behalf of the executor. If the executor wants the status update, then it has to create and send the status update for the task with `TASK_KILLED`.
- `frameworkMessage(ExecutorDriver, byte[] data)`: This is invoked when a framework message arrives to the executor. This is a best effort delivery and no retransmission is attempted for the message in the event of a failure.
- `shutdown(ExecutorDriver)`: This is invoked to notify the executor that itshould terminate all its currently running tasks. If the executor fails to send the terminal update (`TASK_KILLED`, `TASK_FINISHED`, or `TASK_FAILED`) for any tasks, a `TASK_LOST` status update will be created.
- `error(ExecutorDriver driver, String message)`: This is invoked when a fatal error has occurred in the executor or the executor driver. Here, the `message`

parameter contains the error details. Note that the driver will be aborted before invoking this callback.

The ExecutorDriver API

ExecutorDriver is an interface that connects an executor to Mesos. This interface has executor life cycle methods as well as methods to send messages to Mesos. It has the following methods. Similar to `SchedulerDriver` methods, all the methods in `ExecutorDriver` methods return status of the driver.

- `start()`

  `stop()`

  The `start()` and `stop()` methods are used to initialize the driver and cleanup. `start()` has to be executed before any other method invocation.
- `abort()`: This is called when the driver is aborted and no more callbacks can be made to the executor. `abort()` is different from `stop()` so that after `abort()` another driver instance can be started from the same process
- `join()`: This is used to wait for the driver to stop or abort and return the status of the join, which indicates whether the driver called `stop()` or `abort()`.
- `run()`: This method starts the driver and blocks immediately by calling `join()`
- `sendStatusUpdate(TaskStatus status)`: This is used to send the status update message to the scheduler. This method will be retried until an acknowledgement is received. If the executor is terminated, a `TASK_LOST` status update is sent to the scheduler.
- `sendFrameworkMessage(byte[] data)`: This is used to send any message to the scheduler. This uses best effort delivery and is not retried.

`MesosExecutorDriver` provides a native implementation of the `ExecutorDriver` interface. We will use it to invoke executor callbacks for communicating with the Mesos slave process. As noted, `MesosExecutorDriver` is thread-safe and blocking on it doesn't affect the callbacks.

Note

Note that Mesos Scheduler API and Executor API are part of the Mesos core, while SchedulerDriver API and ExecutorDriver API are provided by the language bindings. Most other language bindings use the same names for the drivers.

Developing a Mesos framework

A Mesos framework consists of a scheduler, executor, and launcher. The executors are optional and can be omitted if not required. We will develop our framework using the following steps:

1. Setting up the development environment.
2. Adding the framework scheduler.
3. Adding the framework launcher.
4. Writing executors.
5. Compiling and packaging.
6. Installing the framework on the cluster.

Setting up the development environment

The development environment for a Mesos framework does not require any special setup. We can use any IDE or a simple editor and terminal. All we need is to access language bindings to interact with Mesos. Mesos has language bindings for most of the languages. The Mesos distribution currently includes Java and Python drivers, while bindings for Go, Erlang, Haskell, and Clojure have been developed by the community. A complete list of the language bindings and other resources related to Mesos can be found at https://github.com/dharmeshkakadia/awesome-mesos/. There are two kinds of bindings for many languages: pure and non-pure. Pure bindings use an HTTP wire protocol to communicate with Mesos. Non-pure bindings require `libmesos` to be installed on the machines to communicate with Mesos.

We will use Mesos Java API to write our framework. Writing a framework using other language bindings is very similar to using Java client bindings. If we are using nonpure client bindings, then we need to have Mesos installed on our development machine, otherwise we can work with a remote Mesos cluster. As discussed, we will need client libraries to be available while compiling and running. We can use dependency management tools to grab this for us. We will use Maven for building and packaging purposes. The complete code along with the getting started template is available at https://github.com/dharmeshkakadia/MonteCarloArea/. We will create a framework to estimate the area under the given curve. We will call our framework `MonteCarloArea`; details of the framework are discussed in the next section. This is what the directory structure looks like:

MonteCarloArea

├──pom.xml

├──src

│ └──main

│ └──java

└──target

`src/main/java` contains our source code files and target will contain the compiled binaries. The `pom.xml` file in the root is used to describe Maven dependencies.

Now, let's add the minimum skeleton code for the Mesos framework. We will start by adding the Mesos client libraries to our project. Mesos Java bindings are available via the central Maven repository. Adding the following to the `pom.xml` file will make the latest version of the bindings available to our project:

```
<dependencies>
    <dependency>
        <groupId>org.apache.mesos</groupId>
        <artifactId>mesos</artifactId>
        <version>0.20.1</version>
    </dependency>
</dependencies>
```

Adding the framework scheduler

`org.packt.mesos.MonteCarloScheduler` is our scheduler class. We have to implement the scheduler interface. We will just print the messages without adding any logic to the callbacks for now. We will increasingly add logic to our scheduler and executor to perform the calculation:

```
public class MonteCarloScheduler implements Scheduler {

    public MonteCarloScheduler(){
    }

    @Override
    public void registered(SchedulerDriver schedulerDriver, Protos.
FrameworkID frameworkID, Protos.MasterInfo masterInfo) {
        System.out.println("Scheduler registered with id " +
frameworkID.getValue());
    }

    @Override
    public void reregistered(SchedulerDriver schedulerDriver,
Protos.MasterInfo masterInfo) {
        System.out.println("Scheduler re-registered");
    }

    @Override
    public void resourceOffers(SchedulerDriver schedulerDriver,
List<Protos.Offer> offers) {
        System.out.println("Scheduler received offers " +offers.
size());
    }

    @Override
    public void offerRescinded(SchedulerDriver schedulerDriver,
Protos.OfferID offerID) {

    }

    @Override
    public void statusUpdate(SchedulerDriver schedulerDriver,
Protos.TaskStatus taskStatus) {
        System.out.println("Status update: task "+taskStatus.
getTaskId().getValue()+" state is "+taskStatus.getState());}

    @Override
    public void frameworkMessage(SchedulerDriver schedulerDriver,
Protos.ExecutorID executorID, Protos.SlaveID slaveID, byte[] bytes) {
```

```
    }

    @Override
    public void disconnected(SchedulerDriver schedulerDriver) {

    }

    @Override
    public void slaveLost(SchedulerDriver schedulerDriver,
Protos.SlaveID slaveID) {

    }

    @Override
    public void executorLost(SchedulerDriver schedulerDriver,
Protos.ExecutorID executorID, Protos.SlaveID slaveID, int i) {

    }

    @Override
    public void error(SchedulerDriver schedulerDriver, String
message) {

    }
}
```

`MonteCarloScheduler.resourceOffers()` is the primary method in which the scheduler responds to the offers received from the Mesos master. Currently, the scheduler does nothing useful. By deploying this *empty* scheduler, we should be able to see all the log messages as different events happen.

Adding the framework launcher

The framework launcher is responsible for creating a framework driver instance to manage the framework life cycle. We will call our launcher class App that will have only the main method:

```
public class App {
    public static void main(String[] args) {
        System.out.println("Starting the MonteCarloArea on Mesos with
master "+args[0]);
        Protos.FrameworkInfo frameworkInfo =
Protos.FrameworkInfo.newBuilder()

.setName("MonteCarloArea")
    .build();
        MesosSchedulerDriver schedulerDriver = new
MesosSchedulerDriver(new MonteCarloScheduler(), frameworkInfo,
args[0]);
        schedulerDriver.run();
    }
}
```

Here, we are using `MesosScheculerDriver` to manage the life cycle. The `MesosSchedulerDriver` constructor requires an instance of our framework scheduler and an instance of `FrameworkInfo` describing our framework and the address of the Mesos master. The Mesos master address can be specified in any format accepted by Mesos, such as `host:port`, `zk://host1:port1/path/to/mesos/` or `file://path/to/file/containing/master/URI`. We can optionally pass credentials to the framework. `FrameworkInfo` just sets the framework name. Finally, we start and block the driver using the `run()` method.

Deploying our framework

Deploying a framework requires the following three steps:

1. Compiling and packaging the framework.
2. Installing the framework
3. Launching the framework

Now, let's build our code. We can issue a Maven build from IDE or use the `mvn` command-line interface, and run the following command:

```
ubuntu@master:~ $ mvn package
```

It will compile the framework code and build the jar in the `target/` folder. Once we have built your framework, to install the framework on the Mesos cluster, we need to make the framework binary accessible to all slaves. We can put it into HDFS, and use the `ExecutorInfo` parameter of the `MesosSchedulerDriver`'s constructor for telling Mesos where to look for it. We can set the URI field to point to the HDFS path of the executor. We can also specify the `frameworks_home` parameter in the Mesos slave, where all the framework executors are stored. In this case, `ExecutorInfo` can be set to the relative path and the `frameworks_home` value will be prefixed to the `ExecutorInfo` path by the slave. The other option is to use Docker to package our executors and use `DockerInfo` with the corresponding parameters.

To run the framework, we need to include the Mesos jar and protocol buffer jars in the classpath. One option is to build a *fat* jar that packages all the dependencies with our jar. The other option is to add them to the classpath while running, which is what we will use. We can use deployment tools, such as Chef or shell script, to launch our framework. We can also use other meta-frameworks to launch our framework, as we saw in the previous chapters. Tools, such as mesos-submit, (http://mesos.apache.org/documentation/using-the-mesos-submit-tool/) can be useful when you want to launch a command on nodes. Without such a tool, the developer machine will have to keep the scheduler process running. The mesos-submit is a framework that creates an executor and launches the actual scheduler, which takes over as the framework scheduler. This way mesos-submit scheduler can safely exit.

To launch our framework from the command line, run the following command:

```
ubuntu@master:~ $ java -cp MonteCarloArea.jar:mesos-
0.20.1.jar:protobuf-2.5.0.jar -Djava.library.path=libs/
```

```
org.packt.mesos.App "zk://master:2181/mesos"
```

Here, we have specified jars for the protocol buffer and Mesos Java bindings. Also, note that our Java bindings use native Mesos implementation and, thus, we need to know the location of `libmesos.so`, which in the preceding command is in the `libs` folder. Then, we specify the fully qualified name of our launcher class, `org.packt.mesos.App`. The final argument is the address of the Mesos master.

This will start the framework scheduler. You should see an output similar to the following:

```
I1005 14:06:28.488776  7710 sched.cpp:391] Framework registered with
20141005-130347-1739597066-5050-1453-0005
Scheduler registered with id 20141005-130347-1739597066-5050-1453-
0005
…
Scheduler received offers 2.
…
I1005 14:06:29.381525  7704 sched.cpp:730] Stopping framework
'20141005-130347-1739597066-5050-1453-0005'
```

The master output shows the framework registration and various events:

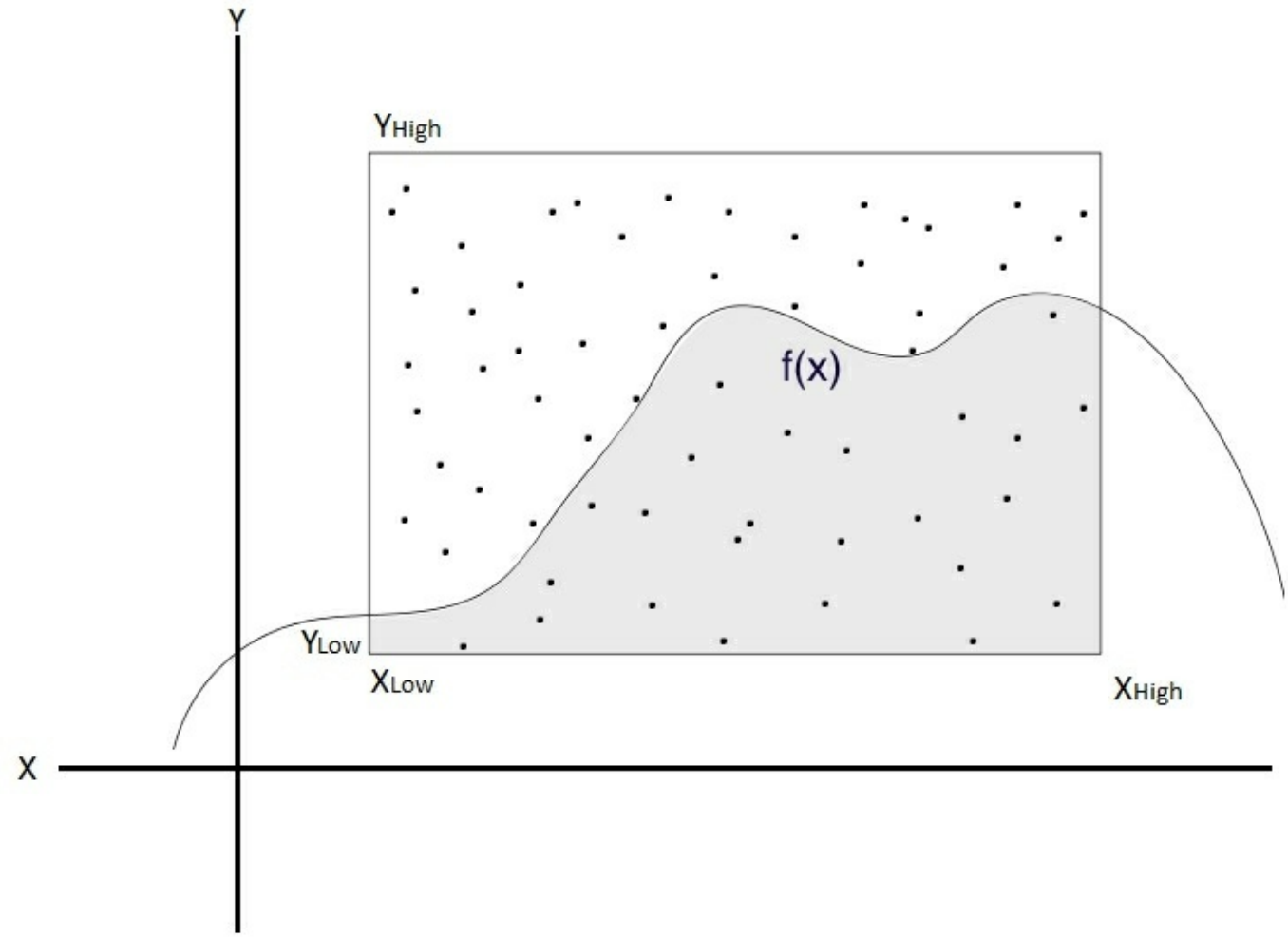
```
I1005 14:06:28.488001  1461 master.cpp:818] Received registration
request from scheduler(1)@10.37.176.103:54774
I1005 14:06:28.488085  1461 master.cpp:836] Registering framework
20141005-130347-1739597066-5050-1453-0005 at
scheduler(1)@10.37.176.103:54774
I1005 14:06:28.488234  1461 hierarchical_allocator_process.hpp:332]
Added framework 20141005-130347-1739597066-5050-1453-0005
I1005 14:06:28.488368  1461 master.cpp:2285] Sending 1 offers to
framework 20141005-130347-1739597066-5050-1453-0005
…
I1005 14:06:29.382316  1459 master.cpp:1034] Asked to unregister
framework 20141005-130347-1739597066-5050-1453-0005
I1005 14:06:29.382344  1459 master.cpp:2688] Removing framework
20141005-130347-1739597066-5050-1453-0005
I1005 14:06:29.382439  1459 hierarchical_allocator_process.hpp:408]
Deactivated framework 20141005-130347-1739597066-5050-1453-0005
I1005 14:06:29.382472  1459 hierarchical_allocator_process.hpp:363]
Removed framework 20141005-130347-1739597066-5050-1453-0005
```

The web interface will also show our framework, and slave logs also contain a list of status messages about the framework. *Ctrl + C* would exit our framework and show messages corresponding to stoping the framework.

Building our framework

Our framework is a very simple framework used to calculate the area under the curve in the given axis-parallel rectangle. We will use a numerical method that is the simplification of the Monte Carlo method to calculate the integral of a function. The following image illustrates the idea. To calculate the area under the curve in a given rectangle, we will take a lot of points from the rectangle and evaluate the curve equation at those points using the *x* coordinate. If the evaluation is less than the *y* coordinates of the point, the point is below the curve at that *x* coordinate. We will count all such points. The area under the curve can be calculated as,

Area under curve = Area of the rectangle * (pointsBelowCurve/totalPoints)



Axis-parallel rectangle with arbitrary function f(x)

The grey area is the value that we are interested in.

The accuracy of the AUC calculated by this method depends on how many points we take. The more points we take, the more accurate our area under the curve calculation will be. However, as we take more and more points, this will become a very resource-intensive task to perform on one machine. Thus, we will need more resources. We will divide the problem into smaller subproblems. We will divide the given rectangle into smaller rectangles, and use Mesos API to use resources seamlessly from multiple machines.

We will use a linear equation to represent our curve. We will use very simple strings to represent each term in the equation. Class term represents one term and expression represents our curve. We will use four coordinates to mark our axis-parallel: $\mathbf{X_{Low}}$, $\mathbf{X_{High}}$, $\mathbf{Y_{Low}}$, and $\mathbf{Y_{High}}$:

```
class Term{
    double coefficient;
    double exponent;

    Term(){
        coefficient=exponent=0;
    }

    Term(double coefficient, double exponent){
        this.coefficient=coefficient;
        this.exponent=exponent;
  }

    public static Term fromString(String term){
        double coefficient=1;
        double exponent=0;
        String[] splits=term.split("x",-1);

        if(splits.length>0) {
            String coefficientString=splits[0].trim();
            if(!coefficientString.isEmpty()) {
                coefficient = Double.parseDouble(coefficientString);
            }
        }

        if (splits.length>1) {
            exponent=1;
            String exponentString = splits[1].trim();
            if (!exponentString.isEmpty()) {
                exponent = Double.parseDouble(exponentString);
```

```
            }
        }
        return new Term(coefficient, exponent);
    }

    @Override
    public String toString() {
        return coefficient+"x^"+exponent;
    }

    public double evaluate(double x){
        return coefficient*Math.pow(x,exponent);
    }
}

public class Expression {
    List<Term> terms;

    public Expression(){
        terms=new ArrayList<Term>();
    }

    public Expression(List<Term> terms){
        this.terms=terms;
    }

    public boolean addTerm(Term term){
        return terms.add(term);
    }

    public double evaluate(double x){
        double value=0;
        for (Term term : terms) {
            value+=term.evaluate(x);
        }
        return value;
    }

    public static Expression fromString(String s){
        Expression expression=new Expression();
        String[] terms = s.split("\\+");
        for (String term : terms) {
            expression.addTerm(Term.fromString(term));
        }
        return expression;
    }

    @Override
    public String toString() {
```

```
        StringBuilder builder=new StringBuilder();
        int i;
        for (i=0; i<terms.size()-1; i++) {
            builder.append(terms.get(i)).append(" + ");
        }
        builder.append(terms.get(i));
        return builder.toString();
    }
}
```

The `fromString()` method in the term parses the given term. Note that our goal is not to write a very sophisticated parser, but to have a very simple version that might not work for an edge case. The `evaluate()` method in the term finds out the value of the term at a given X value. The `evaluate()` method in the expression class sums up the values of terms at a given X.

Adding an executor to our framework

Now, let's add our executor to calculate AUC in the rectangle. `org.packt.mesos.MonteCarloExecutor` is our executor class, which implements the executor interface. It stores the expression, coordinates of the rectangle, and the number of points to be used from the rectangle:

```
public class MonteCarloExecutor implements Executor{
    Expression expression;
    double xLow;
    double xHigh;
    double yLow;
    double yHigh;
    int n;

    public MonteCarloExecutor(Expression expression, double xLow,
double xHigh, double yLow, double yHigh, int n) {
        this.expression = expression;
        this.xLow = xLow;
        this.xHigh = xHigh;
        this.yLow = yLow;
        this.yHigh = yHigh;
        this.n=n;
    }

    @Override
    public void registered(ExecutorDriver executorDriver,
Protos.ExecutorInfo executorInfo, Protos.FrameworkInfo frameworkInfo,
Protos.SlaveInfo slaveInfo) {
        System.out.println("Registered an executor on slave " +
slaveInfo.getHostname());
}

    @Override
    public void reregistered(ExecutorDriver executorDriver,
Protos.SlaveInfo slaveInfo) {
        System.out.println("Re-Registered an executor on slave " +
slaveInfo.getHostname());
    }

    @Override
    public void disconnected(ExecutorDriver executorDriver) {
        System.out.println("Re-Disconnected the executor on slave");
    }

    @Override
    public void launchTask(final ExecutorDriver executorDriver, final
```

```
Protos.TaskInfo taskInfo) {
        System.out.println("Launching task
"+taskInfo.getTaskId().getValue());
        Thread thread = new Thread() {
            @Override
            public void run(){
                //Notify the status as running
    Protos.TaskStatus status = Protos.TaskStatus.newBuilder()

.setTaskId(taskInfo.getTaskId())
.setState(Protos.TaskState.TASK_RUNNING)
.build();
    executorDriver.sendStatusUpdate(status);
                System.out.println("Running task
"+taskInfo.getTaskId().getValue());
                double pointsUnderCurve=0;
                double totalPoints=0;

                for(double x=xLow;x<=xHigh;x+=(xHigh-xLow)/n){
                    for (double y=yLow;y<=yHigh;y+=(yHigh-yLow)/n) {
                        double value=expression.evaluate(x);
                        if (value >= y) {
                            pointsUnderCurve++;
                        }
                        totalPoints++;
                    }
                }
                double area=(xHigh - xLow)*(yHigh - yLow) *
pointsUnderCurve/totalPoints; // Area of Rectangle * fraction of
points under curve
                //Notify the status as finish
                status = Protos.TaskStatus.newBuilder()
                                .setTaskId(taskInfo.getTaskId())

.setState(Protos.TaskState.TASK_FINISHED)

.setData(ByteString.copyFrom(Double.toString(area).getBytes()))
                                .build();
                executorDriver.sendStatusUpdate(status);
                System.out.println("Finished task
"+taskInfo.getTaskId().getValue()+ " with area : "+area);
            }
        };

        thread.start();
    }

    @Override
```

```
    public void killTask(ExecutorDriver executorDriver, Protos.TaskID
taskID) {
        System.out.println("Killing task " + taskID);
    }

    @Override
    public void frameworkMessage(ExecutorDriver executorDriver,
byte[]bytes) {
    }

    @Override
    public void shutdown(ExecutorDriver executorDriver) {
        System.out.println("Shutting down the executor");
    }

    @Override
    public void error(ExecutorDriver executorDriver, String s) {

    }

    public static void main(String[] args) {
        if(args.length<6){
            System.err.println("Usage: MonteCarloExecutor
<Expression> <xLow> <xHigh> <yLow> <yHigh> <Number of Points>");
        }
        MesosExecutorDriver driver = new MesosExecutorDriver(new
MonteCarloExecutor(Expression.fromString(args[0]),Double.parseDouble(
args[1]), Double.parseDouble(args[2]), Double.parseDouble(args[3]),
Double.parseDouble(args[4]), Integer.parseInt(args[5])));
        Protos.Status status = driver.run();
        System.out.println("Driver exited with status "+status);
    }
}
```

Most of the callback methods have only print statements. The AUC calculation happens in `launchTask()`, and we need to add most of the code. `launchTask` should create a separate thread and do the following:

1. Notify Mesos about the status of the task to be run.
2. Run the task.
3. Notify Mesos about the status of the task when finished.

It first sends the status message that the task is running to Mesos. The MonteCarlo logic is simplified to use uniform points rather than random points. It uniformly samples points from the rectangle and evaluates the expression at that point. If the points lie

below the curve, it updates the `pointsUnderCurve` counter. The area under the curve in this rectangle is simply a fraction of `pointsUnderCurve/totalPoints` of the area of the rectangle. At the end, it updates the status to FINISHED and sends the result.

We use `MesosExecutorDriver` to manage the executor life cycle. After a sanity check for arguments, we create a driver instance and block it by invoking `run()`. `MesosExecutorDriver` takes the `Executor` instance as the argument. Note that the executor parses the expression and coordinates from the command-line arguments.

Updating our framework scheduler

Now, our executor is ready to calculate the AUC. In the scheduler, we need to create an executor and collect the results. Let's add the logic to this in our `MonteCarloScheduler` and `App` constructor.

We will add a Boolean flag `taskDone` to ensure that we only calculate the area once. We will also take command-line arguments in the launcher that have expressions and coordinates. We will store these arguments so that we can pass them to our executor. Now the `MonteCarloScheduler` constructor looks like the following:

```
public MonteCarloScheduler(String[] args){
this.args=args;
}
```

The `App` now passes them to the `MonteCarloScheduler` instance:

```
MesosSchedulerDriver schedulerDriver = new MesosSchedulerDriver(new
MonteCarloScheduler(Arrays.copyOfRange(args,1,args.length)),framework
Info,args[0]);
```

If the task is not done, `resourceOffers()` will accept the first offer, and launch a task with `MonteCarloExecutor`:

```
public void resourceOffers(SchedulerDriver schedulerDriver,
List<Protos.Offer> offers) {
        if (offers.size()>0 && !taskDone) {
            Protos.Offer offer = offers.get(0);

            Protos.TaskID taskID =
Protos.TaskID.newBuilder().setValue("1").build();
                System.out.println("Launching task " +
taskID.getValue()+" on slave "+offer.getSlaveId().getValue()+" with
"+task);
            Protos.ExecutorInfo executor =
Protos.ExecutorInfo.newBuilder()

.setExecutorId(Protos.ExecutorID.newBuilder().setValue("default"))

.setCommand(EXECUTOR_CMD+args)
.setName("Test Executor (Java)")
.build();

            Protos.TaskInfo taskInfo = Protos.TaskInfo.newBuilder()
```

```
.setName("MonteCarloTask-" + taskID.getValue())
.setTaskId(taskID)
.setExecutor(Protos.ExecutorInfo.newBuilder(executor))
.addResources(Protos.Resource.newBuilder()

.setName("cpus")

.setType(Protos.Value.Type.SCALAR)

.setScalar(Protos.Value.Scalar.newBuilder()

.setValue(1)))
.addResources(Protos.Resource.newBuilder()

.setName("mem")

.setType(Protos.Value.Type.SCALAR)

.setScalar(Protos.Value.Scalar.newBuilder()

.setValue(128)))
.setSlaveId(offer.getSlaveId())
.build();

schedulerDriver.launchTasks(Collections.singletonList(offer.getId()),
Collections.singletonList(taskInfo));
            taskDone=true;
        }
    }
```

Note

Note that we should also check whether the offer has sufficient resources before launching the task using it, but we have omitted this to keep it brief.

Here, the executor instance is built and holds the command to launch `MonteCarloExecutor`. For the sake of simplification, we have set `EXECUTOR_CMD` to "`java -cp MonteCarloArea.jar:mesos-0.20.1-shaded-protobuf.jar -Djava.library.path=./libs org.packt.mesos.MonteCarloExecutor`" but we should read from the external environment.

`Task` holds the resource requirement of the task and specifies which executor to use. It also specifies where to launch the task by setting the `slaveId` that it receives from the offer. Here, we launch a task with one CPU and 128 MBs memory. Finally, it calls `launchTasks()` on the driver. We can also use `launchTasks()` with filters to specify

the constraints that we have on the remaining resource offers. Note that `launchTasks` takes collections of `OfferId` and `TaskInfo` as arguments. Thus, we have created single element lists. The use of `launchTasks()` that accepts the `OfferId` and `TaskInfo` instances is deprecated.

The `statusUpdate()` method now checks for the `TASK_FINISHED` update, and, on receiving the update, prints the AUC value passed by the executor:

```
public void statusUpdate(SchedulerDriver schedulerDriver,
Protos.TaskStatus taskStatus) {
        System.out.println("Status update: task
"+taskStatus.getTaskId().getValue()+" state is
"+taskStatus.getState());
        if
(taskStatus.getState().equals(Protos.TaskState.TASK_FINISHED)){
            double area =
Double.parseDouble(taskStatus.getData().toStringUtf8());
            System.out.println("Task
"+taskStatus.getTaskId().getValue()+" finished with area : "+area);
            schedulerDriver.stop();
        } else {
            System.out.println("Task
"+taskStatus.getTaskId().getValue()+" has message
"+taskStatus.getMessage());
        }
    }
```

Now, let's run the updated framework. We now need to supply an argument to the curve and rectangle coordinates:

```
ubuntu@master:~ $ java -cp MonteCarloArea.jar:mesos-
0.20.1.jar:protobuf-2.5.0.jar -Djava.library.path=libs/
org.packt.mesos.App "zk://master:2181/mesos" "x" 0 10 0 10 100
```

This runs the framework to calculate the area under the curve for `y=x` line in the rectangle defined by `(0,0)`, `(10,0)`, `(0,10)`, and `(10,10)` using 100 points. The updated framework shows an additional output after registering the framework, which is similar to the following:

```
…
Launching task 1 on slave 20141220-113457-184623020-5050-1249-0 with
" x " 0.0 10.0 0.0 10.0 100
Status update: task 1 state is TASK_RUNNING
Task 1 has message
Status update: task 1 state is TASK_FINISHED
```

```
Task 1 finished with area : 50.495049504950494
```
…

The framework calculates the AUC value to be `50.495049504950494`, which is close to the exact value `50`. We can use more points to get a more accurate answer. Also, the slave log shows various events, such as task assigned to the slave, slave starting the container, and slave fetching the executor and starting it. We can also see the monitoring and status update messages. We will omit them due to space constraints.

Running multiple executors

Till now, we have only one executor that does all the computation and returns the area value. The real power of a distributed framework, such as Mesos, is manifested when having multiple tasks for a given job. Mesos' simple API makes it very easy to move from one executor to a multiexecutor scenario. Since our `MonteCarloExecutor` already computes the area under the curve in a given rectangle, it's very easy to use more computing power. We will divide the given rectangle into smaller rectangles, and ask `MonteCarloExecutor` to calculate the areas in the smaller rectangles. The scheduler is now responsible for dividing the job into smaller tasks. In our scenario, this means that `MonteCarloScheduler` will have the following responsibilities:

1. Divide the given rectangle into the required number of smaller rectangles. We will accomplish this in the `MonteCarloScheduler` constructor. We will store arguments needed for all the tasks in the `task` field. The `tasksSubmitted` and `tasksCompleted` counters are used to keep track of how many tasks are running and completed. `totalArea` holds the total number of AUC values from all the finished tasks:

```
public class MonteCarloScheduler implements Scheduler {
    private LinkedList<String> tasks;
    private int numTasks;
    private int tasksSubmitted;
    private int tasksCompleted;
    private double totalArea;

public MonteCarloScheduler(String[] args, int numTasks){
    this.numTasks=numTasks;
    tasks=new LinkedList<String>();
…
// code for field assignments
…
double xStep=(xHigh-xLow)/(numTasks/2);
double yStep=(yHigh-yLow)/(numTasks/2);

for (double x=xLow;x<xHigh;x+=xStep){
    for (double y=yLow;y<yHigh;y+=yStep) {
        tasks.add(" \" "+args[0]+" \" "+x+" "+(x+xStep)+" "+y+"
"+(y+ yStep)+" "+args[5]);
    }
}
```

2. If there are any pending tasks, distribute them to one of the executors, passing appropriate rectangle coordinates. We set `TaskId` and `ExecutorId` using the

`tasksSubmitted` counter:

```
for (Protos.Offer offer : offers) {
if(tasks.size()>0) {
// launch a task
}
}
```

3. Collect and aggregate the results as tasks finish. Print the final value of the area, once all the tasks are done, and shut down the driver:

```
if (taskStatus.getState().equals(
Protos.TaskState.TASK_FINISHED)){
    tasksCompleted++;
    double area =
Double.parseDouble(taskStatus.getData().toStringUtf8());
    totalArea+=area;
    System.out.println("Task"+taskStatus.getTaskId().getValue()+
" finished with area : "+area);
} else {
    System.out.println("Task
"+taskStatus.getTaskId().getValue()+"has message
"+taskStatus.getMessage());
}
if(tasksCompleted==numTasks){
    System.out.println("Total Area : "+totalArea);
    schedulerDriver.stop();
}
```

We will also add a new parameter to the command-line argument, and the second argument will specify how many tasks we want to run:

```
MesosSchedulerDriver schedulerDriver = new MesosSchedulerDriver(
new MonteCarloScheduler(
Arrays.copyOfRange(args,2,args.length),
Integer.parseInt(args[1])),
frameworkInfo,args[0]);
```

Now, let's launch our framework with these changes. We will run it for the same curve `y = x` in the same rectangle with four tasks:

```
ubuntu@master:~ $ java -cp MonteCarloArea.jar:mesos-
0.20.1.jar:protobuf-2.5.0.jar -Djava.library.path=libs/
org.packt.mesos.App "zk://master:2181/mesos" 4 "x" 0 10 0 10 100
```

We should see output similar to the following:

```
Scheduler registered with id 20141220-113457-184623020-5050-1249-0004
Launching task 1 on slave 20141220-113457-184623020-5050-1249-1 with
" x " 0.0 5.0 0.0 5.0 100
Launching task 2 on slave 20141220-113457-184623020-5050-1249-0 with
" x " 0.0 5.0 5.0 10.0 100
Status update: task 2 state is TASK_RUNNING
Task 2 has message
Status update: task 2 state is TASK_FINISHED
Task 2 finished with area : 0.0
Launching task 3 on slave 20141220-113457-184623020-5050-1249-0 with
" x " 5.0 10.0 0.0 5.0 100
Status update: task 1 state is TASK_RUNNING
Task 1 has message
Status update: task 1 state is TASK_FINISHED
Task 1 finished with area : 12.623762376237623
Status update: task 3 state is TASK_RUNNING
Task 3 has message
Status update: task 3 state is TASK_FINISHED
Task 3 finished with area : 25.0
Launching task 4 on slave 20141220-113457-184623020-5050-1249-1 with
" x " 5.0 10.0 5.0 10.0 100
Status update: task 4 state is TASK_RUNNING
Task 4 has message
Status update: task 4 state is TASK_FINISHED
Task 4 finished with area : 12.625
Total Area : 50.24876237623762
I1220 13:50:03.771772  1733 sched.cpp:747] Stopping framework
'20141220-113457-184623020-5050-1249-0004'
```

Note that scheduler has divided the larger rectangle into the following four rectangles and assigned them to different tasks:

- (0,0),(5,0),(5,0),(5,5)
- (0,5),(5,5),(0,10),(5,10)
- (5,0),(10,0),(5,5),(10,5)
- (5,5),(10,5),(5,10),(10,10)

Rectangle 1 has an AUC of `12.62`, 2 has `0`, 3 has `25`, and 4 has `12.62`. The combined area under the curve reported is `50.24`. Note that this value is closer to the exact value of 50, than the value we got while using only one task, `50.49`. Here, we have to run the framework on a small cluster with two slaves, so you can see that tasks 3 and 4 have to wait for tasks 1 and 2 to finish. We can try a larger number of tasks and would get a more accurate answer.

Advanced topics

We have seen how to implement a very simple framework on Mesos. Real-world frameworks rarely are so simple. In this section, we will briefly discuss some advanced topics and note down some precautions.

Reconciliation

Mesos does a great a job with hiding the nuances of building distributed applications and provides a simpler API. Although, Mesos provides a reliable communication mechanism for framework developers, the fact that the frameworks are distributed systems cannot always be ignored. Mesos has an actor-like programming model with message-passing, with a mix of *at-most-once* and *at-least-once* semantics. The messaging between the master and framework can have failures and lost messages. When this happens, the worldview of the framework and master will be out of sync. An example scenario would be where the framework sends a message to the master and the master dies before receiving the message or before sending the message to the slave. So the task assumes that the master is aware of the intended message, while the master has no idea about it whatsoever. To deal with such cases, there has to be some mechanism to unite the information on the master and the frameworks.

Mesos provides a reconciliation API (http://mesos.apache.org/documentation/latest/reconciliation/) to frameworks to take the appropriate actions whenever such failures are detected. The failure detection is done by the master and scheduler driver which is notified of a disconnection and reregistration. There are two types of reconciliations:

- Offer reconciliation
- Task reconciliation

Offer reconciliation happens automatically, as no offers are persisted, and on disconnection they are no longer valid. On reregistration, the offers are regenerated. Task reconciliation must be done explicitly by the framework on a failure, as the scheduler driver does not persist this information. On reregistering with the Mesos master, the framework should do task reconciliation. For all the tasks that the framework thinks are running, it sends a list of `TaskStatus` to the master. On receiving this message, the master examines the `TaskID` and `SlaveID` fields in `TaskStatus` and sends back the task status for each task in the list. If the status list is empty, the master will send the statuses of all the tasks currently known. The tasks that are no longer found will have the `TASK_LOST` update. The framework writers should perform the reconciliation for all nonterminal tasks, until an update is received for each task using an algorithm similar to the following:

```
startTime = currentTime();
remainingTasks = { All non-terminal tasks }
While(remainingTasks <> {}){
```

```
reconcile(remainingTasks)
wait for update with timeout truncated exponential backoff;
remainingTasks={all tasks in remainingTasks that has not been updated
since startTime}
}
```

Here, we loop till the `remainingTasks` list is nonempty and performs the reconciliation for tasks in the `remainingTasks` list. On each arrival of an update, the `remainingTasks` list is updated with tasks for which no updates have been received yet. As noted earlier, the reconciliation algorithm should be executed on each reregistration. The framework should also ensure that there is only one reconciliation in progress at any given time. This could be offloaded to the language client libraries in future so that the reconciliation is entirely transparent to the framework writers.

Stateful applications

The current design of the Mesos garbage collects the resources from the slave once the task is finished. Also, there is no guarantee that the task will be offered resources on the same slaves after a restart. This makes it difficult to run frameworks that require persistent storage, such as databases and distributed filesystems. There is an effort underway to extend Mesos to make it easier to develop stateful applications on Mesos. Currently, frameworks that require persistent disk use a non-managed location, such as `var/lib/cassandra` for writing data. This can lead to unpredictable behavior if multiple frameworks start using the same location. So, until more support for stateful application is added (https://issues.apache.org/jira/browse/MESOS-1554), it's best to ensure that no two frameworks contest for the same location.

Developer resources

As a developer, it is important to take a look at examples of other frameworks built on Mesos as well as other resources that are helpful while building Mesos frameworks.

Framework design patterns

We are still in the very early days of seeing a lot of Mesos frameworks, but common patterns have already started to emerge to see how different Mesos framework schedulers use resources.

We list some of these patterns:

- Resource mediators: These are frameworks that act just as resource mediators for other frameworks. Examples of such meta-frameworks include all the service frameworks that we have seen in Chapter 5, *Running Services on Mesos*. These frameworks are very general and, typically, they do have limited knowledge about applications for which they are mediating resources.
- Load-based: These are frameworks that adjust the resource usage based on the load on the framework. Scaling up and down in Marathon and Aurora based on constraints are examples of this behavior. Other examples in this category of frameworks include continuous integration frameworks, such as Jenkins on Mesos and Gitlab CI on Mesos that adjust the resource usage of the framework based on the job pending queue length. This category also includes other opportunistic frameworks.
- Reservation-based frameworks: These are frameworks that statically reserve resources while running. These frameworks require all the resources requested before they can proceed and hold on to them for the lifetime of the framework. Different frameworks that we saw in the first section of the book in Chapter 2, *Running Hadoop on Mesos*, Chapter 3, *Running Spark on Mesos,* and Chapter 4, *Complex Data Analysis on Mesos* are mostly examples of this category. With this pattern, existing distributed frameworks (such as existing distributed storage systems) are easy to port as frameworks on Mesos.

Note that these patterns are neither exhaustive nor exclusive. These are frameworks that combine many of these patterns, in addition there are frameworks that do not fit into any of these patterns. This is one of the main advantages of Mesos. It allows you to implement any strategy for the resource usage by providing the common substrate without restricting in any ways.

Framework testing

Testing distributed frameworks is challenging. However, Mesos makes the problem of testing a little easier, as it hides all the complexity of distributed messaging. For most frameworks, ensuring the correctness of an executor is the primary part of testing. Executor testing depends on the logic of the executor. In our case, we can test our executor by writing unit test cases for different curves and compare the outputs. The scheduler testing should make sure that the scheduler is working correctly in isolation. In our case, we need to write test cases that ensure that our work division among different tasks is correct. The most difficult framework testing is to ensure that the framework works correctly in the presence of other frameworks. One way to do this is to use Mesos API mocking and ensure the scheduler state after different callbacks is correct. We should note that most frameworks don't need this level of testing.

RENDLER

Mesosphere has created a web crawler Mesos framework, RENDLER (https://github.com/mesosphere/RENDLER,) that can be an excellent starting point for Mesos framework developers. RENDLER provides implementation in Go, Python, Scala, C++, and Java, and also provides a virtual machine environment for trying out the crawler framework.

Akka-mesos

Akka-mesos (https://github.com/drexin/akka-mesos) provides an Akka library to build Mesos frameworks. Akka (http://akka.io) is a toolkit used to simplify the development of reactive applications using actor-based programming. Akka-mesos provides constructs that aid in providing the common utility functions and patterns for Mesos frameworks.

Summary

In this chapter, we walked you through the Mesos API and implemented a framework to calculate the area under a curve for different functions. We also introduced some useful advanced topics and discussed how developer's resources are useful while developing frameworks on Mesos. In the next chapter, we will see the challenges involved in operating a Mesos cluster.

Chapter 8. Administering Mesos

This chapter is targeted toward system administrators and developers, and we will go through various tools to help you deploy and manage the Mesos cluster. We will also describe features that are very useful in highly available and multi-tenant Mesos deployments. We will conclude by exploring Mesos REST and CLI interfaces.

In this chapter, we will cover the following topics:

- Deployment
- Upgrade
- Monitoring
- Multitenancy
- High availability
- Maintenance
- Mesos interfaces

Deployment

Mesos provides the flexibility to run your Mesos clusters in a wide range of environments. It does not require any special hardware to run on and can be deployed in public clouds or in a private data center, on a wide range of operating systems. So, we can choose whatever environment suits our needs, and Mesos will run in that environment to solve our business problems efficiently.

Mesos build provides shell scripts to aid in the cluster deployment. `mesos-start-cluster.sh` and `mesos-stop-cluster.sh` can be used to start and stop the master and slave processes on nodes, as we have seen in Chapter 1, *Running Mesos*. Manual installations are not only cumbersome but also error-prone. Thus, scripts provide you with a convenient way to try out Mesos. Usage of automated deployment tools is highly recommended when running Mesos in production.

We also saw how to set up Mesos on Amazon Web Service in Chapter 1, *Running Mesos*. There is also the AWS CloudFormation template developed by the community. Mesosphere has detailed instructions on setting up Mesos on Google Compute Cloud and DigitalOcean (http://mesosphere.com/docs/getting-started/cloud/). Mesosphere also provides binary packages for various operating systems. Note that you can always use the manual method as well as automation tools to manage Mesos installation on the various cloud platforms.

Mesos provides many options for automated deployments on clusters. It integrates well with all the deployment tools that most operators have used over the years. There are plugins and/or support for most of the popular deployment tools that you can use with Mesos. There are Chef cookbooks, Puppet modules and, Ansible playbooks developed by the Mesos community at http://mesos.apache.org/documentation/latest/tools/ and https://github.com/dharmeshkakadia/awesome-mesos.

Upgrade

Mesos has a very frequent release cycle. Upgrading it is relatively simple:

- Install the new master binaries
- Restart the master process
- Install the new binaries on the slave
- Restart slave processes
- Upgrade the schedulers by linking to the new Mesos library
- Restart the schedulers

If required, the executors can be upgraded by linking the new Mesos library. As we have seen in Chapter 6, *Understanding Mesos Internals*, tasks survive slave processes and executor restarts. If a Mesos release needs an upgrade of the underlying operating system kernel, restarting the master and slave machines might be required. Mesos upgrade documentation (http://mesos.apache.org/documentation/latest/upgrades/) provides more information for upgrading to different versions.

Mesos infrastructure also requires you to maintain different frameworks so that they're up to date. For some frameworks, such as Spark, the upstream Spark project itself maintains Mesos schedulers and executors. For most frameworks, which are ported to Mesos, there is a separate project to maintain the Mesos port of the framework. In the long run, this becomes difficult to maintain in production. Mesos as the data center operating system needs a package manager that is similar to yum, apt, or brew so that the user can just describe what framework he or she wants to install, and the package manager takes care of the installation. At the time of writing, there is an ongoing discussion about maintaining framework packages for Mesos. Universe (https://github.com/mesosphere/universe) from Mesosphere is an example of a package repository.

Monitoring

Monitoring is a vital part of keeping any infrastructure running. Mesos integrates well with existing monitoring solutions and has plugins for most of the monitoring solutions, including Nagios, collectd (https://github.com/rayrod2030/collectd-mesos), and so on. This allows Mesos to leverage all the years of experience that operators have with these tools. The installation procedure of these plugins is completely dependent on the tool and is similar to any other plugin installation that does not require any knowledge of Mesos. The HTTP endpoint also gives the monitoring information. `http://<master>/metrics/snapshot` spits out a detailed resource report similar to the following code snippet:

```
{
    "master/cpus_percent":0,
    "master/cpus_total":2,
    "master/cpus_used":0,
    "master/disk_percent":0,
    "master/disk_total":32808,
    ...
    "registrar/queued_operations":0,
    "registrar/registry_size_bytes":405,
    "registrar/state_fetch_ms":8.523776,
    "registrar/state_store_ms":12.186112,
    ...
    "system/cpus_total":2,
    "system/load_15min":0.12,
    "system/load_1min":0.22,
    "system/load_5min":0.24,
    "system/mem_free_bytes":1739386880,
    "system/mem_total_bytes":2099142656
}
```

The Mesos community is very active and has developed many tools to help Mesos operators. Angstrom (https://github.com/nqn/angstrom) is a metric collection system for Mesos that also provides the global cluster state. It's written in Go and includes the web interface along with REST API. The API also provides you with access to the statistics based on time ranges as well. It collects aggregate statistics for the CPU, memory, and disk.

Container network monitoring

Starting from 0.20.0, Mesos supports network monitoring for each container (http://mesos.apache.org/documentation/latest/network-monitoring/). It provides network statistics that does not depend on the tasks running on the slave. The statistics are available at the `/monitor/statistics.json` endpoint of the slave.

Network monitoring support has been recently introduced and is currently only limited to Linux with kernel version above 3.6. This feature is built on top of the network namespace in Linux and requires the `libnl3` and `iproute` packages on the slaves (and `libnl3-devel` for building). The network monitoring is not enabled by default and must be enabled by specifying `--with-network-isolator`, while configuring the Mesos build:

```
ubuntu@master:~/mesos/build $ ../configure --with-network-isolator ;
make
```

With network monitoring turned on, each container will get a subset of the ports of the host. By default, the Linux ephemeral port range is `32768` to `61000`. Since ephemeral ports from the host get directly mapped to the containers, the host's ephemeral port range of the host should be confined or squeezed to a smaller range, leaving the rest of the ports to be used with containers running on the host safely. Here is an example of a host port squeezing `57344` to `61000`:

```
ubuntu@master:~ $ echo "57345 61000" >
/proc/sys/net/ipv4/ip_local_port_range
```

The hosts need rebooting to effect the changes. Now, ports `32768` to `57344` can be used by containers. While starting the slave, we need to include `network/port_mapping` in the `--isolation` flag and `--ephemeral_ports` in the resources flag along with rest of the options:

```
ubuntu@master:~ mesos/build/bin $ mesos-slave \
--isolation=cgroups/cpu,cgroups/mem,network/port_mapping \
    --resources=cpus:22;mem:62189;ports:[31000-
32000];disk:400000;ephemeral_ports:[32768-57344] \
    --ephemeral_ports_per_container=1024
\ #rest of the options
```

Also, `--ephemeral_ports_per_container` specifies how many ephemeral ports are allocated to each container. In the preceding example, we specified `1024` ports to be

allocated to each container. This number is recommended to be a power of 2 for performance reasons. Also, the number of containers might be limited by the total number of ports allocated divided by the ports per containers. Mesos exposes nonephemeral ports to the scheduler via resource `ports`, and it is the responsibility of the scheduler to allocate them. Once it is set up, network monitoring gives the following extra network statistics via the `/monitor/statistics.json` endpoint:

- net_rx_bytes
- net_rx_dropped
- net_rx_errors
- net_rx_packets
- net_tx_bytes
- net_tx_dropped
- net_tx_errors
- net_tx_packets

The introduction of per-container network stack (starting from 0.21 Version) also enables the rate limiting per container network usage. This can be useful to restrict executors from affecting other containers significantly. To enable this, we need to specify the `--egress_rate_limit_per_container` flag with the desired egress rate (in bytes per second), such as `--egress_rate_limit_per_container=500KB`.

Multitenancy

The ability to share infrastructure resources and frameworks among multiple users is extremely important for a data center kernel. In a large deployment, without proper support for multitenancy, separate static clusters have to be created for different users, organizations, or use cases, losing all the benefits Mesos provides as the common substrate for the data center. Mesos has first-class support for multitenancy, as we will see in this section.

Authorization and authentication

Authentication is an important feature for large Mesos deployments. Mesos added support for framework authorization starting from 0.20.0 release (http://mesos.apache.org/documentation/latest/authorization/). It uses **Simple Authentication and Security Layer** (**SASL**) library to implement the authentication that supports various authentication mechanisms. This allows:

- Frameworks to register and reregister with authorized roles
- Frameworks to launch tasks with authorized users
- Authorized *principals* to shut down frameworks using the `shutdown/` endpoint of HTTP API

Mesos implements authorization through **Access Control Lists** (**ACL**). For each interaction with the Mesos master, the master will check whether the request ACL is authorized to make such a request. If it is not authorized, an error message is sent. Each ACL entry specifies three things. The way to think about them is, *Subjects* can take *Actions* on a set of *Objects*.

1. Actions: Currently, the supported actions are as follows:
    - `register_frameworks`: This action can register or reregister frameworks. When a framework tries to register or reregister with a master, it supplies ACLs as part of `FrameworkInfo`. Master will check whether `FrameworkInfo.principal` is authorized to receive offers for the role specified by `FrameworkInfo.role`.
    - `run_tasks`: This action can run tasks or executors. The ACLs are checked to see whether `FrameworkInfo.principal` is authorized to run the task/executor as the specified user.
    - `shutdown_frameworks`: This action can shut down frameworks. The ACLs are checked to ensure that the principal is authorized to shut down the framework.
2. Subjects: Currently, this supports only *principals*. Principals can be framework principals used by the `register_frameworks` and `run_tasks` actions, or it can be usernames used by the `shutdown_frameworks` action.
3. Objects: Currently, the supported objects are as follows:
    - `roles`: Roles that can be specified by a framework for resources, used by the `register_frameworks` action
    - `users`: The username that will be used to launch tasks/executors, used by the `run_tasks` action

- `framework_principals`: Framework principals that can be shut down by HTTP endpoint

The ACLs are matched in the order; so the first matching ACL will determine whether the request has authorization or not. If no ACLs match, `ACLs.permissive` determines the authorization. If the field is set to true, then the nonmatched request is authorized. By default, it is set to `true`. The ACLs are enabled by specifying a JSON file, describing the rules to the master. The following is an example of the ACLs files:

```
{
  "register_frameworks": [
    {
      "principals": { "values": ["foo"] },
       "roles": { "values": ["developer","operation"] }
    },
    {
      "principals": { "values": "bar" },
      "roles": { "values": ["barOnlyRole"] }
    },
    {
      "principals": { "type": "NONE" },
      "roles": { "values": ["barOnlyRole "] }
    }
  ],

  "run_tasks": [
    {
      "principals": { "values": ["a", "b"] },
      "users": { "values": ["abUser"] }
    },
    {
      "principals": { "values": [ "c" ] },
      "users": { "values": ["cUser"] }
    },
    {
      "principals": { "values": [ "c" ] },
      "users": { "type": "NONE" }
    },
    {
      "principals": { "type": "ANY" },
      "users": { "values": ["guest"] }
    },
    {
      "principals": { "type": "NONE" },
      "users": { "values": ["root"] }
    }
```

```
  ],

  "shutdown_frameworks": [
    {
      "principals": { "values": ["a", "b"] },
      "framework_principals": { "values": ["c"] }
    },
  ]
}
```

In the preceding configuration file, the `foo` framework can register with the developer and operational roles. The principal `bar` can register with the `barOnlyRole` role, and no one else can register with `barOnlyRole`. The `a` and `b` frameworks can run a task as `abUser`. `c` can run tasks only as `cUser` and no other user. Any framework can run a task as `guest`, and no one can run tasks as `root`.

Here is another example:

```
{
  "permissive": "false",
  "register_frameworks": [
    {
       "principals": { "values": ["foo"] },
       "roles": { "values": ["fooOnlyRole"] }
    },
   ],
  "shutdown_frameworks": [
    {
  "principals": { "values": ["ops"] },
  "framework_principals": { "type": "ANY" }
    }
   ]
}
```

The preceding example shows when `permissive` is set to `false`. Here, the `foo` principal can only register with the `fooOnlyRole` role and no other roles. Also, no other framework will be allowed to register with any other roles. Only the `ops` principal can shut down any framework using an HTTP endpoint.

Advanced features, such as adding authentication support for slave registration, Kerberos integration, encrypting all the network messages, and so on, are planned in future Mesos releases. Also, at the time of writing, all the messages in Mesos are sent on wire unencrypted, which may be a concern in some environments. There is work underway to have all the messages use SSL/TLS

(https://issues.apache.org/jira/browse/MESOS-910).

Tip

If we want to enable multiple UNIX users to submit to the same cluster, we need to run the Mesos slaves as the root, otherwise they will fail to setuid.

API rate limiting

The multi-framework environment that Mesos encourages for better cluster utilization, might lead to poor performance of some frameworks if other frameworks behave in a greedy manner. Different frameworks have different **Service Level Agreements** (**SLAs**) and very different natures of processing (batch, service, and so on). So, a framework can keep the master busy with its own messages, preventing the master from processing messages from other frameworks, thus resulting in starvation for them. To prevent this from happening, Mesos supports the framework API rate limiting feature (http://mesos.apache.org/documentation/latest/framework-rate-limiting). Operators can specify the rate value for a framework that determines how many API requests per second will be processed by the master from a particular framework. The outstanding messages are stored in the Mesos master memory.

When a framework exceeds the capacity, `FrameworkErrorMessage` will be sent to the framework. This will cause the scheduler driver to abort and calls an error callback. Note that this does not kill any tasks or the scheduler itself. If a framework does not assume that all the messages to the master are processed, then the framework can choose to restart or failover the scheduler and continue working.

Framework rate limiting configuration is JSON-based and has the following parameters:

- `principal`: This identifies the framework or the group of frameworks for which it is throttled.
- `qps`: This is queries per second or the throttling rate.
- `capacity`: This controls the number of outstanding messages from the given `principal` that are allowed to be queued in the master memory. If `qps` is not specified, then the capacity value is ignored. If `qps` is specified, but capacity is left unspecified, the capacity is assumed unlimited.

Rate limiting is enabled by specifying a configuration file with the `--rate-limits` flag on the master. Note that this configuration is loaded by the master on startup and is retained until the master terminates. Here is an example of a configuration file:

```
{
  "limits": [
    {
      "principal": "a",
      "qps": 100
      "capacity": 100000
```

```
    },
    {
      "principal": "b",
      "qps": 500
    },
    {
      "principal": "c",
    }
  ],
  "aggregate_default_qps": 5000,
  "aggregate_default_capacity": 1000000
}
```

Here, there are three frameworks, `a`, `b` and, `c`, that have defined limits; the rest of the frameworks combined have `5000` queries per seconds and 1 million outstanding messages limitation. The framework `a` will be throttled at `100` QPS and will be allowed `100000` messages pending on the master, while the framework `b` will have `500` QPS and will be allowed infinite messages on the master. The framework `c` will not be rate limited. `aggregate_default_qps` and `aggregate_default_capacity` control the aggregate QPS and capacity for all the frameworks for which the limit is not specified. If these fields are not mentioned in the configuration, the rest of the frameworks are not throttled.

The capacity value needs to be configured carefully. If the capacity value is too low, then the resources will be underutilized, while if the capacity value is too high, the queued messages can cause the master to run out of memory. The number of messages that can be handled by the master is dependent on the memory limits of the master, framework, message size, and so on. Mesosaurus (https://github.com/mesosphere/mesosaurus) is a benchmarking framework for Mesos that can be used to simulate load and various framework behaviors of a Mesos cluster. Note that at the time of writing, the tool is under development, but it already has many useful features. We can start by setting the capacity value to a small number, so that it can tolerate a surge in framework messaging, and increase if required.

The master exposes counters for all the messages received and processed for that framework in the metrics via HTTP endpoints. The `http://<master>/metrics/framework/frameworkA/messages_received` and `http://<master>/metrics/framework/frameworkA/messages_processed` endpoints give the number of messages received and processed by the master from the `frameworkA` framework. The framework can monitor these counters to get an idea of the message queue length for the framework. Normally, the difference between the two

counters should not be very significant. A framework might see a large difference if it is being throttled.

Although the rate limiting feature is very useful, at the time of writing, it's still immature, and you need to be cautious before using it in production. Also, there is a proposal to allow the framework to slow down the message rate itself when the master sends them a soft alert. This way, the framework scheduler will have a chance to avoid error messages if it throttles down itself.

High availability

Most businesses depend on the availability of the infrastructure for continuous business operations. Mesos as a data center kernel not only provides high availability to the applications, but also has several features to ensure that various Mesos components are resilient to failures.

Master high availability

Mesos high availability depends critically on the availability of the Mesos master. As noted previously, Mesos uses ZooKeeper to ensure that one Mesos master is always available even in face of failures. The ZooKeeper elects the acting master, and the other master will be on standby. We have seen how to run Mesos with ZooKeeper in Chapter 6, *Understanding Mesos Internals*.

ZooKeeper configuration, at the minimum, should set the `dataDir`. Its value must be set to the persistent storage directory and never to the default value `/tmp` given in the sample ZooKeeper configuration file. In multiserver settings, a ZooKeeper instance determines its identity reading the `myid` field. ZooKeeper looks for the `myid` file in the data directory containing only a single line that is the machine's unique identity (for example, "1"). The ZooKeeper configuration options are detailed in the ZooKeeper documentation (https://zookeeper.apache.org/). Apart from the admin documentation on the ZooKeeper website, the book *ZooKeeper: Distributed Process Coordination* by Flavio Junqueira and Benjamin Reed covers the various best practices for operating ZooKeeper in production.

Tip

It is recommended that you use different ZooKeeper clusters for the Mesos master leader election from the ZooKeeper cluster used by other systems to avoid having the entire Mesos infrastructure down because other systems overwhelmed ZooKeeper.

The Mesos master process commits suicide when it loses connection with ZooKeeper. Mesos can survive a failure of N-1 masters with N size quorum (which requires 2N -1 master processes). To ensure high availability, it is recommended that you use three or five masters, which can survive one or two master failures respectively. It's important to ensure that the number of Mesos masters running is only 2N-1. Running more master processes may end up corrupting the replicated log. This can be enforced by external tools or can be set up with the Mesos master whitelist (https://issues.apache.org/jira/browse/MESOS-1546). Different ZooKeeper servers as well as different Mesos masters should be run in independent failure zones if possible.

Operators may need to change the quorum size as the Mesos cluster grows or shrinks. While replacing the master or increasing the quorum size, it is important to start with the empty replicated log. Currently, changes in the quorum size are only possible by restarting the cluster. You can refer to the online reconfiguration of the quorum size at

https://issues.apache.org/jira/browse/MESOS-683. Here is an example of the steps used to move from quorum size two to three by increasing the number of masters from three (say, A, B, C) to five (say, A, B, C, D, and E).

1. Stop the current master nodes A, B, and C that are currently running with the --quorum=2 flag.
2. Change the quorum to three for original masters A, B, and C by restarting them with --quorum=3.
3. Start a new master node D and E one by one, with an empty log, with --quorum=3, and let them catch up on the replicated log.

The steps for decreasing the quorum size are similar, as noted in the Mesos operation guide (http://mesos.apache.org/documentation/latest/operational-guide).

Note that the Mesos master cannot restart by itself. It relies on an external supervisor daemon, such as runit (http://smarden.org/runit) and monit (https://mmonit.com/monit), or on manual restart. There are many small features in Mesos that prevent code bugs or operator errors from bringing down the Mesos cluster. For example:

- The Mesos master commits suicide when it loses connection with the ZooKeeper. This prevents the master from operating on a stale state.
- The Mesos slaves ignore shutdown requests from master processes that are currently not the leading masters.
- Limiting slave removal, which we will see in the next section.

Slave removal rate limiting

Mesos supports rate limiting slave removal. For example, in a hypothetical scenario, a buggy master process looses all the slave acknowledgement messages and starts to shut them down. Without rate limiting, the Mesos cluster will lose all the slaves in a matter of seconds. With the introduction of rate limiting, the idea is to slow down these kinds of behaviors so that the operators can get a chance to remedy the situation. In this scenario, the operator might notice the slave going down via monitoring and might be able to prevent the unavailability by killing the buggy master process. Mesos supports two types of rate limiting while slave removal:

- The percentage of slaves being removed while failover, which is controlled via the `--recovery_slave_removal_limit` flag. This flag specifies the maximum percentage of slaves that can be removed from the registry and shut down when the master fails over. For example, a value of 10 percent means that on a failover, if the master cannot connect to more than 10 percent of the slaves, it will shoot off itself rather than removing the slaves. The default value is 100 percent, which means that there will be no limit on the slave removal while failover.
- Rate at which the slaves are being removed is controlled via the `--slave_removal_rate_limit` flag. This flag specifies the maximum number of slaves that can be removed over time in the form of a number of slaves/duration. For example, the value of 3/40 minutes means that at maximum, two slaves can be removed in a 40 minutes window. By default, slaves will be removed as soon as they fail health checks. This feature was added in Mesos 0.22 Version.

Slave recovery

Mesos supports the seamless upgrade of the Mesos slaves (http://mesos.apache.org/documentation/latest/slave-recovery/). The ability to update the slave without impacting applications running on the slave is important, as it allows slaves to be kept up to date without incurring any downtime. The executors can keep running while the slave process is down and can reconnect to the slave once it comes back up again. This provides resilience against the slave failures as well, is especially important for stateful services, which may take a long time to warm up, and can have a significant impact on the service availability.

The recovery is implemented by checkpointing information about running tasks and executors to the local disk. The Mesos slave process on restart would be able to recover from the checkpointed information and reconnect to the executors. The recovery is done only if slaves have checkpointing enabled and for frameworks that want recovery benefits. The frameworks with checkpointing enabled will receive resource offers from only those slaves, which have checkpointing enabled. The framework can choose to disable checkpointing for reasons, such as IO overhead.

Tip

Note that, in theory, it's possible to run multiple slave processes on a single slave node, but this is not recommended. A slave process can manage all the resources on a single node and having two slave processes adds unnecessary overhead.

By default, checkpointing is disabled on the slaves. To enable this on a slave, specify `--checkpoint` while starting the slave process. The `-recover` flag specifies whether the slave will reconnect to the running executors (`--recover=reconnect`) or kill them (`--recover=cleanup`). If no checkpointing information exists, the slave registers with the master as a new slave. The recovery by default will abort in 15 minutes. If the recovery needs more time, the `--recovery_timeout` flag can be used to specify the amount of time while waiting. Any errors during recovery will cause the slave process to fail. If the `--strict=false` flag is specified, the slave recovery errors are ignored. To enable checkpointing in frameworks, the `FrameworkInfo.checkpoint` field should be set to `true` while registering with the master.

Note

Note that starting from Version 0.22, checkpointing is enabled by default on slaves, and

the `--checkpoint` flag is removed from the slave options.

The checkpointing feature also makes the status updates more reliable. The executor driver caches all the status updates sent by the executor and sends them to the slave when it reconnects with the restarted slave. So, all the frameworks would benefit from this transparently. The executors can continue sending the status update as normal. Without the slave checkpointing the status updates, during the master failover, the updates might get lost. This is because the Mesos master is stateless and a failed over master reconstructs the state from the information provided by slaves. This is also true even if executors do terminate when the slave is down. In a rare case, where the status update is lost, state reconciliation aids the recovery process, as described in the previous chapter.

Maintenance

Maintenance allows the slaves to be removed from the cluster for performing hardware upgrades or maintenance tasks, such as upgrading the operating system on the slave. Maintenance features in Mesos provide you with the ability to inform frameworks about the slave that wants to go through maintenance so that frameworks do not launch new tasks on the slave. Mesos can drain the tasks from a set of slaves and provides you with a way to notify the frameworks about future draining. This allows frameworks to take a preemptive action and smoother operation while draining.

Tip

Note that at the time of writing, the maintenance feature is still under heavy development.

The following are the steps to be performed by a slave for maintenance.

1. A slave who wants to be drained starts mentioning his intentions in the resource offers.
2. The Mesos master stops offering these resources to frameworks.
3. Mesos sends inverse offers to frameworks that are running tasks on the slave.
4. The framework on receiving inverse offers can decide to reschedule the tasks running on the slave to other nodes if possible.
5. The slave drains all the running tasks in a given amount of time.
6. The slave sends unregistered events to the Mesos master notifying that it's going down. This way, the master knows about the disappearance of the slave node immediately, rather than having to wait for timeout.
7. Slave nodes can now undergo any upgrades safely.
8. After the slave maintenance is finished, the slave, once again, starts sending resource offers to the master and will be part of the cluster.

Once the slave is scheduled for maintenance, the offers from the slave will have the `resource_expiry_time` field set, which means that resources in these offers will not be valid after that time. These resources can be requested again via *inverse offers*. *Inverse offers* are requests to the framework to return the resources for the specified slaves within the given deadline. Once the slave is drained, the master will mark the slave as *deactivated*. Once the maintenance on a slave is finished, the operator can notify the Mesos master using `unschedule`, after which the master will be send normal

offers to the slave.

There is work underway to expose the maintenance primitive via the following HTTP endpoints:

- `/maintenance/schedule`: This will schedule the slave for the maintenance. The following is an example of a POST request, which asks for both `slave1` and `slave2` to be scheduled for maintenance at the specified time:

```
{
  "schedule": [
    { "hostname" : "slave1", "time" : "2015-11-04T20:00:00" },
    { "hostname" : "slave2", "time" : "2015-11-04T22:00:00" },
   ]
}
```

- `/maintenance/unscheduled`: This notifies the completion of the maintenance of the slave:

```
{
  "unschedule": [
    { "hostname" : "slave1" },
    { "hostname" : "slave2" }
   ]
}
```

- `/maintenance/status`: This can be used to get the current status of the maintenance.

Mesos interfaces

Mesos external interfaces provide you with ways to integrate other tools with Mesos.

The Mesos REST interface

Mesos provides a REST interface to interact with Mesos. We can explore the REST API through a browser or through command-line tools, such as curl. The following table lists the REST endpoints provided by Mesos. All endpoints return data in the JSON format by default:

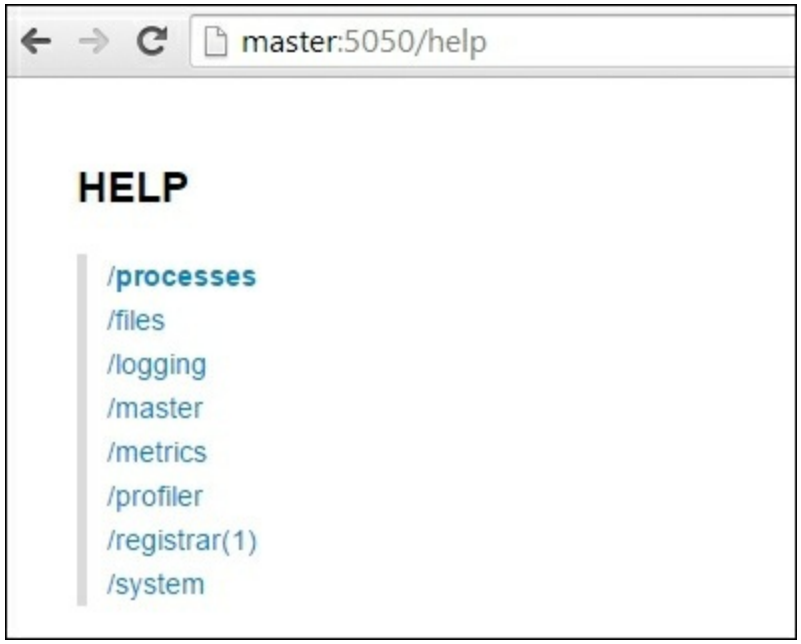
Command	**Use**
`/__processes__`	This lists the processes on the cluster

`/files/browse`
`/files/debug`
`/files/download`
`/files/read`
	Mesos files REST interface is used for browsing, reading, and so on

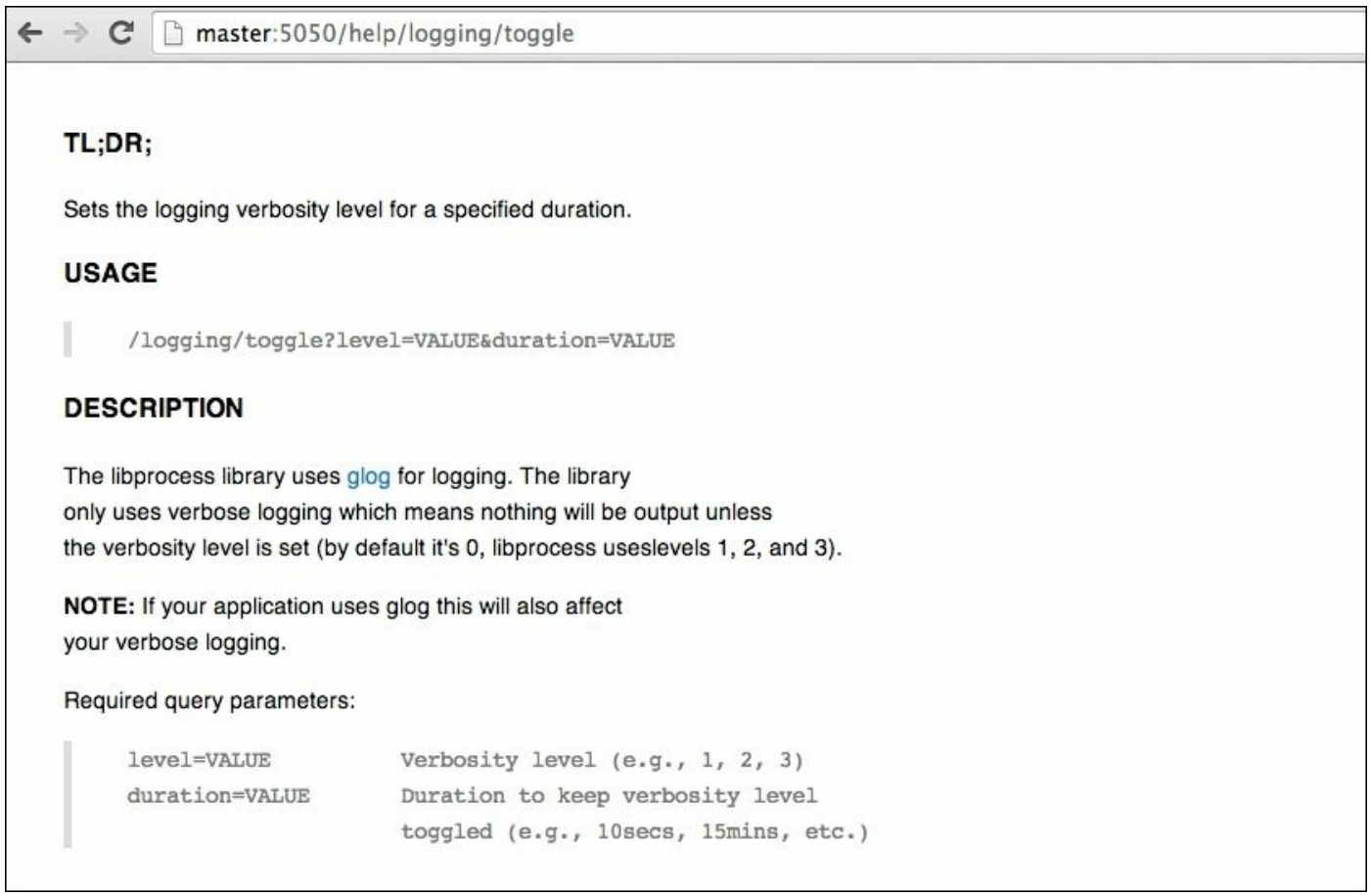
`logging/toggle`	This toggles the logging mode for the given period
`/master/health`	This returns the master health status as HTTP code. Return code 200 specifies that the master is healthy
`/master/redirect`	This redirects to the currently active master using HTTP 307
`/master/observe`	This receives the health information of hosts that provide a comma-separated list of hosts, name of monitors, and levels of health
`/master/roles`	This lists currently assigns roles
`/master/state`	This is the current state of the cluster, including all the information on frameworks, slaves, and so on
`/master/stats`	This is the summary of statistics of a cluster usage
`/master/tasks`	This lists all the tasks from all active frameworks
`/master/shutdown`	This shuts down a framework whose frameworkId has been provided
`/registrar(1)/registry`	This returns the content of the registry

`/profiler/start` `/profiler/stop`	This starts/stops the Mesos profiler
`/metric/snapshot`	This provides a snapshot of the current metrics

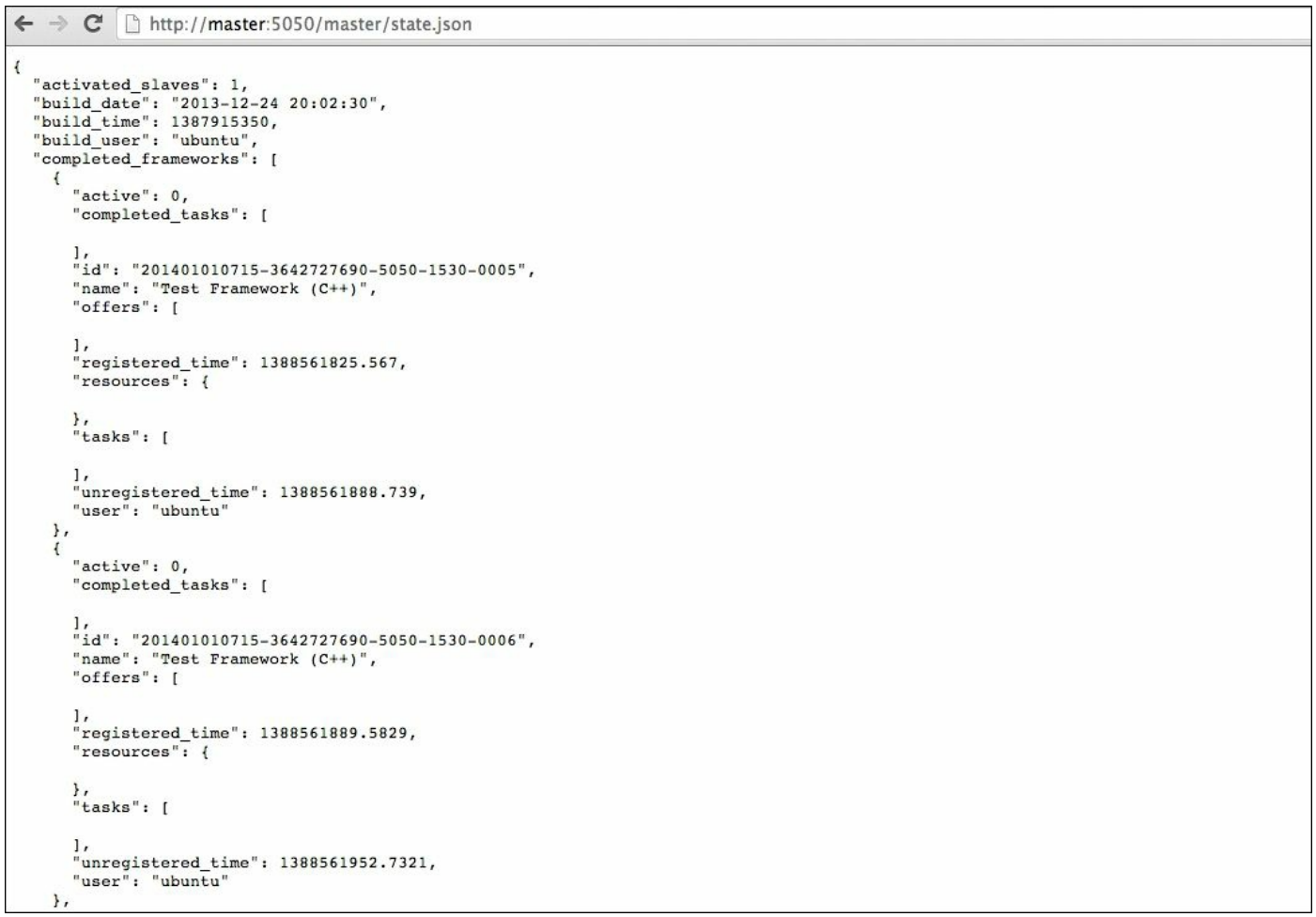
The Mesos web interface is built on top of REST API. All the help pages are themselves available as endpoints. You can list all the available REST endpoints by navigating to `http://master:5050/help`:



We can get more detailed help about an endpoint by clicking on the endpoint links. For example, you can check the toggle documentation for more details by navigating to `http://master:5050/help/logging/toggle/`:



For example, the master stat information can be obtained by navigating to `http://master:5050/master/state.json`:



Mesos configuration options common to master and slave

The Mesos CLI

The command-line interface has been one of the most important arrows in the quiver when dealing with large-scale infrastructure. The Mesos community has developed a powerful CLI for Mesos (https://github.com/mesosphere/mesos-cli).

One of the difficulties while debugging problems in Mesos deployments is that Mesos treats all the resources in the cluster as equal, without a specific identity, and it can be difficult, and at times impossible, to glue information from the multiple tools and interface. For example, performing actions, such as SSHing the slave sandbox, which is running a particular task, or knowing which nodes are running tasks of the given framework, can be very hard and repetitive. The Mesos CLI is meant to streamline these workflows. It is task-centric and most commands take `taskID` as an argument. Lazy matching ensures that we don't have to type the full lengthy IDs and partial matches return all the matching results. It has an expected behavior like UNIX commands and works well with piping input and output to other commands. The Mesos CLI has been written in Python and can be installed using `pip`:

```
ubuntu@master:~ $ pip install mesos.cli
```

Note

Note that the binary name is `mesos` and would be installed in `/usr/local/bin`. So, if we have Mesos binary installed on the machine, we can remove Mesos binary, or we can change the installation location using `--install-option="--prefix=$PREFIX_PATH"`.

Mesos CLI has a lot of very useful features. One of the most useful features is the familiar command completion. Mesos CLI completes almost every Mesos entities, such as taskids, filenames, and so on. To enable the completion, we need to run the following commands:

- For bash:

  ```
  ubuntu@master:~ $ complete -C mesos-completion mesos
  ```

- For zsh:

  ```
  ubuntu@master:~ $ source mesos-zsh-completion.sh
  ```

We can add this to the shell startup script. The Mesos CLI configuration is driven by a JSON file. By default, Mesos CLI searches for the configuration file in the following

locations, and an alternative location can be specified via `MESOS_CLI_CONFIG`:

```
./.mesos.json
~/.mesos.json
/etc/.mesos.json
/usr/etc/.mesos.json
/usr/local/etc/.mesos.json
```

Here is an example of a configuration file:

```
{
    "profile": "default",
    "default": {
        "master": "zk://localhost:2181/mesos",
        "log_level": "warning",
        "log_file": "/tmp/mesos-cli.log"
    // http or https. Scheme to use to connect to Mesos
    "scheme": "http"
    }
}
```

Mesos CLI does not require Mesos to be installed locally and can be informed where the Mesos master is using the following command:

```
ubuntu@master:~ $ mesos config master zk://localhost:2181/mesos
```

CLI supports profiles that are useful to manage multiple Mesos clusters using CLI. Switching configuration is achieved with just one command:

```
ubuntu@master:~ $ mesos config profile new-profile
```

`mesos [cmd] --help` shows you all the options. The following table lists down the various commands:

Command	Description

`mesos ls taskID path`
 	
`mesos find taskID file [file]`
 	
`mesos cat taskID file [file]`
 	
`mesos head`
	This is similar to ls, find, cat, head, and tail running from the task's sandbox.

`taskID file [file]`
 	
`mesos tail taskID file [file]`	

`mesos scp file [file] remotePath`	This copies local file(s) to remotePath on every slave. We will need an SSH access on all the slaves.
`mesos ssh taskID`	This will SSH to the sandbox that is running the specified task. We will need an SSH access on the slave node running the task for this to work.
`mesos events`	This observes events happening in the entire cluster, master, and all the slaves. You can specify the time interval between successive polling using -s (default interval is 5 sec).
`mesos config [key] [value]`	This outputs the Mesos CLI JSON configuration. If only a key is specified, it returns the value of the parameter. If both key and value are specified, it sets the value of a parameter key. A key can be any valid configuration parameter.
`mesos resolve [master-config]`	This returns the address of the current leading master.
`mesos state [slave]`	This returns the full JSON, describing the state of the master or matching slaves.

All the subcommands are separate scripts. We can also invoke all the subcommands directly by prefixing `mesos` to it, for example, `mesos cat` is equivalent to `mesos-cat`. The `mesos` script scans `PATH` and everything starting with `mesos-` would be treated as a subcommand. We can add our own subcommands in a language of our choice by naming our command `mesos-somecmd`, which will invoke the `somecmd` script from `PATH`. `mesos-config`, `mesos-resolve`, and `mesos-state` provide recurring constructs, and can be used to build new commands.

Configuration

We will describe all Mesos configuration parameters with their descriptions and default values. The latest list of configuration parameters is maintained at http://mesos.apache.org/documentation/latest/configuration. The options are first searched in environment variables and then on the command line. The environment variable names are the same as options, but with `MESOS_` prefixed to the uppercase option name. For example, the command line option of `--cluster` becomes the environment variable `MESOS_CLUSTER`. The units to measure time duration are user-friendly units, such as seconds, minutes, hours, and so on.

The following options are common to both the master and slave:

Option	Description	Default value
`--ip`	This is the IP address to which the master process binds. It is recommended that you use this option when running multiple network cards on a machine.	This is the default interface IP
`--port`	This is the port to listen on.	`5050`
`--log_dir`	This is the path to write log files.	There is no default value and no logs will be written if unspecified
`--logbufsecs`	This specifies the log buffering interval.	0
`--logging_level`	This log messages at or above this level. The possible levels are `INFO`, `WARNING`, and `ERROR`.	`INFO`
`--external_log_file`	This specifies the externally managed log file, which will be exposed in the web interface and HTTP API. This is useful when using stderr for logging, as the log file is otherwise unknown to Mesos.	
`--hooks`	This is a comma-separated list of hook modules to be installed on the master.	
`--modules`	This is a list of modules to be loaded into the internal subsystem. This can be a JSON string or path to the file containing the JSON string.	

`--hostname`	This is the hostname announced by the master and slaves. When left unset, the system hostname is used.	
`--quiet or --no-quiet`	This controls the logging behavior. `--quiet` disables logging to stderr.	`false`
`--version`	This shows the Mesos version	

The Mesos master

The following table is a list of Mesos master configuration options, which can run the `mesos-master --help` command.

Option	Description	Default value
`--quorum`	This is the quorum size to be used when using replicated log-based registry. The number determines the number of masters that have to agree on the state written to the registry. You should set this value to be a number greater than numberOfMasters/2	
`--work_dir`	This is the path where all the frameworks, work directories, and replication logs will be stored. This is a required parameter.	
`--zk`	This is the ZooKeeper URL that is used for leader election among masters. This parameter is required in HA deployment.	
`--zk_session_timeout`	This is the timeout for ZooKeeper sessions.	10 sec
`--log_auto_initialize`	This checks whether to initialize the replicated log automatically. If set to false, the replicated log has to be manually initialized before the first use.	`true`
`--acls`	This is the ACL configuration in form of JSON. The value can be a JSON string or path to the file containing the JSON string.	
`--allocation_interval`	This is the amount of time to wait between batch allocations.	1 sec
`--authenticate`	This checks whether unauthenticated frameworks will be allowed to register. If set to true, unauthenticated frameworks will not be allowed to register.	`false`
`--authenticate_slaves`	This checks whether unauthenticated slaves will be allowed to register. If set to true, unauthenticated slaves will not be allowed to register.	`false`
	This is the authenticator implementation to be used for	crammd5 (Challenge Response

`--authenticators`	authenticating slaves and frameworks. An alternate module can be specified via the modules flag.	Authentication Mechanism – Message Digest Algorithm)
`--credential`	This is the path to the text or the JSON file containing the credentials information.	
`--cluster`	This is the human-readable name for the Mesos cluster displayed on the web interface.	
`--framework_sorter`	This is the policy to allocate resources to a given user's frameworks.	`drf`
`--user_sorter`	This specifies the resource allocation policy among users.	`drf`
`--offer_timeout`	This is the time duration before a resource offer is revoked from a framework. This is useful when running frameworks that drop offers or hold on offers.	
`--rate_limits`	This specifies the rate limit configuration in JSON format. The value can be the JSON string or the file that has the JSON string, specifying the rate limit configuration.	
`--recovery_slave_removal_limit`	This specifies the maximum percentage of slaves that can be removed while failover.	100 percent
`--slave_removal_rate`	This specifies the maximum rate at which slaves can be removed by the master. The value is of the form (number of slaves)/duration.	Not set
`--registry`	This is the persistence strategy to be used for registry. The possible values are `replicated_log` or `in_memory`.	`replicated_log`
`--registry_fetch_timeout` and `--registry_store_timeout`	This is the amount of time to wait while fetching/storing data from the registry, after which the attempt is declared as, failed.	Fetch defaults to 1 min and store defaults to 5 sec
`--registry` or `--no-registry_strict`	If `--no-registry_strict`, the registrar will never reject admission, readmission, or removal of a slave. `--no-registry_strict` can be used to bootstrap on the cluster that is already running. Note that, this flag is marked as experimental.	`--no-registry_strict`

`--roles`	This is a list of roles that the frameworks in the cluster can assume, separated by a comma. For example, `--roles="test,prod"`.	
`--root_submissions` or `--no-root_submission`	This specifies whether or not root is allowed to submit frameworks.	`true`
`--slave_reregister_timeout`	This is the timeout within which slaves are expected to reregister, when a new master is elected. Slaves who do not reregister will be removed from the registry. Any attempts of communicating master after this will be shutdown. The value should be set to at least 10 minutes.	10 minutes
`--webui_dir`	This specifies the directory path to web UI files.	
`--weights`	This is a comma-separated list of `role=weight` pairs, indicating priorities. A user's dominant share is divided by weight. So, a weight of 2 ensures twice as much resources as one with weight of 1. Within a role, the framework that is furthest from its weighted fair share is given resources. For example, `--weights="test=1,prod=2"`.	
`--whitelist`	This specifies the path to the file that contains the list of slaves to which the offers should be made.	By default there is no whitelist which means all the slaves are accepted.
`--resource_monitoring_interval`	This is the monitoring interval for an executor resource usage.	1 second
`--max_executors_per_slave`	This is the maximum number of executors allowed per slave. This is used with a network isolator.	

Mesos common configuration options of master and slave

The Mesos slave

The following table lists Mesos slave options, which can also be obtained by running the `mesos-slave --help` command.

Option	Description	Default value
`--attributes`	This is a comma-separated list of attributes associated with the node, which can be used to constraint scheduling. Attributes can be thought of as tags and are specified as "`name:value`" pairs. For example, `--attributes="rack:r1, special_hardware:1,attr1:value1"`.	No attributes
`--master`	This specifies the master to connect to. It can be an IP address or hostname of one or more masters or ZooKeeper URI. You can also specify the path to the file containing any of the previous two. This is a required option. For example, `--master=localhost:5050 --master=10.0.0.10:5050,10.0.0.20:5050 --master=zk://user:password@10.0.0.11:2181/mesos --master=/etc/mesos/zk`	
`--checkpoint` or `--no-checkpoint`	This checks whether to checkpoint the slave state and framework information to a disk. Checkpointing allows slaves to recover in the event of slave restarts.	`true`
`--work_dir`	This is the location where framework work directories are placed.	`/tmp/mesos`
`--recover`	This specifies the recovery strategy, which can be either reconnect or cleanup. Reconnect means that the slave will reconnect any old live executors. The recover option is used only if `--checkpoint` is true, otherwise no recovery is performed and slaves register as new slaves with the master. Cleanup means that any running executors from a previous slave process will be killed.	reconnect
`--strict` or `--no-strict`	This checks whether recovery errors are considered fatal. If set to true, all recovery errors are considered fatal. If set to false, any expected errors during recovery are ignored and the slave tries to recover as much state as possible.	`true`

--recovery_timeout	This is the amount of time to wait for slaves to recover. After this duration, any executors that fail to reconnect to the slave will self-terminate.	15 minutes
--isolation	This is the isolation mechanism to be used. The possible values are posix/cpu,posix/mem To use cgroups as isolator, set to "cgroups/cpu,cgroups/mem".	posix/cpu,posix/me
--cgroups_enable_cfs or --no-cgroups_enable_cfs	This enables hard limits on CPU resources via CFS bandwidth limiting sub-feature.	false
--cgroups_hierarchy	This is the cgroups hierarchy root path.	/sys/fs/cgroup
--cgroups_limit_swap	This enables the memory limits on both memory and swap, instead of just memory.	false
--cgroups_root	This is the name of the root cgroup.	mesos
--containerizers	This specifies a comma-separated list of containerizer implementations. Currently, the supported containerizers are Mesos, External, and Docker. The order specifies the order in which they will be tried. For example, --containerizers="mesos,docker".	mesos
--containerizer_path	This specifies the location of the external containerizer executable. It is required when using --isolation=external.	
--default_container_image	This is the default image used by an external containerizer (when a task doesn't specify the image). It only works with --isolation=external. For example, —default_container_image=docker:/// libmesos/Ubuntu:13.10	
--default_container_info	ContainerInfo that will be included in ExecutorInfo of the tasks that does not explicitly specify a ContainerInfo. The value is specified in JSON format.	
--docker	This is the absolute path to the Docker executable for Docker containerizer.	docker
--docker_remove_delay	This is the amount of time to wait before removing Docker containers.	6 hours

--docker_sandbox_directory	This is the absolute path for the directory in the container, where the sandbox is mapped to.	/mnt/mesos/sandbox
--docker_stop_timeout	This is the time taken, as a duration, for Docker to wait after stopping an instance before killing.	0 sec
--resources	This is the total number of advertised resources on the slave per role. Using roles, we can restrict the resources used by the role. The format is "resource(role):value". Ex. --resources="cpus(prod):16;cpus(test):2;disk(*):"	
--default_role	Any resources in the --resources flag that do not specify a role will be assigned to this role. Also, resources that are automatically detected but are not present in the --resources flag will get this role.	It defaults to "*", indicating that all the roles have access to the resource.
--disk_watch_interval	This is the periodic time interval used to check the disk usage.	1 minute
--container_disk_watch_interval	This is the interval between disk quota checks for containers. This flag is used by posix/disk isolator.	15 seconds
--enforce_container_disk_quota or --no-enforce_container_disk_quota	This enables the disk quota enforcement for containers. This flag is used by the posix/disk isolator.	false
--executor_registration_timeout	This is the time duration for an executor to register with a slave, after which it's considered hung.	1 min
--executor_shutdown_grace_period	This is the time duration for an executor to shut down.	5 sec
--frameworks_home	This is the path that is prepended to relative executor URIs.	
--gc_delay	This is the maximum amount of time to wait before cleaning up executor directories. The gc might happen before this time, depending on the disk usage.	1 week
--gc_disk_headroom	This adjusts the disk headroom used to calculate the maximum executor directory age, which is calculated by: gc_delay * max(0.0, (1.0 - gc_disk_headroom - disk usage)) every --disk_watch_interval duration.	0.1

`--hadoop_home`	This is the path to `HADOOP_HOME` that is used to fetch executors from HDFS. This has no default value.	
`--launcher_dir`	This specifies the location of the Mesos binaries.	`/usr/local/lib/mes`
`--registration_backoff_factor`	If all the slaves try to register or re-register with master at once, a network storm can happen. To avoid that, each slave initially waits for a random time between `[0,registration_backoff_factor]` to connect to the master. Subsequent retries are exponentially backed off over this interval (nth retry is over interval `[0,registration_backoff_factor*2^n]` up to maximum of 1 min.	1 second
`--credential`	This is the text of a JSON file containing the credentials information.	
`--authenticatee`	This is the authenticate implementation used for authenticating slaves and frameworks. An alternate module can be specified via the modules flag.	crammd5 (Challenge Response Authentication Mechanism – Messag Digest Algorithm)
`--perf_interval`	This is the interval between subsequent perf stat sample invocations.	1 minute
`--perf_duration`	This is the duration of a perf stat sampling. This must be less than the `perf_interval` value.	10 seconds
`--perf_events`	This is a list of comma-separated perf events to sample for each container, when using the `perf_event` isolator. Event names are sanitized by downcasing and replacing hyphens with underscores when reported in the `PerfStatistics` protobuf, for example, cpu-cycles becomes cpu_cycles. A list of events can be obtained using `perf list` command.	
`--resource_monitoring_interval`	This is the periodic time interval used to monitor an executor resource usage.	1 second
`--slave_subsystems`	This is a comma-separated list of cgroup subsystems to run the slave binary, such as memory, cpuacct, and so on. The current functionality is intended for resource monitoring and no cgroup limits are set, they are	

	inherited from the root mesos cgroup.	
`--switch_user` or `--no-switch_user`	This checks whether to run the tasks as the user who submitted them, rather than the user running the slave.	`true`
`--ephemeral_ports_per_container`	This is the number of ephemeral ports allocated to a container by the network isolator. The value must be a power of 2.	`1024`
`--eth0_name --lo_name`	This is the name of public and loopback network interfaces. If left unspecified, the network isolator will guess it.	
`--egress_rate_limit_per_container`	This is the limit on the egress traffic for each container in bytes/sec. If the value is 0 or unspecified, the network isolator will not impose any limits on the containers' egress traffic.	
`--network_enable_socket_statistics`	This checks whether to collect socket statistics for each container or not.	`false`

Mesos slave configuration options

Mesos build options

While building from a source, the following options can be used to control the Mesos build to adapt to given deployment conditions. All the options can also be listed using the `configure --help` command.

The following options control building of some features of Mesos:

Option	**Description**	**Default value**
`--enable-shared`	This checks whether to build shared libraries.	`true`
`--enable-static`	This checks whether to build static libraries.	`true`
`--enable-fast-install`	This checks whether to optimize for the fast installation.	`true`
`--disable-`		

`libtool-lock`	This disables libtool locking.	`false`
`--disable-java` `--disable-python`	This disables building Java or Python bindings.	`false`
`--enable-debug` `--enable-optimize`	This checks whether to enable debugging and optimizing flags.	`false`
`--disable-bundled`	This checks whether to build using preinstalled dependencies instead of libraries bundled with Mesos.	`false`

`--disable-bundled-distribute`
`--disable-bundled-pip`
`--disable-bundled-wheel`
	This checks whether to exclude building and using the bundled `distribute`, `pip`, and `wheel` packages respectively instead of the installed version in PYTHONPATH.	`false`

`--disable-python-dependency-install`	This checks whether to install Python packages during `make install`.	`false`

The following table lists some packaging related flags:

Option	**Description**	**Default value**
`--with-gnu-ld`	This assumes that the C compiler uses GNU linker	No

`--with-zookeeper`
`--with-leveldb`
`--with-glog`
`--with-protobuf`
`--with-gmock`
	This excludes building and using bundled versions of ZooKeeper, LevelDB, glog, protocol buffer, or gmock, and instead use the version located in the directory specified by the respective flag.	No

`--with-curl`		

`--with-sasl`
`--with-zlib`
`--with-apr`
`--with-svn`
	This specifies the location of CURL, SASL, ZLib, APR, or SVN packages.	

`--with-sysroot`	This searches for dependent libraries in the specified directory.	
`--with-network-isolator`	This checks whether to build a network isolator.	`false`

Apart from the preceding options, the following environment variables influence the build configuration. They are useful when build against the nonstandard installation location for various packages.

Environment variable	**Description**
`JAVA_HOME`	This is the location where JDK is installed
`JAVA_CPPFLAGS`	These are the preprocessor flags passed to **Java Native Interface** (**JNI**)
`JAVA_JVM_LIBRARY`	This is the location where `libjvm.so` can be found
`MAVEN_HOME`	This is the location where maven is installed
`PROTOBUF_JAR`	This is the absolute path of the protocol buffer jar
`PYTHON`	This specifies which Python interpreter to be used
`PYTHON_VERSION`	This specifies which Python version to be used

Summary

In this chapter, we took a look at the various features Mesos had to offer for production usage. We also ventured into various tools and the best practices for managing Mesos clusters. This concludes the book. Let's build scalable applications!

Index

A

- Access Control Lists (ACL) / Authorization and authentication
- advanced configuration, Hadoop on Mesos
  - about / Advanced configuration for Hadoop on Mesos
  - task resource allocation / Task resource allocation
  - metrics reporting / Metrics reporting
  - authentication / Authentication
  - container isolation / Container isolation
  - configuration parameters / Additional configuration parameters
- advanced topics
  - about / Advanced topics
  - reconciliation / Reconciliation
  - stateful applications / Stateful applications
- Airbnb
  - about / Airbnb
- Akka
  - about / Akka-mesos
  - URL / Akka-mesos
- Akka-mesos
  - URL / Akka-mesos
  - about / Akka-mesos
- allocation module
  - about / Allocation module
  - URL / Allocation module
- allocator module
  - about / Mesos modules
- Amazon Elastic Compute Cloud (EC2)
  - about / Mesos cluster on Amazon EC2
- Angstrom
  - URL / Monitoring
- anonymous modules
  - about / Mesos modules
- Apache Cassandra
  - URL / Complex data and the rise of the Lambda architecture, Cassandra on Mesos

    - about / Complex data and the rise of the Lambda architecture, Cassandra on Mesos
    - running, on Mesos / Cassandra on Mesos
    - configuration options / Cassandra on Mesos
- Apache Hama
    - URL / Complex data and the rise of the Lambda architecture
- Apache incubator project Myriad
    - URL / An introduction to Hadoop
- Apache Kafka
    - URL / Complex data and the rise of the Lambda architecture
- Apache Portable Runtime Library (APRL) / Installing prerequisites
- Apache Samza
    - about / Complex data and the rise of the Lambda architecture
    - URL / Complex data and the rise of the Lambda architecture
- Apache Storm
    - about / Storm
    - URL / Storm
    - on Mesos / Storm on Mesos
- application groups, Marathon / Application groups
- application health checks, Marathon / Application health checks
- artifact store, Marathon
    - about / The artifact store
- Aurora
    - about / Aurora
    - URL / Aurora
    - components / Aurora
    - client / Aurora, An Aurora client
    - state machine / Aurora
    - event bus / Aurora
    - event bus subscribers / Aurora
    - storage / Aurora
    - storage architecture / Aurora
    - storage interface / Aurora
    - master-slave architecture / Aurora
    - features / Aurora
    - job life cycle / Job life cycle
    - running / Running Aurora
    - cluster configuration / Aurora cluster configuration

    - job configuration / Aurora job configuration
    - client commands / An Aurora client
    - URL, for documentation / An Aurora client
    - example / An Aurora example
    - cron jobs / Aurora cron jobs
- authentication modules
    - about / Mesos modules
- auxiliary services
    - about / Auxiliary services
    - shared filesystem / Auxiliary services
    - consensus service / Auxiliary services
    - service fabric / Auxiliary services
    - operational services / Auxiliary services
- AWS Management Console
    - URL / Mesos cluster on Amazon EC2

B

- Berkeley Data Analytics Stack (BDAS)
  - URL / Introducing Spark
- binary packages, Mesos
  - Homebrew, for Mac / Mac OS
  - Fedora / Fedora
- Bulk Synchronous Processing (BSP) / Complex data and the rise of the Lambda architecture

C

- capacity parameter
  - about / API rate limiting
- case studies, Mesos
  - Twitter / Twitter
  - HubSpot / HubSpot
  - Airbnb / Airbnb
- Cassandra
  - URL / Cassandra
- Cassandra Query Language (CQL) / Cassandra on Mesos
- CentOS
  - about / CentOS
- Chronos
  - about / Chronos
  - URL / Chronos
  - REST API / The Chronos REST API
  - APIs / The Chronos REST API
  - example / A Chronos example
- client commands, Aurora
  - Job / An Aurora client
  - Task / An Aurora client
  - Sla / An Aurora client
  - quota / An Aurora client
  - cron / An Aurora client
  - config / An Aurora client
  - beta-update / An Aurora client
- Cloudera Distribution Hadoop (CDH) / Installing Hadoop on Mesos
- cluster computing frameworks
  - about / Cluster computing frameworks
- cluster configuration, Aurora
  - name / Aurora cluster configuration
  - slave_root / Aurora cluster configuration
  - zk / Aurora cluster configuration
  - zk_port / Aurora cluster configuration
  - schedulerzkpath / Aurora cluster configuration
  - scheduler_uri / Aurora cluster configuration
  - proxy_url / Aurora cluster configuration

    - auth_meachanism / Aurora cluster configuration
- CLUSTER operator / Constraints
- coarse-grain mode, Spark on Mesos
    - URL / Tuning Spark on Mesos
- commands, Mesos
    - mesos-local.sh / Build Mesos
    - mesos-tests.sh / Build Mesos
    - mesos.sh / Build Mesos
    - gdb-mesos-* / Build Mesos
    - lldb-mesos-* / Build Mesos
    - valgrind-mesos-* / Build Mesos
    - mesos-daemon.sh / Build Mesos
    - mesos-start-cluster.sh / Build Mesos
    - mesos-stop-cluster.sh / Build Mesos
    - mesos-start-masters.sh / Build Mesos
    - mesos-stop-masters.sh / Build Mesos
    - mesos-start-slaves.sh / Build Mesos
    - mesos-stop-slaves.sh / Build Mesos
- communication
    - about / Communication
- community, Mesos
    - about / The Mesos community
- complex data
    - about / Complex data and the rise of the Lambda architecture
- Complex Event Processing (CEP) / Complex data and the rise of the Lambda architecture
- configuration
    - about / Configuration
    - Mesos master / The Mesos master
    - Mesos slave / The Mesos slave
    - Mesos build options / Mesos build options
- configuration options, Mesos-DNS
    - Masters / Mesos-DNS configuration
    - refreshSeconds / Mesos-DNS configuration
    - ttl / Mesos-DNS configuration
    - domain / Mesos-DNS configuration
    - port / Mesos-DNS configuration
    - resolvers / Mesos-DNS configuration

    - timeout / Mesos-DNS configuration
    - email / Mesos-DNS configuration
- configuration parameters
    - reference / Configuration
- configuration parameters, Spark on Mesos
    - spark.cores.max / Tuning Spark on Mesos
    - spark.mesos.extra.cores / Tuning Spark on Mesos
    - spark.mesos.executor.home / Tuning Spark on Mesos
    - spark.mesos.executor.memoryOverhead / Tuning Spark on Mesos
- considerations, data center kernel
    - scalability / The Mesos architecture
    - flexibility / The Mesos architecture
    - maintainability / The Mesos architecture
    - utilization and dynamism / The Mesos architecture
    - fairness / The Mesos architecture
- constraints, in Marathon
    - about / Constraints
    - operators, used for specifying / Constraints
- container network monitoring
    - about / Container network monitoring
- containers command
    - about / External containerizer
- cron jobs, running with Aurora
    - URL / Aurora cron jobs
- CRUD (Create, Update, Read, Delete) operations
    - about / The artifact store
- CSV
    - about / CSV

D

- decorator
  - about / Mesos hooks and decorators
- deployment
  - about / Deployment
- destroy command
  - about / External containerizer
- developer resources
  - about / Developer resources
  - framework design patterns / Framework design patterns
  - framework testing / Framework testing
  - RENDLER / RENDLER
  - Akka-mesos / Akka-mesos
- Directed Acyclic Graph (DAG) execution engine
  - about / Introducing Spark
- Distributed File System (DFS) / An introduction to Hadoop
- Distributed Shared Memory (DSM) systems / Introducing Spark
- DNS (Domain Name Service) / Service discovery
- DNS-based discovery
  - about / Service discovery
- DNS record generator
  - about / Mesos-DNS
- DNS resolver
  - about / Mesos-DNS
- Docker
  - URL / Packaging, Docker containerizer
  - about / Docker containerizer
- Docker containerizer
  - about / Docker containerizer
- Dominant Resource Fairness (DRF)
  - about / The Mesos scheduler
  - properties / The Mesos scheduler
  - URL / The Mesos scheduler
- DStream / Spark Streaming
- dynamic reservation
  - about / Dynamic reservation

E

- Elasticsearch
  - about / Complex data and the rise of the Lambda architecture
  - URL / Complex data and the rise of the Lambda architecture
- event bus, Marathon
  - about / Event bus
- Executor API
  - about / Communication
- executor API
  - about / The executor API
  - void registered(ExecutorDriver, ExecutorInfo, FrameworkInfo, SlaveInfo) / The executor API
  - void reregistered(ExecutorDriver, SlaveInfo) / The executor API
  - void disconnected(ExecutorDriver) / The executor API
  - void launchTask(ExecutorDriver, TaskInfo) / The executor API
  - void killTask(ExecutorDriver, TaskID) / The executor API
  - void shutdown(ExecutorDriver) / The executor API
  - void error(ExecutorDriver driver, String message) / The executor API
- ExecutorDriver API
  - about / The ExecutorDriver API
  - Status start() method / The ExecutorDriver API
  - Status stop() method / The ExecutorDriver API
  - Status abort() method / The ExecutorDriver API
  - Status join() method / The ExecutorDriver API
  - Status sendStatusUpdate(TaskStatus status) / The ExecutorDriver API
  - Status sendFrameworkMessage(byte[] data) / The ExecutorDriver API
- extensibility, Mesos / Extending Mesos
- External containerizer (EC)
  - about / External containerizer
- External Containerizer Program (ECP)
  - about / External containerizer
  - MESOS_LIBEXEC_DIRECTORY / External containerizer
  - MESOS_WORK_DIRECTORY / External containerizer
  - MESOS_DEFAULT_CONTAINER_IMAGE / External containerizer
- external containerizers
  - about / External containerizer
- Extract-Transform-Load (ETL) jobs / Chronos

F

- fault tolerance
  - about / Fault tolerance
  - ZooKeeper / ZooKeeper
  - failure detection and handling / Failure detection and handling
  - registry / Registry
- Fedora
  - about / Fedora
- fine-grained mode, Spark on Mesos
  - about / Tuning Spark on Mesos
  - URL / Tuning Spark on Mesos
- framework design patterns
  - about / Framework design patterns
  - resource mediators / Framework design patterns
  - load-based / Framework design patterns
  - reservation based / Framework design patterns
- frameworks
  - about / Frameworks

G

- Graphite
  - about / Graphite
  - URL / Graphite
- GROUP_BY operator / Constraints

H

- Hadoop
  - about / An introduction to Hadoop
  - URL / An introduction to Hadoop
  - example job / An example Hadoop job
- Hadoop Distributed File System (HDFS)
  - about / An introduction to Hadoop / Spark on Mesos
- Hadoop documentation
  - URL / An introduction to Hadoop
- Hadoop MapReduce
  - about / An introduction to Hadoop
- Hadoop on Mesos
  - about / Hadoop on Mesos
  - URL / Installing Hadoop on Mesos
  - installing / Installing Hadoop on Mesos
  - advanced configuration / Advanced configuration for Hadoop on Mesos
- HDFS (Hadoop Distributed File System)
  - about / Complex data and the rise of the Lambda architecture
- HDFS as Mesos framework
  - URL / An introduction to Hadoop
- HierarchicalDRF / The Mesos scheduler
- high availability
  - about / High availability
  - master high availability / Master high availability
  - slave removal rate limiting / Slave removal rate limiting
  - slave recovery / Slave recovery
- hook
  - about / Mesos modules, Mesos hooks and decorators
- HubSpot
  - about / HubSpot

I

- Internet of Things (IoT) / Complex data and the rise of the Lambda architecture
- isolator modules
  - about / Mesos modules

J

- Java Native Interface (JNI)
  - about / Mesos build options
- Java package naming conventions
  - URL / Module naming
- job object, attributes
  - about / Aurora job configuration
  - task / Aurora job configuration
  - role / Aurora job configuration
  - Environment / Aurora job configuration
  - Cluster / Aurora job configuration
  - instances / Aurora job configuration
  - priority / Aurora job configuration
  - production / Aurora job configuration
  - update_config / Aurora job configuration
  - health_check_config / Aurora job configuration
  - constraints / Aurora job configuration
  - contact / Aurora job configuration
  - max_task_failures / Aurora job configuration
  - service / Aurora job configuration
- JobTracker
  - about / An introduction to Hadoop

K

- Kafka on Mesos
  - URL / Complex data and the rise of the Lambda architecture
- Kubernetes project
  - URL / Packaging

L

- labelling
  - about / Task labels
- Lambda architecture
  - URL / Complex data and the rise of the Lambda architecture
  - about / Complex data and the rise of the Lambda architecture
  - batch layer / Complex data and the rise of the Lambda architecture
  - speed layer / Complex data and the rise of the Lambda architecture
  - serving layer / Complex data and the rise of the Lambda architecture
- launch command
  - about / External containerizer
- LevelDB
  - about / Registry
  - URL / Registry
- LIKE operator / Constraints

M

- mailing lists
  - about / Mailing lists
- maintenance
  - about / Maintenance
  - steps / Maintenance
- Marathon
  - about / Marathon
  - URL / Marathon
  - API / The Marathon API
  - running / Running Marathon
  - example / Marathon example
  - constraints / Constraints
  - event bus / Event bus
  - artifact store / The artifact store
  - application groups / Application groups
  - application health checks / Application health checks
- master high availability
  - about / Master high availability
- MesLeast Recently Used (LRU) / Garbage collection
- Mesos
  - about / Introducing Mesos
  - master / The master
  - slave / Slaves
  - framework / Frameworks
  - features / Why Mesos?
  - prerequisites, installing / Installing prerequisites
  - building / Build Mesos
  - commands / Build Mesos
  - starting / Start Mesos
  - running, Vagrant used / Running Mesos using Vagrant
  - community / The Mesos community
  - mailing lists / Mailing lists
  - Spark Streaming, running on / Running Spark Streaming on Mesos
  - flow of events / Resource allocation
  - deployment / Deployment
  - upgrading / Upgrade

    - monitoring / Monitoring
- Mesos-DNS
    - URL / Mesos-DNS
    - about / Mesos-DNS
    - DNS record generator / Mesos-DNS
    - DNS resolver / Mesos-DNS
    - installing / Installing Mesos-DNS
    - configuration / Mesos-DNS configuration
    - running / Running Mesos-DNS
- Mesos API
    - about / The Mesos API
    - reference / The Mesos API
    - messages / Mesos messages
    - scheduler API / The scheduler API
    - SchedulerDriver API / The SchedulerDriver API
    - executor API / The executor API
    - ExecutorDriver API / The ExecutorDriver API
- Mesos architecture
    - about / The Mesos architecture
    - entities / The Mesos architecture
    - Mesos slave / Mesos slave
    - Mesos master / Mesos master
    - frameworks / Frameworks
    - communication / Communication
    - auxiliary services / Auxiliary services
- Mesosaurus
    - URL / API rate limiting
    - about / API rate limiting
- Mesos build options
    - about / Mesos build options
- Mesos CLI
    - about / The Mesos CLI
- Mesos cluster, on Amazon EC2
    - about / Mesos cluster on Amazon EC2
- Mesos community
    - URL / Deployment
- Mesos containerizer
    - about / Mesos containerizer

    - URL / Mesos containerizer
- Mesos framework
    - developing / Developing a Mesos framework
    - developer environment, setting up / Setting up the development environment
    - framework scheduler, adding / Adding the framework scheduler
    - framework launcher, adding / Adding the framework launcher
    - deploying / Deploying our framework
    - building / Building our framework
    - executor, adding / Adding an executor to our framework
    - framework scheduler, updating / Updating our framework scheduler
    - multiple executors, running / Running multiple executors
- Mesos interfaces
    - about / Mesos interfaces
    - Mesos REST interface / The Mesos REST interface
    - Mesos CLI / The Mesos CLI
- Mesos master
    - about / Mesos master
    - configuration options / The Mesos master
- Mesos master whitelist
    - URL / Master high availability
- Mesos messages
    - MasterInfo / Mesos messages
    - SlaveInfo / Mesos messages
    - Resource / Mesos messages
    - Attribute / Mesos messages
    - Request / Mesos messages
    - FrameworkInfo / Mesos messages
    - CommandInfo / Mesos messages
    - ContainerInfo / Mesos messages
    - DiscoveryInfo / Mesos messages
    - ExecutorInfo / Mesos messages
    - HealthCheck / Mesos messages
    - ResourceUsage / Mesos messages
    - Offer / Mesos messages
    - TaskInfo / Mesos messages
    - TaskStatus / Mesos messages
    - Filters / Mesos messages
- Mesos modules

    - about / Mesos modules
    - URL / Mesos modules
    - authentication modules / Mesos modules
    - isolator modules / Mesos modules
    - anonymous modules / Mesos modules
    - allocator module / Mesos modules
    - hook / Mesos modules
    - module, naming / Module naming
    - module compatibility / Module compatibility
- Mesosphere
    - URL / Mesos cluster on Amazon EC2, Deployment
- Mesos resource or attribute
    - URL / Mesos slave
    - about / Mesos slave
- Mesos REST interface
    - about / The Mesos REST interface
- Mesos scheduler
    - about / The Mesos scheduler
- Mesos slave
    - about / Mesos slave
- Mesos slave options
    - about / The Mesos slave
- Mesos source code
    - URL / Installing Hadoop on Mesos
- Mesos web UI
    - about / Mesos Web UI
- metrics reporting
    - about / Metrics reporting
    - enabling / Metrics reporting
    - CSV / CSV
    - Graphite / Graphite
    - Cassandra / Cassandra
- modern data centers
    - about / Modern data centers
- monit
    - URL / Master high availability
- monitoring
    - about / Monitoring

    - container network monitoring / Container network monitoring
- multi-node Mesos cluster
    - about / Multi-node Mesos clusters
- multitenancy
    - about / Multitenancy
    - authentication / Authorization and authentication
    - authorization / Authorization and authentication
    - API rate limiting / API rate limiting

N

- NFS (Network File System) / Auxiliary services
- NoSQL
  - on Mesos / NoSQL on Mesos

O

- Operator API
  - about / Communication
- optional parameters, Storm-mesos configuration
  - mesos.framework.name / Storm-Mesos configuration
  - mesos.framework.role / Storm-Mesos configuration
  - mesos.framework.checkpoint / Storm-Mesos configuration
  - mesos.allowed.hosts / Storm-Mesos configuration
  - mesos.disallowed.hosts / Storm-Mesos configuration
  - mesos.offer.lru.cache.size / Storm-Mesos configuration
  - mesos.local.file.server.port / Storm-Mesos configuration
  - mesos.master.failover.timeout.secs / Storm-Mesos configuration
  - mesos.supervisor.suicide.inactive.timeout.secs / Storm-Mesos configuration
- oriented architectures (SOA) / Introduction to services
- output operations, Spark Streaming
  - foreachRDD(f) / Spark Streaming
  - print() / Spark Streaming
  - saveAsObjectFiles(prefix, [suffix]) / Spark Streaming
  - saveAsTextFiles(prefix, [suffix]) / Spark Streaming
  - saveAsHadoopFiles(prefix, [suffix]) / Spark Streaming

P

- packaging
  - about / Packaging
- parameters, Storm-mesos configuration
  - mesos.executor.uri / Storm-Mesos configuration
  - mesos.master.url / Storm-Mesos configuration
  - storm.zookeeper.servers / Storm-Mesos configuration
  - nimbus.host / Storm-Mesos configuration
- persistent storage, in Mesos
  - URL / Complex data and the rise of the Lambda architecture
- Platform-as-a-Service (PaaS)
  - about / HubSpot
- prerequisites, Mesos
  - installing / Installing prerequisites
  - CentOS / CentOS
  - Ubuntu / Ubuntu
- principal parameter
  - about / API rate limiting
- Process object
  - about / Aurora job configuration
- Process object, attributes
  - name / Aurora job configuration
  - cmdline / Aurora job configuration
  - max_failures / Aurora job configuration
  - ephemeral / Aurora job configuration
  - min_duration / Aurora job configuration
  - final / Aurora job configuration

Q

- qps parameter
  - about / API rate limiting
- quorum size
  - reference / Master high availability

R

- reconciliation
  - about / Reconciliation
  - offer reconciliation / Reconciliation
  - task reconciliation / Reconciliation
- reconciliation API
  - URL / Reconciliation
- recover command
  - about / External containerizer
- register_frameworks action / Authorization and authentication
- registry
  - about / Registry
- RENDLER
  - about / RENDLER
  - URL / RENDLER
- reservation
  - about / Reservation
  - static reservation / Static reservation
  - dynamic reservation / Dynamic reservation
- Resilient Distributed Datasets (RDD)
  - URL / Introducing Spark
  - about / Introducing Spark
- resource allocation
  - about / Resource allocation
  - Mesos scheduler / The Mesos scheduler
  - weighted DRF / Weighted DRF
  - reservation / Reservation
- resource isolation
  - about / Resource isolation
  - Mesos containerizer / Mesos containerizer
  - Docker containerizer / Docker containerizer
  - external containerizer / External containerizer
- Riak
  - URL / Complex data and the rise of the Lambda architecture
  - about / Complex data and the rise of the Lambda architecture
- Riak on Mesos
  - URL / Complex data and the rise of the Lambda architecture

- runit
  - URL / Master high availability
- run_tasks action / Authorization and authentication

S

- Samza on Mesos
  - URL / Complex data and the rise of the Lambda architecture
- Scheduler API
  - about / Communication
- scheduler API
  - about / The scheduler API
  - void registered(SchedulerDriver, FrameworkID, MasterInfo) method / The scheduler API
  - void reregistered(SchedulerDriver, MasterInfo) method / The scheduler API
  - void disconnected(SchedulerDriver schedulerDriver) method / The scheduler API
  - void resourceOffers(SchedulerDriver, List<Offer>) method / The scheduler API
  - void offerRescinded(SchedulerDriver, OfferID) method / The scheduler API
  - void statusUpdate(SchedulerDriver, TaskStatus) method / The scheduler API
  - void frameworkMessage(SchedulerDriver, ExecutorID, SlaveID, byte[] data) method / The scheduler API
  - void slaveLost(SchedulerDriver, SlaveID) method / The scheduler API
  - void executorLost(SchedulerDriver, ExecutorID, SlaveID, int status) method / The scheduler API
  - void error(SchedulerDriver, String message) method / The scheduler API
- SchedulerDriver API
  - about / The SchedulerDriver API
  - Status start() method / The SchedulerDriver API
  - Status stop(boolean failover) method / The SchedulerDriver API
  - Status abort() method / The SchedulerDriver API
  - Status join() method / The SchedulerDriver API
  - Status run() method / The SchedulerDriver API
  - Status requestResource(Collection<Request>) method / The SchedulerDriver API
  - Status launchTasks(Collection<OfferID> offerIds, Collection<TaskInfo> tasks) method / The SchedulerDriver API
  - Status declineOffer(OfferID offerId, Filters filters) method / The SchedulerDriver API
  - Status reviveOffers() method / The SchedulerDriver API
  - Status sendFrameworkMessage(ExecutorID executorId, SlaveID slaveId,

byte[] data) method / The SchedulerDriver API
    - Status reconcileTasks(Collection<TaskStatus> statuses) method / The SchedulerDriver API
- service discovery
    - about / Service discovery
    - DNS-based discovery / Service discovery
    - Mesos-DNS / Mesos-DNS
- service level agreements (SLAs)
    - about / Cluster computing frameworks
- Service Level Agreements (SLAs) / API rate limiting
- services
    - about / Introduction to services
    - running / Introduction to services
    - Marathon / Marathon
    - Chronos / Chronos
    - Aurora / Aurora
    - packaging / Packaging
- ShreadFilesystem isolator / Mesos containerizer
- shutdown_frameworks action / Authorization and authentication
- Simple Authentication and Security Layer (SASL) library / Installing prerequisites, Authorization and authentication
- single-node Mesos cluster
    - about / Single-node Mesos clusters
- slave recovery
    - about / Slave recovery
- slave removal rate limiting
    - about / Slave removal rate limiting
    - types / Slave removal rate limiting
- Spark
    - URL / Introducing Spark
    - URL, for programming guide / Introducing Spark
    - components / Spark job scheduling
    - URL, for documentation / Tuning Spark on Mesos
- Spark job scheduling
    - about / Spark job scheduling
    - reference / Spark job scheduling
- Spark on Mesos
    - about / Spark on Mesos

    - installing / Spark on Mesos
    - tuning / Tuning Spark on Mesos
    - fine-grained mode / Tuning Spark on Mesos
    - coarse-grain mode / Tuning Spark on Mesos
    - configuration parameters / Tuning Spark on Mesos
- Spark standalone
    - about / Spark Standalone mode
    - URL / Spark Standalone mode
    - installing / Spark Standalone mode
- Spark Streaming
    - about / Spark Streaming
    - URL / Spark Streaming
    - supported operations / Spark Streaming
    - window-based operations / Spark Streaming
    - output operations / Spark Streaming
    - running, on Mesos / Running Spark Streaming on Mesos
    - tuning / Tuning Spark Streaming
- Spark Streaming, tuning
    - batch size, selecting / Selecting the batch size
    - garbage collection / Garbage collection
    - concurrency / Concurrency
    - failures, handling / Handling failures
    - task overheads / Task overheads
- Spark tarball
    - download link / Spark Standalone mode
- stateful applications
    - about / Stateful applications
    - reference / Stateful applications
- storage architecture, Aurora
    - about / Aurora
    - volatile storage / Aurora
    - log manager / Aurora
    - snapshot manager / Aurora
    - backup manager / Aurora
- storage interface, Aurora
    - conssistentRead / Aurora
    - weaklyConsistentRead / Aurora
- Storm-mesos configuration

    - about / Storm-Mesos configuration
    - parameters / Storm-Mesos configuration
    - toplogy.mesos.executor.cpu / Storm-Mesos configuration
    - topology.mesos.worker.mem.mb / Storm-Mesos configuration
    - topology.mesos.executor.cpu / Storm-Mesos configuration
    - topology.mesos.executor.mem.mb / Storm-Mesos configuration
    - optional parameters / Storm-Mesos configuration
- Storm on Mesos
    - URL / Storm on Mesos
- supported operations, Spark Streaming
    - cogroup(stream2, [numTasks]) / Spark Streaming
    - count() / Spark Streaming
    - countByValue() / Spark Streaming
    - filter(f) / Spark Streaming
    - join(stream2, [numTasks]) / Spark Streaming
    - map(f) / Spark Streaming
    - flatMap(f) / Spark Streaming
    - reduce(f) / Spark Streaming
    - reduceByKey(f, [numTasks]) / Spark Streaming
    - repartition(numPartitions) / Spark Streaming
    - transform(f) / Spark Streaming
    - union(stream2) / Spark Streaming
    - updateStateByKey(f) / Spark Streaming

T

- Tachyon
  - URL / Complex data and the rise of the Lambda architecture
- task labels
  - about / Task labels
- task object, attributes
  - Name / Aurora job configuration
  - Processes / Aurora job configuration
  - Resource / Aurora job configuration
  - constraints / Aurora job configuration
  - max_failures / Aurora job configuration
  - max_concurrency / Aurora job configuration
  - finalization_wait / Aurora job configuration
- TaskTrackers
  - about / An introduction to Hadoop
- test frameworks
  - running / Running test frameworks
- Twitter
  - about / Twitter

U

- Ubuntu
  - about / Ubuntu
- UNIQUE operator / Constraints
- Universe, from Mesosphere
  - reference / Upgrade
- UNLIKE operator / Constraints
- update command
  - about / External containerizer
- upgrade
  - about / Upgrade
  - URL / Upgrade
- usage command
  - about / External containerizer
- UTC (Coordinated Universal Time) / A Chronos example

V

- Vagrant
  - used, for running Mesos / Running Mesos using Vagrant
  - URL / Running Mesos using Vagrant

W

- wait command
  - about / External containerizer
- weighted DRF
  - about / Weighted DRF
- window-based operations, Spark Streamin
  - reduceByKeyAndWindow(f, invFunc, windowLength, slideInterval, [numTasks]) / Spark Streaming
- window-based operations, Spark Streaming
  - countByWindow(windowLength, slideInterval) / Spark Streaming
  - countByValueAndWindow(windowLength, slideInterval, [numTasks]) / Spark Streaming
  - window(windowLength, slideInterval) / Spark Streaming
  - reduceByWindow(f, windowLength, slideInterval) / Spark Streaming
  - reduceByKeyAndWindow(f, windowLength, slideInterval, [numTasks]) / Spark Streaming

Z

- ZooKeeper
  - about / ZooKeeper
  - URL / ZooKeeper
  - installing / ZooKeeper
- ZooKeeper Atomic Broadcast (ZAB)
  - about / ZooKeeper
- ZooKeeper documentation
  - URL / Master high availability